Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ИНСТРУМЕНТЫ И СРЕДСТВА ПРОГРАММИРОВАНИЯ

Курс лекций
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2006

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью курса «Инструменты и средства программирования» является получение студентами теоретических знаний и практических навыков создания качественного программного обеспечения с использованием современных средств быстрой разработки приложений.

Практически любой современный программный продукт построен с использованием объектно-ориентированного программирования. Первая часть курса рассматривает основные понятия ООП, применяя в качестве языковой основы Object Pascal и систему Delphi. Затрагиваются вопросы создания компонент Delphi. Полученные знания закрепляются и развиваются во второй части курса, посвященной изучению .NET Framework и языка Visual Basic .NET.

В середине 2000 года корпорация Microsoft представила новую модель для создания приложений, основой которой является платформа .NET[1] (.NET Framework). Платформа .NET образует каркас, который включает технологии разработки Windows-приложений, Web-приложений и Web-сервисов, технологии доступа к данным и межпрограммного взаимодействия. 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. Одним из языков программирования для .NET является язык Visual Basic .NET. Этот язык сочетает простой синтаксис 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.

Курс содержит фрагменты кода и небольшие программы, иллюстрирующие теоретический материал. Примеры могут служить основой при написании лабораторных работ, связанных с объектно-ориентированным программированием с использованием Visual Basic .NET.

[1] Произносится как «дот-нэт».

ЧАСТЬ I. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ OBJECT PASCAL.

1. Объектно-ориентированное программирование. Синтаксис определения класса.

При разработке программ возможно использование нескольких подходов. Императивное[1] программирование подразумевает разработку программы методом последовательного уточнения и детализации. Исходная задача разбивается на несколько связанных подзадач, затем каждая подзадача делится на некоторые более мелкие. Данный процесс продолжается вплоть до элементарных команд («присвой значение переменной», «выполни функцию» – отсюда и название стиля программирования). Однако усложнение программных продуктов потребовало новой методики написания программ. Выход был найден, когда в программировании применили принципы реального мира. В повседневной жизни мы сталкиваемся с множеством объектов («автомобиль», «дом», «улица»), наблюдаем их взаимодействие, и сами взаимодействуем с ними. *Объектно-ориентированное программирование* (*ООП*) предполагает представление программы как процесса взаимодействия некоторых вполне самостоятельных единиц, по аналогии с реальным миром называемых *объектами*. Работа программиста в этом случае заключается в наиболее полном описании набора объектов задачи и кодировании их связей. Подобный стиль программирования позволяет существенно сократить время разработки сложных программ и программных комплексов. Упрощается также сопровождение и отладка программ. ООП оказалось настолько продуктивной идеей, что большинство современных языков программирования являются либо чисто объектно-ориентированными (Java, C#, Visual Basic.NET), либо содержат средства ООП в качестве надстройки (C++, Object Pascal).

ООП основывается на трех принципах: *инкапсуляция*, *наследование*, *полиморфизм*. Рассмотрим первый из них. *Инкапсуляция* – это логическое объединение в одном программном типе, называемом *класс*, как данных, так и подпрограмм для их обработки. Данные класса хранятся в *полях класса*, подпрограммы для работы с полями называются *методами класса*.

Рассмотрим синтаксис определения класса в Object Pascal. Вот пример класса для представления информации о человеке:

```
type TPerson = class
       fName: string;
       fAge: Integer;
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;
```

Описание класса размещается в секции описания типов. Это может быть глобальная секция типов программы, секция типов в интерфейсной части или в

[1] Императив – в грамматике: то же, что повелительное наклонение.

разделе реализации модуля, но не секция типов в подпрограмме. Для определения класса используется ключевое слово `class`. Вначале описываются поля класса. Полями класса `TPerson` являются `fName` и `fAge`[1]. После описания всех полей следует указание методов класса. Методы `TPerson` это процедура `SetAge` и функция `SayName`. Класс содержит только заголовки методов, реализация методов описывается отдельно. Если класс описан в модуле, реализация методов должна находиться в том же модуле в секции `implementation`. Для указания того, что подпрограмма является реализацией метода класса, используется синтаксис `<имя класса>.<имя метода>`. При реализации методов обращение к полям класса происходит без указания каких-либо дополнительных спецификаторов:

```
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;
function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;
```

После того как класс описан, можно объявить переменную класса, называемую *экземпляром класса* или *объектом*:

```
var Man: TPerson;
```

В реальной жизни классу соответствует абстрактное понятие, которое конкретизируется в своих проявлениях. Так, класс `TPerson` как бы пытается описать человека вообще, его параметры и действия, производимые им и над ним. Объект `Man` – это уже конкретный человек, и конкретизировать его можно через значения полей.

В Object Pascal для работы с полями и методами объекта используется синтаксис вида `<имя объекта>.<имя поля или метода>`. Это напоминает работу с записью:

```
Man.fName := 'John Dow';
Man.SetAge(35);
writeln(Man.SayName, Man.fAge);
```

Приведем полный пример консольного приложения, содержащего определение класса и работу с объектами этого класса:

```
program MyFirstOOPProgramm;
{$APPTYPE CONSOLE}
type TPerson = class
       fName: string;
       fAge: Integer;
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;
```

[1] Для имен полей в Object Pascal традиционно используется префикс "f" (field – поле).

```
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;
function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;
var Man: TPerson;
begin
  //инициализация объекта, не рассматривалась
  Man := TPerson.Create;
  Man.fName := 'John Dow';
  Man.SetAge(35);
  writeln(Man.SayName, Man.fAge);
end.
```

Вопросы и упражнения.

1. Если попытаться описать как класс понятие «автомобиль», какие поля и методы вы смогли бы выделить? Проделайте то же с понятиями «дом», «комплексное число», «числовая матрица».

2. Отнесите ниже перечисленные понятия к одной из четырех категорий: класс, объект, набор объектов, набор классов.

Автомобиль, автомобиль «Жигули», мой автомобиль, все марки автомобилей, выпускаемые в Германии, все автомобили Германии.

3. Опишите на Object Pascal класс для представления комплексных чисел.

2. Инициализация объектов. Конструктор, деструктор.

Object Pascal использует так называемую *ссылочную объектную модель*[1]. Это означает, что все объекты размещаются в памяти динамически, объектные переменные фактически являются указателями на данные объекта в динамической памяти и имеют одинаковый размер (4 байта). Однако для доступа к данным объекта не используется разыменователь ^ (мы записываем `Man.fName`, а не `Man^.fName`, хотя подразумевается именно второе). Для начального размещения объектов в динамической памяти служит особый вид методов, называемых *конструкторами*.

Чтобы объявить метод как конструктор, используется ключевое слово `constructor`. Оно записывается вместо слова `procedure` в объявлении класса и при реализации метода (конструктор не может быть функцией). Компилятор автоматически добавляет к телу конструктора некий код, выделяющий участок в динамической памяти для полей объекта и «очищающий» этот участок. Так как конструктор необходимо выполнить перед использованием объекта, то в тело конструктора обычно помещают операторы инициализации объекта, например, задание начальных значений для полей. Отметим, что Object Pascal допускает существование в классе нескольких конструкторов. Традиционное имя для конструктора – `Create`.

[1] *Объектная модель* – правила реализации общих концепций ООП в конкретном языке программирования.

Добавим конструктор в класс TPerson:

```
type TPerson = class
       . . .
       constructor Create;
     end;
. . .
constructor TPerson.Create;
begin
  fAge := 1;
  fName := ‘Person’;
end;
```

Синтаксис вызова конструктора: <объектная переменная> := <имя класса>.<имя конструктора>:

```
Man := TPerson.Create;
```

Как метод, конструктор можно вызывать в виде <имя объекта>.<имя конструктора>:

```
Man.Create;
```

Подобный вызов означает просто выполнение тела конструктора (реинициализацию полей). Его можно применять только для тех объектов, которые уже размещены в памяти.

Если объект не используется, то занимаемая им динамическая память должна быть освобождена. Для уничтожения объектов предназначены особые методы – *деструкторы*.

Для объявления деструкторов используется ключевое слово destructor. Тело деструктора это подходящее место для финальных действий с объектом. Традиционное имя для деструктора – Desrtoy.

Добавим деструктор в класс TPerson:

```
type TPerson = class
       . . .
       constructor Create;
       destructor Destroy;
     end;
. . .
destructor TPerson.Destroy;
begin
  fAge := 0;
  fName := ‘’;
end;
```

Теперь полный цикл работы с объектом Man выглядит следующим образом:

```
Man := TPerson.Create;   //создание объекта
Man.fName := 'John Dow'; //работа с объектом
Man.Destroy;             //уничтожение объекта
```

Деструктор можно вызвать только у инициализированного объекта. Попытка вызвать деструктор у неинициализированного объекта может привести к исключительной ситуации в работе программы.

В качестве примера работы с конструкторами и деструкторами рассмотрим класс для представления структуры данных «бинарное дерево целых чисел». Бинарное дерево целых чисел хранит в каждом узле некий целочисленный вес, а так же ссылки (возможно пустые) на правое и левое поддерево. Реализуем в классе единственный метод для подсчета веса всего дерева.

```
type TBTree = class
         Weight: Integer;
         Left, Right: TBTree;
         function GetWeight: Integer;
     end;
function TBTree.GetWeight;
begin
  Result := Weight;
  if Left <> nil then inc(Result, Left.GetWeight);
  if Right <> nil then inc(Result, Right.GetWeight);
end;
```

Наделим наш класс двумя конструкторами. Первый будет устанавливать вес узла, второй кроме этого будет инициализировать левое и правое поддеревья. Хотя эти два конструктора можно назвать по-разному, дадим им одинаковые имена и воспользуемся возможностью перегрузки подпрограмм:

```
type TBTree = class
       . . .
       constructor Create(W: Integer); overload;
       constructor Create(W: Integer; LTree, RTree: TBTree);
                                                    overload;
     end;
constructor TBTree.Create(W: Integer);
begin
  Weight := W;
end;
constructor TBTree.Create(W: Integer; LTree, RTree: TBTree);
begin
  Weight := W;
  Left := LTree;
  Right := RTree;
end;
```

Работа с нашим классом может выглядеть так:

```
var T: TBTree;
. . .
T := TBTree.Create(10);
T.Left := TBTree.Create(20);
T.Right := TBTree.Create(100,
                         TBTree.Create(1000), TBTree.Create(5));
Y := T.GetWeight; //10+20+100+1000+5
```

Добавим в класс `TBTree` деструктор. Одно из правил ООП гласит: «Если объект во время работы самостоятельно резервировал динамическую память,

она должна быть освобождена деструктором класса». Деструктор для класса TBTree будет выглядеть так:

```
type TBTree = class
       . . .
       destructor Destroy;
     end;
destructor TBTree.Destroy;
begin
  if Left <> nil then Left.Destroy;
  if Right <> nil then Right.Destroy;
end;
```

Полный пример консольного приложения с классом TBTree:

```
program BinaryTree;
{$APPTYPE CONSOLE}
type TBTree = class
       Weight: Integer;
       Left, Right: TBTree;
       constructor Create(W: Integer); overload;
       constructor Create(W: Integer; LTree, RTree: TBTree);
                                                        overload;
       function GetWeight: Integer;
       destructor Destroy;
     end;
constructor TBTree.Create(W: Integer); overload;
begin
  Weight := W;
end;
constructor TBTree.Create(W: Integer; LTree, RTree: TBTree);
                                                        overload;
begin
  Weight := W;
  Left := LTree;
  Right := RTree;
end;
function TBTree.GetWeight;
begin
  Result := Weight;
  if Left <> nil then inc(Result, Left.GetWeight);
  if Right <> nil then inc(Result, Right.GetWeight);
end;
destructor TBTree.Destroy;
begin
  if Left <> nil then Left.Destroy;
  if Right <> nil then Right.Destroy;
end;
var T: TBTree;
begin
  T := TBTree.Create(10);
  T.Left := TBTree.Create(20);
  T.Right :=
```

```
        TBTree.Create(100,TBTree.Create(1000),TBTree.Create(5));
  writeln(T.GetWeight); //10+20+100+1000+5
  T.Destroy;
end.
```

Вопросы и упражнения.

1. Может ли из тела одного конструктора класса вызываться другой конструктор класса?

2. Являются ли имена `Create` и `Destroy` обязательными?

3. Опишите класс для представления структуры данных «связный список», использовав в классе конструктор и деструктор.

3. Параметр self.

Вернемся к рассмотрению класса `TPerson`. Этот класс содержит два поля и два метода. Представим, что у нас имеется 100 объектов этого класса. Так как каждый объект конкретизируется значениями своих полей, то в памяти должно содержаться 100 наборов полей класса. Означает ли это, что и код методов класса будет продублирован 100 раз? Определенно, нет. Код методов класса содержится в памяти в единственном экземпляре, как и код любой подпрограммы. Однако как метод определяет, с полями какого объекта он работает?

```
var A, B: TPerson;
. . .
A.SetAge(10); //SetAge работает с A.fAge
B.SetAge(40); //SetAge работает с B.fAge
```

Для выявления конкретного объекта с которым происходит работа любому методу передается *скрытый параметр* `self`. Этот параметр указывает на объект, вызывающий метод. Тип параметра `self` совпадает с типом класса. Например, на уровне компилятора описание метода `SetAge` и работу с ним можно представить следующим образом:

```
procedure TPerson.SetAge(Age: Integer; self: TPerson);
begin
  if Age > 0 then self.fAge := Age
end;
. . .
TPerson.SetAge(10, A);
TPerson.SetAge(40, B);
```

Подчеркнем два важных момента, касающихся методов:

1. параметр `self` передается в любой метод;

2. методы можно воспринимать как обыкновенные подпрограммы, которые принимают дополнительный параметр `self`.

Практически всегда в явном использовании `self` нет необходимости. Одно из исключений – использование одинаковых идентификаторов для полей класса и параметров метода. Предположим, что класс `TPerson` содержит поле с идентификатором `Age`, а не `fAge`. Тогда корректная реализация метода `TPerson.SetAge` должна выглядеть следующим образом:

```
procedure TPerson.SetAge(Age: Integer);
```

```
begin
//Age – параметр метода, self.Age – поле объекта
  if Age > 0 then self.Age := Age
end;
```

Следующий код показывает несколько нетривиальный пример использования `self`. Перепишем метод `GetWeight` и деструктор `Destroy` из класса `TBTree`:

```
function TBTree.GetWeight;
begin
  if self = nil then Result := 0
  else Result := Weight + Left.GetWeight + Right.GetWeight
end;
destructor TBTree.Destroy;
begin
  if self <> nil then begin
                        Left.Destroy;
                        Right.Destroy;
                      end
end;
```

В деструкторе `Destroy` проверяется при помощи `self` является ли объект инициализированным. Если объект инициализирован, вызываются деструкторы для левого и правого поддеревьев.

Вопросы и упражнения.

1. Проанализируйте следующий листинг. Объясните, почему данный код будет выполняться, несмотря на отсутствие вызова конструктора.

```
type TDummyClass = class
       procedure PrintMe;
     end;
procedure TDummyClass.PrintMe;
begin
  writeln('Hello!')
end;
var X: TDummyClass;
begin
  X.PrintMe
end.
```

4. Ограничение видимости членов класса.

Вернемся к рассмотрению класса `TPerson`. Данный класс содержит четыре *члена класса* – два поля и два метода. И поля, и методы свободно доступны из любого объекта:

```
var Man: TPerson;
. . .
Man.SetAge(35);   //работа с полем через метод
Man.fAge := -100; //работа с полем непосредственно
```

Приведенный пример показывает недостаток подобного «раздолья». Одно из назначений метода `TPerson.SetAge` – *корректная* установка возраста. В то же время в любой момент этот метод можно обойти, установив возраст непосредственно через поле.

В Object Pascal существуют специальные *директивы ограничения видимости*, которые при описании класса позволяют контролировать видимость его членов. Эти директивы разделяют объявление класса на *секции видимости*.

Рассмотрим две директивы – `private` и `public`. Все члены класса из секции видимости `private` доступны для использования только в том модуле или программе, которые содержат объявление класса. В этой секции обычно размещаются поля и методы, описывающие внутренние особенности реализации класса. Элементы из секции `public` не имеет ограничений на использование.

Подчеркнем следующие особенности использования директив. Во-первых, внутри модуля с описанием класса директивы ограничения видимости не действуют. Во-вторых, по умолчанию для членов класса установлена директива видимости `public`. И, наконец, порядок директив в описании класса произволен, допускается повторение директив. Однако принято описывать члены класса в порядке увеличения их видимости.

Применим данные директивы к классу `TPerson`, поместив его в отдельном модуле. Обратите внимание, в какие секции модуля помещено описание класса и реализация методов:

```
unit TPersonClass;
interface
type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;

implementation
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;
function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;
end.
```

Теперь попытка обратится к полю `fAge` в программе, использующей класс `TPerson`, вызовет ошибку компиляции:

```
program MyOOPProgramm;
uses TPPersonClass
var Man: TPerson;
```

```
. . .
Man.fAge := -100; //ошибка компиляции!
```

Таким образом, при помощи директив мы защитили поля класса от «грубого вторжения» пользователей. Однако у TPerson появился следующий недостаток – теперь значения полей нельзя непосредственно прочитать. Этот недостаток устраняется введением в класс дополнительных методов. В общем случае в ООП принято осуществлять доступ к полям через методы класса. Говорят, что такие методы составляют *интерфейс класса*.

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function GetAge: Integer;
       procedure SetName(Name: string);
       function SayName: string;
     end;
```

Обсудим преимущества, которые дает использование методов для доступа к полям. Первое преимущество: при использовании методов данные объекта могут быть представлены в разных форматах без дублирования. Для иллюстрации рассмотрим простой класс TTemp, назначение которого – хранить данные о температуре. В классе будет использоваться две пары методов для чтения и записи значения поля, что позволит получать температуру в двух вариантах – градусах Цельсия и кельвинах:

```
type TTemp = class
     private
       fTemp: double;
     public
       procedure SetTempCelsius(Temp: double);
       function GetTempCelsius: double;
       procedure SetTempKelvin(Temp: double);
       function GetTempKelvin: double;
     end;
. . .
procedure TTemp.SetTempCelsius;
begin
  fTemp := Temp + 273.15
end;
function TTemp.GetTempCelsius;
begin
  Result := Temp - 273.15
end;
procedure TTemp.SetTempKelvin;
begin
  fTemp := Temp
end;
function TTemp.GetTempKelvin;
```

```
begin
  Result := Temp
end;
. . .
var Temperature: TTemp;
. . .
Temperature.SetTempCelsius(20); //установил в градусах Цельсия
A := Temperature.GetTempKelvin; //а сколько это в Кельвинах?
```

Второе преимущество использования методов для доступа к полям класса: разработчик класса может незаметно для пользователей изменять структуру хранения данных класса. Например, в классе `TTemp` мы могли бы хранить температуру в градусах Цельсия. Для этого нам пришлось бы переписать методы класса. Но если оставить их заголовки прежними, пользователь подобной замены не заметит.

Подводя итог, можно сформулировать следующее правило: **создатель класса должен исключить возможность прямой работы с полями класса, для работы с данными класса следует использовать набор интерфейсных методов.**

Вопросы и упражнения.

1. Хотя в секции `private` обычно размещаются поля, там могут находиться и методы. Как вы можете их охарактеризовать, для чего могут быть нужны такие методы?

2. Опишите класс для представления даты и времени. Какие внутренние форматы для данных класса можно использовать? Какими интерфейсными методами вы наделите класс?

3. Структура данных «стек целых чисел» характеризуется методами для помещения и извлечения числа в стек и проверки стека на пустоту. В каком формате могут храниться данные класса, представляющего такую структуру? Предложите 2–3 варианта реализации такого класса.

5. Свойства.

В предыдущей главе обсуждались преимущества, которые дают интерфейсные методы класса. Однако с точки зрения пользователя класса применение методов для доступа к полям имеет небольшой недостаток – громоздкость вызова. Развитие концепций ООП привело к появлению понятия *свойства* (property). Свойство – это член класса, работа с которым происходит так же, как с полем объекта. Разница между полем и свойством заключается в следующем: обращение к свойству компилятор транслирует в вызов метода или обращение к полю, следовательно, при работе со свойствами могут выполняться некоторые действия.

Для объявления свойства используется ключевое слово `property`. Далее следует имя свойства и указывается его тип. После директивы `read` указывается имя поля или метода для чтения свойства. Аналогично, после директивы `write` указывается имя поля или метода для записи свойства. Считается, что свойство записывается, когда ему присваивается некое значение, в противном

случае свойство читается. Тип полей, используемых после `read` и `write`, должен совпадать с типом свойства. Метод, используемый для чтения простого свойства, должен быть функцией без параметров, тип возвращаемого значения которой совпадает с типом свойства. Метод для записи – процедура с одним параметром, имеющим тип свойства. Принято соглашение, согласно которому имена методов чтения свойств начинаются с префикса `Get`, а имена методов записи – с префикса `Set`. Объявление свойства может следовать только после объявления полей и методов, которые используются свойством.

Добавим свойства в класс `TPerson`:

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function GetAge: Integer;
       procedure SetName(Name: string);
       function SayName: string;
       property Age: Integer read GetAge write SetAge;
       property Name: string read SayName write SetName;
     end;
```

Работу со свойствами демонстрирует следующий фрагмент программы:

```
var Man: TPerson;
. . .
Man.Name := 'Alex'; //оттранслируется в Man.SetName('Alex')
Man.Age := 101; //оттранслируется в Man.SetAge(101)
```

Подчеркнем, что свойства класса призваны облегчить работу **пользователя** с классом. Фактически, свойства «живут» только до компиляции программы, во время которой заменяются методами или полями. В отличие от полей свойства не занимают места в памяти. Это накладывает определенные ограничения на их использование. Свойства нельзя передавать в качестве `var`-параметров в подпрограммы, к ним нельзя применить операцию взятия адреса.

Применение свойств является «хорошим тоном» ООП. В пользу свойств говорит тот факт, что в объектной библиотеке VCL весь доступ к полям классов и компонент организован через свойства.

Употребление свойств позволяет «сэкономить» на методах класса. Например, в классе `TPerson` методы `GetAge` и `SetName` введены только для того, чтобы обеспечить полный доступ к полям, никаких особых дополнительных действий они не выполняют. Заменим в определении свойств эти методы полями `fAge` и `fName`. В свою очередь, методы, обслуживающие свойства, поместим в секцию `private`, так как именно свойства теперь будут составлять интерфейс класса:

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
```

```
    procedure SetAge(Age: Integer);
    function SayName: string;
  public
    property Age: Integer read fAge write SetAge;
    property Name: string read SayName write fName;
  end;
```

Рассмотрим некоторые нюансы описания и использования свойств. Если опустить директиву `read`, можно получить свойство только для записи. Если опустить `write`, получим свойство, значение которого можно читать, но не записывать. Однако какая-то из директив должна в объявлении свойства присутствовать.

Одна пара методов может использоваться для обслуживания нескольких свойств. Чтобы различить в теле метода, к какому свойству относится его вызов, свойству присваивается *индекс*, само свойство в этом случае называется *индексированным*:

```
type TExample = class
    . . .
    function Get(A: Integer): byte;
    procedure Set(B: Integer; C: byte);
    . . .
    property Prop1: byte index 0 read Get write Set;
    property Prop2: byte index 1 read Get write Set;
    property Prop3: byte index 2 read Get write Set;
  end;
```

`Prop1`, `Prop2`, `Prop3` – индексированные свойства. Для создания таких свойств используется директива `index` с уникальной целой константой. Для чтения всех индексированных свойств применяется метод-функция с одним целым параметром. Запись индексированных свойств осуществляется методом-процедурой, первый параметр которого – целое число. Применение полей для чтения и записи индексированных свойств невозможно.

```
var Ex: TExample;
. . .
Ex.Prop1 := 100; //Ex.Set(0,100)
k := Ex.Prop3; //k := Ex.Get(2)
```

Рассмотрим пример класса `TArray`. Он будет представлять массив вещественных чисел, в котором кроме самих чисел хранится количество элементов, а также имеется свойство для получения максимального элемента. Начальный вариант такого класса может выглядеть так:

```
type TArray = class
  private
    fData: array[1..1000] of double; //массив «с запасом»
    fLength: Integer; //реальная длина массива
  public
    procedure WriteElement(Ind: Integer; Value: double);
    function ReadElement(Ind: Integer): double;
    property Length: Integer read fLength;
    property Max: double read FindMax;
```

```
    end;
procedure TArray.WriteElement(Ind: Integer; Value: double);
begin
  if (Ind > 0) and (Ind <= 1000) then fData[Ind] := Value;
  if Ind > fLength then fLength := Ind;
end;
function TArray.ReadElement(Ind: Integer): double;
begin
  if (Ind > 0) and (Ind <= fLength) then Result := fData[Ind]
  else Result := 0
end;
function TArray.FindMax: double;
var i: Integer;
begin
  Result := fData[1];
  for i := 2 to fLength do
    if fData[i] > Result then Result := fData[i]
end;
. . .
var Mas: TArray;
. . .
Mas.WriteElement(1, 10);
Mas.WriteElement(2, 100);
Mas.WriteElement(10, 1);
k := Mas.Max; // k=100
l := Mas.Length; // l=10
```

Обратите внимание: свойства `Length` и `Max` являются свойствами только для чтения. Более того, свойство `Max` вообще можно считать «виртуальным», так как оно не связано не с одним полем класса (но пользователь класса этого не заметит).

Для пользователя класса `TArray` естественным желанием является получить более удобный способ доступа к данным в `fData`. Object Pascal разрешает объявлять так называемые *свойства-массивы*, представляющие индексированное множество свойств. Подобные свойства используются в основном в классах, данные которых подразумевают доступ с использованием различных индексаторов. Добавим свойство-массив в класс `TArray` (заодно перенесем методы `WriteElement` и `ReadElement` в секцию `private`):

```
type TArray = class
     private
     . . .
       procedure WriteElement(Ind: Integer; Value: double);
       function ReadElement(Ind: Integer): double;
     public
     . . .
       property Data[I: Integer]: double read ReadElement
                                          write WriteElement;
     end;
```

Для объявления свойства-массива после имени свойства в квадратных скобках указывается имя и тип индекса. Как и для индексированных свойств,

для доступа к свойствам-массивам возможно только использование методов. Первый параметр этих методов должен совпадать по типу с индексом свойства. Ниже приведен пример работы со свойством Data и указано, во что транслируются обращения к нему:

```
for i := 1 to 5 do
  Mas.Data[i]:=1000;//транслируются в Mas.WriteElement(i,1000)
```

Тип индекса свойства-массива не ограничен диапазоном (индекс может быть строкового или вещественного типа). Допускается работа только с элементами свойства-массива, а не со всем свойством целиком.

Свойства-массивы могут быть многомерными. В этом случае количество необходимых параметров у методов чтения и записи увеличивается на соответствующее число.

Если при описании свойства-массива добавить в конце описания директиву default, то такое свойство становится *основным свойством класса*. Для основного свойства можно указывать индекс непосредственно после имени объекта, не используя идентификатор свойства. Сделаем свойство Data основным:

```
type TArray = class
     . . .
       property Data[I: Integer]: double read ReadElement
                                  write WriteElement; default;
     end;
```

Теперь с ним можно работать так:

```
for i := 1 to 5 do
  Mas[i] := 1000; // вместо Mas.Data[i] := 1000
```

Только свойство-массив может быть основным свойствам класса. У класса может быть только одно основное свойство.

Вопросы и упражнения.

1. Приведите пример класса, в котором можно было бы использовать индексированные свойства.

2. Завершите описание класса TArray, добавив к нему конструктор и поместив его в отдельном модуле. Какие еще члены класса можно добавить в TArray?

3. При обращении к свойству Max класса TArray каждый раз производиться поиск максимального элемента в массиве. Предложите модификацию класса, в которой устранен данный недостаток.

4. В программе встретилась строка Obj.Prop[2,'Alex',0.3] := 1. Возможно ли такое? Если да, какое объявление может соответствовать свойству Prop и во что транслируется обращение к нему.

5. Создайте класс, в котором используется свойство-массив со строковым типом индекса. Для чего может использоваться такой класс?

6. Отношения между классами. Наследование.

Если предположить, что классы и объекты в ООП являются отражением понятий реальной жизни, то легко можно установить те отношения, которые могут существовать между классами. Первый тип отношений соответствует ситуации, когда один класс включает в себя объекты других классов. Такой тип отношений называется *агрегированием* (иначе называемым *отношением has-a* или *отношением part-of*). Данный тип отношения моделируется включением в класс полей-объектов. Например, класс для представления группы людей может иметь следующее описание:

```
type TCrowd = class
       . . .
       fPeople: array[1..100] of TPerson;
```

Для классов, реализующих агрегирование, конструктор, как правило, занимается созданием объектов-полей, деструктор уничтожает эти объекты:

```
constructor TCrowd.Create;
var i: Integer;
begin
  for i := 1 to 100 do
    fPeople[i] := TPerson.Create
end;
```

Следующий тип отношений связан с ситуацией, когда понятие, соответствующее одному классу, уточняется понятием, соответствующим другому классу. Пусть нам необходим класс для описания служащих – `TEmployee`. Мы можем рассуждать так: любой служащий является человеком (`TPerson`), но служащий – это такой человек, который получает зарплату. Отношение между классами `TEmployee` и `TPerson` называется *наследованием* (*отношение is-a*[1]). Наследование является одним из базовых принципов ООП. Наследование предполагает создание новых классов на основе существующих. В нашем случае мы можем не писать класс `TEmployee` «с нуля», а воспользоваться классом `TPerson` как основой. При наследовании новый класс называется *классом-потомком* (или *дочерним классом, производным классом*), старый – *классом-предком* (или *родительским классом, базовым классом*). При помощи наследования можно строить так называемое *дерево классов* (или *иерархию классов*), последовательно уточняя описание класса и переходя от абстрактных понятий к частным.

Рассмотрим синтаксис описания наследования классов. При описании класса-потомка имя класса-предка указывается после ключевого слова `class` в круглых скобках. Само описание класса-потомка включает только те члены, которых нет в предке. Наследник включает все члены предка автоматически.

Описание класса `TEmployee` может выглядеть следующим образом:

```
type TEmployee = class(TPerson)
     private
       fSalary: double;
       procedure SetSalary(Value: double);
```

[1] По-английски: *Every Employee* ***is a*** *Person*.

```
public
  property Salary: double read fSalary write SetSalary;
end;
```

Как наследник, TEmployee содержит все поля, методы и свойства TPerson и, кроме этого, добавляет собственное поле fSalary, метод SetSalary и свойство Salary.

Объекты классов-потомков совместимы по присваиванию с объектами классов-предков. При этом действует следующее правило: **объекту родительского класса можно присвоить объект дочернего класса, но не наоборот**:

```
var Man: TPerson;
    Employee: TEmployee;
. . .
Man  := Employee; //допустимо
Employee := Man;  //ошибка компиляции
```

Обосновывается вышеуказанное правило следующим образом. Так как дочерний класс может добавлять к родительскому новые поля, то при присваивании объекту дочернего класса объекта родительского класса некоторые из полей, возможно, не будут инициализированы. Это является недопустимым.

Отметим следующие особенности объектной модели языка Object Pascal. Наследование в Object Pascal разрешено только от одного предка. Все классы в Object Pascal имеют одного общего предка. Таким предком является класс TObject. Объявление вида TPerson = class и TPerson = class(TObject) полностью эквивалентны. Класс TObject подробнее будет рассмотрен ниже. Пока же заметим, что TObject содержит пустой конструктор Create и пустой деструктор Destroy, которые можно использовать в простых классах (чем мы и пользовались в некоторых примерах).

С наследованием связана директива ограничения видимости protected. Члены класса из секции protected могут использоваться вне пределов модуля с объявлением класса, но только потомками класса.

Вопросы и упражнения.

1. В каком отношении находятся классы, соответствующие следующим понятиям: «человек», «солдат», «оружие», «автомат», «рота», «полк», «командир», «командир роты», «командир полка», «оружейный склад».

2. Опишите наследника класса TArray, добавив туда свойство для минимального элемента и процедуру случайного заполнения массива данными.

3. Пусть класс TAncestor содержится в модуле UnitA, а его наследник TDescendant – в модуле UnitB. Класс TAncestor содержит поле Field и метод Method. При каких условиях возможна работа с ними в классе TDescendant?

4. В Object Pascal используется ссылочная объектная модель. В связи с этим присваивание объекту дочернего класса объекта родительского класса можно осуществить, использовав операции с указателями и приведение типов. Проиллюстрируйте этот факт на примере. К каким проблемам может привести подобное присваивание?

7. Перекрытие методов.

Достаточно типичной ситуацией при проектировании иерархии классов является следующая: наследник выполняет такие же действия как предок, но своим особым образом. Рассмотрим класс для представления животных способных издавать звуки:

```
type TPet = class
       procedure Speak;
     end;
. . .
procedure TPet.Speak;
begin
  Beep; //ничего особенного – просто сигнал
end;
```

Потомки `TPet` – классы `TDog` и `TCat` – тоже способны издавать звуки, но делают это по-своему. ООП предоставляет возможность описать в классе-потомке метод с тем же именем, что и в классе-предке, но с собственной реализацией. Подобная возможность называется *перекрытием (замещением) методов*:

```
type TDog = class(TPet)
       procedure Speak; //перекрытие метода TPet.Speak
     end;
     TCat = class(TPet)
       procedure Speak; //перекрытие метода TPet.Speak
     end;
. . .
procedure TDog.Speak;
begin
  writeln('Bowwow');
end;
procedure TCat.Speak;
begin
  writeln('Mew');
end;
```

При перекрытии методов их сигнатуры (то есть количество и тип параметров) могут различаться.

Если дочерний класс не перекрывает метод предка, а добавляет новый метод с тем же именем, необходимо воспользоваться директивой `overload`, чтобы явно указать на перегрузку. Естественно, такие методы должны различаться сигнатурой:

```
type TCatWithVolume = class(TCat)
       procedure Speak(Volume: Integer);overload;
       //мы не перекрыли TCat.Speak, а добавили новый метод
     end;
```

Рассмотрим вопрос с перекрытием полей классов. Классическое ООП не допускает подобного. Однако в Object Pascal дочерний класс может объявить поле с тем же именем, что и поле в родительском классе, но с другим типом. В

этом случае методы в дочернем классе будут работать с новым полем, методы в родительском классе – со старым. На практике такая возможность практически не используется, так как способна основательно запутать проектировщика и пользователей класса.

Достаточно часто метод класса-потомка не заменяет действия метода класса-предка, а дополняет их. Чтобы не дублировать код, в методе класса-потомка можно вызвать метод класса-предка. Для вызова перекрытых методов ближайшего класса-предка применяется конструкция `inherited <имя метода класса-предка>`. Если имя и параметры вызываемого у предка метода совпадают с именем и параметрами вызывающего, достаточно записать только ключевое слово `inherited`:

```
procedure TCat.Speak;
begin
  inherited; //вызов TPet.Speak – звуковой сигнал
  writeln('Mew');
end;
```

Если в непосредственном предке метод, вызываемый через `inherited`, отсутствует, компилятор проводит поиск такого метода по иерархии классов вплоть до корневого класса. Если метод не найден, никаких действий не предпринимается.

В иерархии классов работа конструктора класса-потомка, как правило, начинается с вызова конструктора класса-предка для корректной инициализации полей предка. Добавим конструктор в класс `TEmployee` (используется вариант класса `TPerson` с конструктором из главы 2):

```
type TEmployee = class(TPerson)
     public
     . . .
       constructor Create;
     end;
constructor TEmployee.Create;
begin
  inherited; //можно было написать inherited Create
             //fAge := 1, fName := 'Person'
  fName := 'Employee';
end;
```

Для деструкторов в иерархии классов действует правило, согласно которому вызов унаследованного деструктора происходит в конце работы класса-потомка.

Вопросы и упражнения.

1. Создайте иерархию классов, в которой присутствует перекрытие полей. Покажите, какие проблемы возникают при использовании таких классов.

2. Придумайте иерархию классов, в которой присутствовало бы перекрытие методов, причем перекрываемые методы различались бы сигнатурой.

3. Квадратная матрица целых чисел характеризуется своим размером n и хранит n^2 элементов. Диагональная матрица размера n хранит n элементов

главной диагонали, остальные равны нулю. Создайте на основе этих утверждений иерархию классов, включив в классы конструкторы и деструкторы. Используйте свойства для доступа к данным классов. Предложите несколько вариантов для хранения данных в классе.

4. Обоснуйте, почему вызов унаследованного конструктора обычно происходит *в начале работы* конструктора, а вызов унаследованного деструктора – *в конце работы* деструктора. Покажите на примерах, какие проблемы могут возникнуть, если не следовать этому правилу.

8. Полиморфизм.

Полиморфизм является одним из трех принципов ООП. Прежде чем дать точное определение полиморфизма и рассмотреть синтаксис его реализации в Object Pascal, изучим ситуации, приводящие к понятию полиморфизма.

Рассмотрим простую иерархию классов для графических объектов – базовый класс `TFigure` и его наследники `TSquare` и `TCircle`. Нас интересуют некоторые методы этих классов. Наделим `TFigure` методом рисования `Draw` и методом стирания фигуры `Hide`. Естественно, классы `TSquare` и `TCircle` перекрывают эти методы (у класса `TFigure` эти методы вообще могут быть пустыми):

```
type TFigure = class
     // схематическое описание класса
       procedure Draw;
       procedure Hide;
     end;
     TSquare = class(TFigure)
       procedure Draw;
       procedure Hide;
     end;
     TCircle = class(TFigure)
       procedure Draw;
       procedure Hide;
     end;
procedure TFigure.Draw;
begin
end;
procedure TFigure.Hide;
begin
end;
procedure TSquare.Draw;
begin
  // тут как-то рисуем квадратик
end;
procedure TSquare.Hide;
begin
  // а здесь стираем его
end;
procedure TCircle.Draw;
begin
  // аналогично для кружка
end;
```

```
procedure TCircle.Hide;
begin
  // стираем кружок
end;
```

Рассмотрим следующую ситуацию. Пусть имеется некая подпрограмма (возможно метод определенного класса), в которой происходит рисование графического объекта. Этот объект передается подпрограмме в качестве параметра. Каким должен быть тип этого параметра? Следуя правилу для присваивания объектов, заключаем, что типом параметра должен быть `TFigure`, что позволит передать в подпрограмму объекты любых его дочерних классов. Заголовок подпрограммы может выглядеть следующим образом:

```
procedure WorkWithObjects(X: TFigure);
begin
  . . .
  X.Draw
end;
```

Использовать подпрограмму `WorkWithObjects` можно так:

```
var A: TCircle;
    B: TSquare;
. . .
A := TCircle.Create;
B := TSquare.Create;
WorkWithObjects(A);
WorkWithObjects(B);
```

Небольшое отступление: обратите внимание на использование конструкторов для создания объектов. Так как конструкторы в `TCircle` и `TSquare` не описывались, то в данном примере используется конструктор `TObject.Create`, доставшийся этим классам «по наследству». Однако этот конструктор создает именно объекты класса `TCircle` и `TSquare`. Если бы эти классы имели разный набор полей, для объектов резервировался бы разный объем динамической памяти. Напомним, что выделением памяти для объекта занимается некий дополнительный код, присутствующий в любом конструкторе неявно.

Вернемся к нашему примеру. Хотя он синтаксически корректен и компилируется, работать он будет неверно. Несмотря на тип фактических параметров процедуры `WorkWithObjects`, при работе эта подпрограмма будет вызывать метод `TFigure.Draw`.

Смоделируем вторую ситуацию. Пусть имеется массив из объектов класса `TFigure` или его наследников. Инициализируем такой массив и попытаемся нарисовать все графические объекты в цикле:

```
var Figures: array[1..3] of TFigure;
. . .
  // корректно по правилам присваивания для объектов
  Figures[1] := TCircle.Create;
  Figures[2] := TSquare.Create;
  Figures[3] := TCircle.Create;
```

```
  // пытаемся нарисовать все объекты в цикле
  for i := 1 to 3 do
    Figures[i].Draw;
```

Ожидается, что в результате работы цикла будут нарисованы круг, квадрат и круг. Тем не менее, будет получена лишь последовательность из трех вызовов `TFigure.Draw`.

Еще одна ситуация. Добавим в класс `TFigure` метод для перемещения фигур. Перемещение фигуры – это стирание фигуры, изменение ее координат и рисование ее на новом месте:

```
type TFigure = class
     . . .
       procedure Move;
     end;
procedure TFigure.Move;
begin
  Hide;
    // здесь изменили координаты фигуры
  Draw
end;
```

Логика работы метода `Move` сохраняется для дочерних классов `TCircle` и `TSquare`. Значит, переопределять этот метод в дочерних классах не требуется. Однако следующий вызов приведет не к перемещению квадрата, а к вызову `TFigure.Hide`, изменению координат (квадрата) и вызову `TFigure.Draw`:

```
var Square: TSquare;
. . .
  Square.Move; //квадрат не переместился!
```

Подобные расхождения в наших ситуациях межу желаемым поведением объектов и действительным, возникают из-за того, что нами использовались *статические методы*[1]. Адрес статического метода вычисляется на этапе компиляции. Он определяется классом объекта и не зависит от того, с каким классом будет связан объект на этапе выполнения. Рассмотрим строку вызова `X.Draw` из процедуры `WorkWithObjects`. Во время компиляции программы компилятор пытается определить тип переменной `X`. Он может сделать это по объявлению переменной в заголовке процедуры. Тип `X` это `TFigure`, значит запись `X.Draw` означает вызов метода `TFigure.Draw` (если вспомнить о неявном параметре `self`, то более точно – `TFigure.Draw(X)`).

Нам необходимо, чтобы поведение объекта, вызов его методов, определялись непосредственно в период выполнения программы. Это означает, что адрес метода должен вычисляться непосредственно в период выполнения по тому типу, который объект имеет в данный момент. Подобным образом работают *виртуальные методы*. Они функционируют по следующей схеме. Каждый объект наряду со значениями своих полей хранит указатель на специальную *таб-*

[1] Термин *статический метод* специфичен для Object Pascal, в котором статические методы противопоставляются виртуальным и динамическим. В других языках программирования под статическим методом обычно понимают методы, работающие с классом, а не объектом.

лицу виртуальных методов (virtual method table – VMT). Таблица виртуальных методов индивидуальна и единственна для каждого класса. В ней хранятся адреса всех виртуальных методов класса (как собственных, так и унаследованных). Связь между объектом и VMT класса осуществляется во время начальной инициализации объекта, то есть первого вызова конструктора. Виртуальные методы идентифицируются по константе-смещению в VMT. Во время выполнения программы из экземпляра объекта извлекается указатель на VMT и, используя константу-смещение, вычисляется адрес необходимого метода.

Для объявления виртуального метода используется директива `virtual`. Исходный метод, объявленный как виртуальный в классе-предке, перекрывается в классе-потомке методом с тем же именем и параметрами, что и исходный, и помечается директивой `override`. Если хотя бы одно из этих требований не соблюдено, связь между виртуальными методами в иерархии классов теряется.

Отредактируем классы `TFigure`, `TSquare` и `TCircle`, сделав их методы виртуальным:

```
type TFigure = class
       procedure Draw;virtual;
       procedure Hide;virtual;
     end;
     TSquare = class(TFigure)
       procedure Draw;override;
       procedure Hide;override;
     end;
     TCircle = class(TFigure)
       procedure Draw;override;
       procedure Hide;override;
     end;
```

При реализации методов директивы `virtual` и `override` не указываются.

Если внести подобные изменения в описания классов, то во всех перечисленных ситуациях работа будет происходить так, как нам требуется.

С учетом перечисленных примеров можно дать следующее определение полиморфизма. *Полиморфизм* – особый вид перекрытия методов при наследовании, при котором программный код, работавший с методами родительского класса, пригоден для работы с измененными методами дочернего класса.

Вопросы и упражнения.

1. Приведите примеры иерархии классов и ситуаций, которые требуют использования полиморфизма.

2. В какой секции видимости – `private`, `public` или `protected` – следует размещать виртуальные методы класса?

3. Класс `TAncestor` содержит 5 виртуальных методов, его наследник `TDescendant` вводит 2 новых виртуальных метода. Каков размер VMT класса `TDescendant`, если в программе объявлено 10 объектов этого класса (один элемент VMT занимает 4 байта).

9. Дополнительные аспекты использования полиморфизма.

Для реализации полиморфизма можно использовать не только виртуальные методы. Методы, объявленные с директивой `dynamic` (*динамические методы*) по функциональности аналогичны виртуальным. Иная внутренняя организация хранения их адресов позволяет сократить накладные расходы памяти при создании больших иерархий классов, но замедляет работу с такими методами. Для перекрытия динамических методов в дочерних классах также используется директива `override`:

```
type TFigure = class
       procedure Draw;virtual; //один виртуальный
       procedure Hide;dynamic; //второй - динамический
     end;
     TSquare = class(TFigure)
       procedure Draw;override;
       procedure Hide;override;
     end;
```

Вернемся к иерархии классов `TFigure`, `TCircle`, `TSquare`. В базовом классе `TFigure` реализация методов `Draw` и `Hide` абсолютно не важна, так как они все равно будут перекрываться в классах-наследниках. Мы оформили реализацию этих методов в виде пустых процедур. Более элегантное решение состоит в объявлении таких методов как *абстрактных*, не нуждающихся в реализации. Абстрактные методы объявляются при помощи директивы `abstract`, указанной после директивы `virtual` или `dynamic`:

```
type TFigure = class
       procedure Draw;virtual;abstract;
       procedure Hide;virtual;abstract;
     end;
//реализацию TFigure.Draw и TFigure.Hide писать не надо
```

Директива `abstract` применяется только для виртуальных и динамических методов в корневых классах иерархий. Попытка создать экземпляр класса, содержащего абстрактные методы, вызовет предупреждение, а обращение к абстрактному методу сгенерирует исключительную ситуацию.

Полиморфизм позволяет создавать в классах так называемые *виртуальные свойства*. Для этого методы чтения и записи свойства объявляются как виртуальные (использование динамических методов не допускается). Определив такие свойства в базовом классе, мы можем менять их поведение, перекрывая методы чтения и записи в дочерних классах. Виртуальные методы работы со свойствами обычно помещают в секцию `protected`.

Рассмотрим, как связан полиморфизм с вопросами создания и уничтожения объектов. Предположим, что имеется иерархия классов и планируется использовать набор объектов этих классов, разместив объекты в массиве или списке. Создание объектов в наборе операция достаточно специфическая, каждый объект создается вызовом персонального конструктора. А вот уничтожать созданные объекты можно было бы и «скопом», в одном цикле. Для этого деструктор в иерархии классов должен быть полиморфным. Создатели Object Pascal

посчитали данную ситуацию стандартной и объявили деструктор `TObject.Destroy` как виртуальный. Чтобы не обрывать цепочку виртуальных методов, рекомендуется деструкторы в пользовательских классах объявлять без параметров, с директивой `override`.

Одним из ключевых понятий системы Windows является *сообщение*. Сообщение – это сигнал приложению от операционной системы о том, что произошло некоторое событие (нажатие клавиши, перемещение мыши, запуск приложения). Каждый вид сообщений идентифицируется уникальной числовой константой. Object Pascal позволяет создавать методы для обработки сообщений. Схема внутренней реализации таких методов напоминает динамические методы. Метод обработки сообщений это всегда процедура с единственным `var`-параметром. За заголовком метода следует директива `message` и номер сообщения Windows, которое этот метод будет обрабатывать. Этот номер обычно представляется мнемонической константой:

```
type TSomeClass = class
     . . .
       procedure WmUser(var M: TMessage);message wm_User;
     end;
```

Модуль `Messages` содержит константы для сообщений, а также большое количество типов, идентифицирующих конкретные сообщения.

Вопросы и упражнения.

1. Приведите примеры иерархии классов, в которой можно было бы определить виртуальные свойства.

10. RTTI и размещение класса и объектов в памяти.

Рассмотрим одну полезную особенность объектной модели языка Object Pascal. Для любого класса в Object Pascal после компиляции программы сохраняется некая дополнительная информация, которая размещается в памяти непосредственно перед VMT. Эта информация называется *информацией о типе периода времени выполнения* (runtime type information, RTTI). Как было сказано выше, любой объект кроме данных полей содержит указатель на VMT (возможно пустую, если у класса и его предков нет виртуальных методов). Следовательно, во время работы программы любой объект может получить доступ к RTTI своего класса. Схема размещения объектов и класса в памяти представлена на рис. 1.

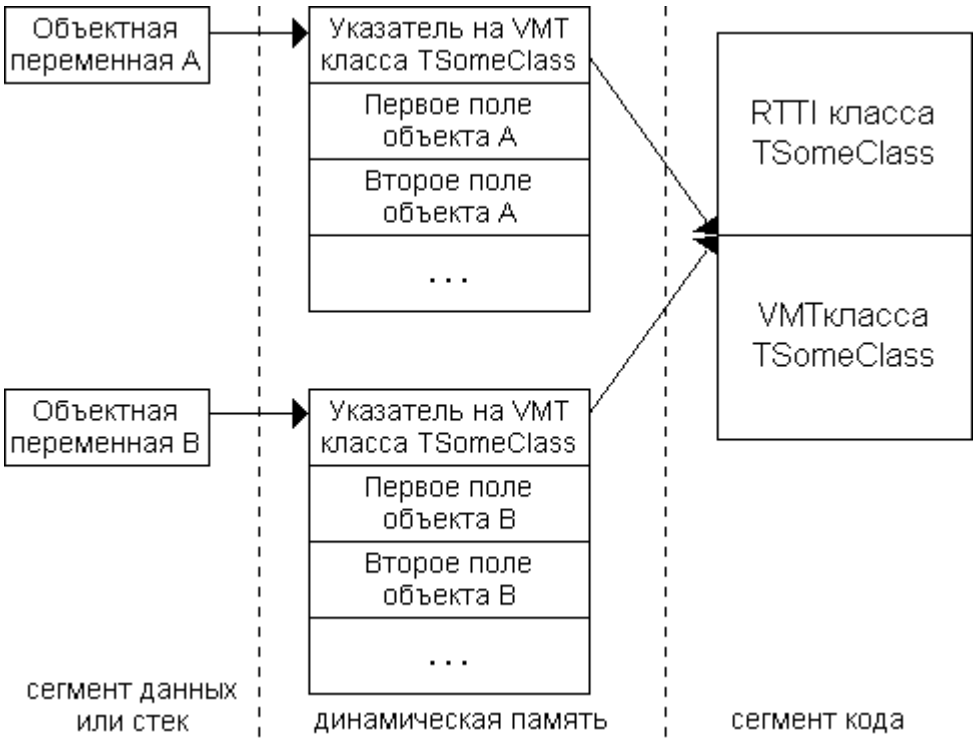

Рис. 1. Схема размещения объектов и класса в памяти

Данные, хранящиеся в RTTI, не документированы. Однако известно, что RTTI в числе прочих хранятся следующие параметры:

1. указатель на VMT класса-предка;
2. указатель на строку с именем класса;
3. указатель на таблицу динамических методов класса;
4. размер экземпляра объекта в байтах.

Эти данные позволяют во время выполнения программы *контролировать* (type checking) и *приводить* (type casting) объектные типы.

Для контроля типов используется оператор `is`. Выражение `<объект> is <класс>` возвращает `true`, если `<объект>` принадлежит классу `<класс>` или потомкам этого класса:

```
if Man is TPerson then . . .
```

Для приведения типов используется оператор `as` в следующей форме:

```
(Man as TPerson).SetAge(10);
```

Допустима традиционная конструкция приведения типов в виде `TPerson(Man).SetAge(10)`, однако оператор `as` является более безопасным. В случае неудачи (то есть когда объект не относится к рассматриваемому классу или его потомкам) он генерирует обрабатываемую исключительную ситуацию, а жесткое приведение типов может привести к краху приложения.

11. Метаклассы.

Delphi предоставляет возможность для расширения своей функциональности путем написания пользовательских компонент. По существу, написание компонент – это написание новых классов. Для системы программирования

важно иметь возможность манипулировать классами, созданными пользователем.

Для работы с классами в Object Pascal существует специальный тип данных – *метакласс*. Определение в программе метакласса и объявление соответствующей переменной выглядит следующим образом:

```
type TPersonClass = class of TPerson;
var PersonRef: TPersonClass;
```

Для определения метакласса используется конструкция `class of <имя класса>`. Что можно присваивать переменной `PersonRef`? Ей присваивается либо класс, объявленный после `class of`, либо его дочерние классы:

```
PersonRef := TPerson;
. . .
PersonRef := TEmployee; // TEmployee = class(TPerson)
```

Значением переменной типа метакласс фактически является указатель на VMT соответствующего класса. В приведенном примере `PersonRef` содержит указатель на VMT класса `TPerson`, затем указатель на VMT класса `TEmployee`.

В Object Pascal имеется предопределенный тип `TClass`, объявленный следующим образом:

```
type TClass = class of TObject;
```

Так как `TObject` является предком для всех классов, то переменной типа `TClass` можно присваивать любой класс.

Повторим, что ценность метаклассов состоит в возможности создавать программы, которые манипулируют классами, не существующими в момент написания программы. Примером этого служит работа с формой. Когда мы конструируем форму, мы создаем новый класс, порожденный от `TForm`. Класс `TForm` содержит виртуальный конструктор. Метод `TApplication.CreateForm` ответственен за создание и визуализацию формы. Его заголовок имеет вид `TApplication.CreateForm(T: class of TForm; var F: TForm)`. В теле метода создается объект `F` класса `T`. Благодаря ссылке на класс и виртуальному конструктору метод `CreateForm` может создать объект не только класса `TForm`, но и любого дочернего класса.

Вопросы и упражнения.

1. Можно ли, используя переменную типа метакласс, получить доступ к полям объекта?

2. Используя иерархию классов `TFigure`, `TSquare` и `TCircle`, напишите функцию, которая создавала бы объект класса, переданного ей в качестве параметра. Какие изменения в классы требуется внести для этого?

12. Классовые методы и общая информация класса.

В предыдущих примерах для вызова метода класса использовался экземпляр класса. Object Pascal позволяет описать в классе такие методы, для работы с которыми нет необходимости создавать объект. Такие методы называются

классовыми методами (class method)[1]. Для их объявления достаточно указать слово `class` непосредственно перед ключевыми словами `procedure` или `function` при описании класса и при реализации метода:

```
type TSomeClass = class
       class procedure Hello;
     end;
class procedure TSomeClass.Hello;
begin
  writeln('Hello. This is class method')
end;
```

Так как в момент вызова классового метода экземпляр класса может и не существовать, то в теле такого метода запрещено обращение к полям объекта и обычным методам. Однако классовый метод может вызывать другие классовые методы и конструктор класса.

Для вызова классового метода возможно применение как синтаксиса `<имя класса>.<имя классового метода>`, так и `<имя объекта>.<имя классового метода>`. Для вызова классового метода можно также использовать метакласс:

```
type TSomeClassRef = class of TSomeClass;
var C: TSomeClass;
    R: TSomeClassRef;
. . .
TSomeClass.Hello; //все вызовы корректны
C := TSomeClass.Create; //дают одинаковый результат
C.Hello;
R := TSomeClass;
R.Hello;
```

Указатель `self`, который передается в классовый метод, содержит адрес VMT класса. Разрешено объявление виртуальных классовых методов. Для классовых методов действуют обычные правила видимости атрибутов класса.

Классовые методы обычно отражают действия, которые специфичны для класса в целом, а не для конкретных объектов. В этом смысле можно сказать, что код таких методов *разделяется* между всеми объектами класса. К сожалению, Object Pascal не предусматривает стандартных средств для создания полей, данные в которых также разделялись бы между всеми объектами класса. Однако разделяемые поля можно моделировать с использованием модулей. В качестве примера попытаемся создать в классе `TEmployee` разделяемое поле, содержащее надбавку к зарплате служащего. Во-первых, поместим класс в отдельный модуль. Во-вторых, в секцию модуля `implementation` поместим переменную для разделяемого поля. И, наконец, определим в классе `TEmployee` методы для чтения и записи этой переменной:

```
unit TEmployeeClass;
interface
type TEmployee = class
```

[1] В литературе по ООП для таких методов часто используется термин «разделяемые» (shared methods) или «статические» (static methods).

```
      procedure SetIncrease(Value: Integer);
      function GetIncrease: Integer;
    end;
implementation
var Increase: Integer = 0;
procedure TEmployee.SetIncrease(Value: Integer);
begin
  Increase := Value
end;
function TEmployee.GetIncrease: Integer;
begin
  Result := Increase
end;
end.
```

Использоваться общее поле может следующим образом:

```
var A, B, C: TEmployee;
. . .
writeln(A.GetIncrease); //вначале надбавка равна 0
writeln(B.GetIncrease); //у любого объекта
C.SetIncrease(1000);    //поменяли у одного
writeln(A.GetIncrease); //=1000, поменялась у всех объектов
```

Логично предоставить доступ к общему полю не только через конкретный объект, но и через класс. Для этого объявим методы работы с полем как классовые:

```
type TEmployee = class
      class procedure SetIncrease(Value: Integer);
      class function GetIncrease: Integer;
    end;
. . .
var A, B, C: TEmployee;
. . .
A.SetIncrease(1000); //можно работать через объект
writeln(TEmployee.GetIncrease); //а можно через класс
```

Вопросы и упражнения.

1. Опишите класс, который содержит метод, позволяющий узнать количество созданных объектов этого класса в приложении.
2. Приведите пример класса с виртуальными классовыми методами.
3. Можно ли использовать классовые методы для обслуживания свойств?

13. Обработчики событий. Указатели на методы.

Рассмотрим класс `TPerson` и метод `SetAge` установки значения возраста:

```
type TPerson = class
    . . .
      procedure SetAge(Age: Integer);
    end;
procedure TPerson.SetAge(Age: Integer);
```

```
begin
  if Age > 0 then fAge := Age
end;
```

Представим себе, что мы хотим предоставить пользователю контроль над ситуацией, когда значение параметра Age не подходит для установки поля. Употребляя терминологию ООП, можно сказать: мы хотим предоставить пользователю контроль над определенным *событием* (неправильное значение параметра), происходящим при работе с объектом. Мы хотим разрешить пользователю иметь *обработчик* данного события. Одно из решений, которое можно предложить в этой ситуации, заключается в следующем: поместим в класс TPerson виртуальный метод, вызываемый в результате события:

```
type TPerson = class
     . . .
       procedure SetAge(Age: Integer);
       procedure DoSomething;virtual;
     . . .
     end;
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
  else DoSomething;
end;
procedure TPerson.DoSomething;
begin
end;
```

Теперь для того, чтобы обработать событие, пользователю достаточно породить дочерний класс от TPerson и перекрыть метод DoSomething:

```
type TMyPerson = class(TPerson)
       procedure DoSomething;override;
     end;
procedure TMyPerson.DoSomething;
begin
  writeln('Something is wrong!')
end;
```

Такой подход для обработки событий имеет недостатки. Пользователи вынуждены «плодить» многочисленные производные классы, усложняя логику программы. Если бы в предыдущем примере планировалось создать десять объектов класса TPerson с разной обработкой события, мы вынуждены были бы породить от TPerson десять дочерних классов.

Устранить данный недостаток можно, если использовать для обработки событий процедурные типы. В этом случае в классе объявляется свойство соответствующего типа, после создания объекта свойству присваивается написанная пользователем подпрограмма-обработчик.

```
type TProc = procedure;
type TPerson = class
     . . .
       fEvent: TProc;
```

```
        procedure SetAge(Age: Integer);
        property Event: TProc read fEvent write fEvent;
      end;
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
  else fEvent
  // желательно дополнительно проверять fEvent на nil
end;
. . .
var Man, Woman: TPerson;

procedure ManEvent;
begin
  writeln('Something is wrong with my age')
end;

procedure WomanEvent;
begin
  writeln('I am 18 y.e.!')
end;
. . .
Man := TPerson.Create;      //создание объектов
Woman := TPerson.Create;
Man.Event := ManEvent;      //назначение обработчиков событий
Woman.Event := WomanEvent;
Man.SetAge(-10);           //здесь эти обработчики сработают
Woman.SetAge(-10);
```

Подобный подход лучше, чем порождение многочисленных дочерних классов, однако он является «не совсем объектно-ориентированным». По принципам «чистого» ООП программа это набор объектов взаимодействующих классов. В нашем случае процедуры обработки события не являются методами класса, а «висят в воздухе». Попробуем объединить их в специальный класс, чтобы соблюсти принципы объектно-ориентированной разработки:

```
type TEventClass = class
        procedure ManEvent;
        procedure WomanEvent;
      end;
procedure TEventClass.ManEvent;
begin
  writeln('Something is wrong with my age')
end;
procedure TEventClass.WomanEvent;
begin
  writeln('I am 18 y.e.!')
end;
```

Однако в таком варианте попытка назначить объектам обработчики вызывает синтаксическую ошибку:

```
var Man, Woman: TPerson;
```

```
    EventObject: TEventClass;
. . .
EventObject := TEventClass.Create;
Man.Event := EventObject.ManEvent;   //синтаксическая ошибка!
```

Хотя метод `TEventClass.ManEvent` имеет сигнатуру типа `TProc`, он (как любой метод) получает неявный параметр `self`. Для манипуляций с методами необходимо использовать специальный процедурный тип, который называется *указателем на метод* (method pointer). Этот тип объявляется следующим образом:

```
type TProc = procedure of object;
```

Конструкция `of object` после сигнатуры подпрограммы говорит об объявлении указателя на метод. Если изменить объявление типа поля и свойства в классе `TPerson` на тип указателя на метод, то для обработки событий можно будет использовать методы другого класса (при условии совпадения сигнатур):

```
type TProc = procedure of object;
type TPerson = class
     . . .
       fEvent: TProc;
       procedure SetAge(Age: Integer);
       property Event: TProc read fEvent write fEvent;
     end;
```

Переменная типа указатель на метод занимает 8 байт и содержит адрес метода и ссылку на объект (значение `self` для конкретного объекта).

Подход, при котором методы обработки событий объектов сосредоточены в одном классе, называется *делегированием*.

Обработка событий и делегирование эффективно используется в IDE Delphi. Любой компонент, помещаемый на форму при проектировании, представляет собой объект некоего класса. Компоненты обладают многочисленными событиями. Тип простейших событий с единственным параметром `Sender` описан следующим образом:

```
type TNotifiEvent = procedure(Sender: TObject) of object;
```

В качестве примера рассмотрим класс `TEdit`. Он обладает событием `OnClick`. Для реализации работы с ним класс содержит поле `FOnClick` и свойство `OnClick`.

```
type TEdit = class(TCustomEdit)
     . . .
       FOnClick: TNotifiEvent;
       property OnClick: TNotifiEvent read FOnClick
                                        write FOnClick;
     end;
```

Если в процессе разработки мы помещаем на форму компонент `Edit1: TEdit` и пишем для него обработчик события `OnClick`, то фактически мы пишем новый метод класса формы `TForm1.Edit1Click`. При создании формы в

начале выполнения приложения метод формы связывается со свойством компонента (без нашего участия):

```
Edit1.OnClick := Form1.Edit1Click; //Form1 – объект TForm1
```

При щелчке на компоненте Edit1 системой вызывается метод TEdit.Click. В нем производится проверка наличия обработчика события OnClick и вызов его, если он существует.

```
if Assigned(OnClick) then OnClick(self);
```

Функция Assigned возвращает значение «истина», если ее аргумент не равен nil. Параметр Sender передаваемый в обработчик события, позволяет различать объекты, которые вызвали событие. Это помогает использовать один обработчик для разных объектов. Предположим, на форме имеется компоненты Edit1 и Button1 классов TEdit и TButton соответственно. Необходимо при щелчке на компоненте Edit1 установить его свойство Text равным 'Hi', при щелчке на Button1 – свойство Caption равным 'Hello':

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
  if Sender is TEdit then TEdit(Sender).Text := 'Hi'
  else if Sender is TButton
       then TButton(Sender).Caption := 'Hello'
end;
```

Приведенный выше обработчик события Edit1Click решает эту задачу. Этот обработчик нужно назначить событию Button1.OnClick, выбрав его в выпадающем списке на закладке Events в Инспекторе Объектов. В примере использовался приведение и контроль типов объектов.

Вопросы и упражнения.

1. Почему метод TPerson.DoSomething был объявлен пустым, а не абстрактным?

2. Почему переменная типа указатель на метод должна хранить значение self?

3. На форме размещены два компонента Edit1 и Edit2 и кнопка. Компонент Edit1 имеет обработчик события OnClick. Напишите код, позволяющий во время работы приложения при нажатии на кнопку назначить компоненту Edit2 обработчик события OnClick первого компонента.

14. Публикуемые члены класса.

Ранее были рассмотрены следующие директивы ограничения видимости членов класса – private, protected и public. Директивы перечислены в порядке увеличения открытости секций видимости. В язык Object Pascal введены специальные средства, которые позволяют осуществлять доступ к членам класса через IDE. Благодаря этому можно работать со свойствами, проанализировать метод класса и назначить его обработчиком события, используя Инспектор объектов.

Директива ограничения видимости published позволяет осуществлять доступ к членам класса в процессе его разработки. Члены класса из секции published – *публикуемые* – не имеют ограничений на использование. Компилятор генерирует для них специальную информацию, позволяющую работать с ними в IDE и Инспекторе объектов.

На состав секции published наложены определенные ограничения. Будем говорить, что класс поддерживает RTTI, если он скомпилирован с директивой $M+, либо являться потомком класса, скомпилированного с этой директивой (директива помещается перед описанием класса)[1]. Если в классе планируется секция published, то это должен быть класс с поддержкой RTTI. В таких классах секция published – это секция по умолчанию. Если в секцию published помещается поле, то его тип – это только класс, причем с поддержкой RTTI. Публикуемые свойства не могут быть свойствами-массивами. Тип публикуемого свойства должен быть порядковым, множеством в пределах Integer, классом с поддержкой RTTI, Variant, вещественным типом, любым строковым типом. Для таких свойств обязательно наличие директив read и write.

Для публикуемых свойств возможно указание спецификаторов default и stored. Эти спецификаторы управляют процессом сохранения публикуемых свойств в DFM файле.

Директива default служит для указания для свойства значения по умолчанию. Это значение должен устанавливать конструктор класса. Свойство сохраниться в DFM файле, если его значение отлично от принятого по умолчанию. Не путайте директиву default для публикуемых свойств с директивой для основного свойства класса. Пример свойства с директивой default приведен ниже:

```
type TC = class
     . . .
     published
       property P: Integer read fP write fP default 5;
     end;
. . .
constructor TC.Create;
begin
  P := 5;
end;
```

Директива stored используется для указания того, будет ли свойство запоминаться в DFM файле. В объявлении свойства за директивой stored должно следовать либо имя поля булевого типа данного класса, либо логическая константа, либо имя метода-функции класса, не имеющего параметров и возвращающего булево значение. Свойство запоминается в DFM файле, если значение, следующее за stored, равно true.

```
type TC = class
```

[1] Термин «класс с поддержкой RTTI» не должен вводить в заблуждение. RTTI есть в любом классе, в *классе с поддержкой RTTI* в таблицу RTTI помещается некая дополнительная информация.

```
. . .
function NeedStore: Boolean;
published
 property P: Integer read fP write fP
                                stored NeedStore default 5;
end;
```

Заметим, что нормальной ситуацией для `published`-свойств является такая, при которой они сохраняются в `DFM` файле (без указания `stored` свойства сохраняются).

Для членов класса существует возможность изменения ограничения видимости в классе-потомке с `protected` на `public` или `published`. Обычно это используют при создании новых компонент. При переносе свойств достаточно указать ключевое слово `property` и имя свойства в новой секции, не указывая тип свойства и методы или поля для чтения и записи. При необходимости можно указать новое значение свойства по умолчанию:

```
type TArrow = class(TGraphicControl)
     . . .
     published
       property Height default 20;
     end;
```

Рис. 2 демонстрирует возможное изменение ограничения видимости:

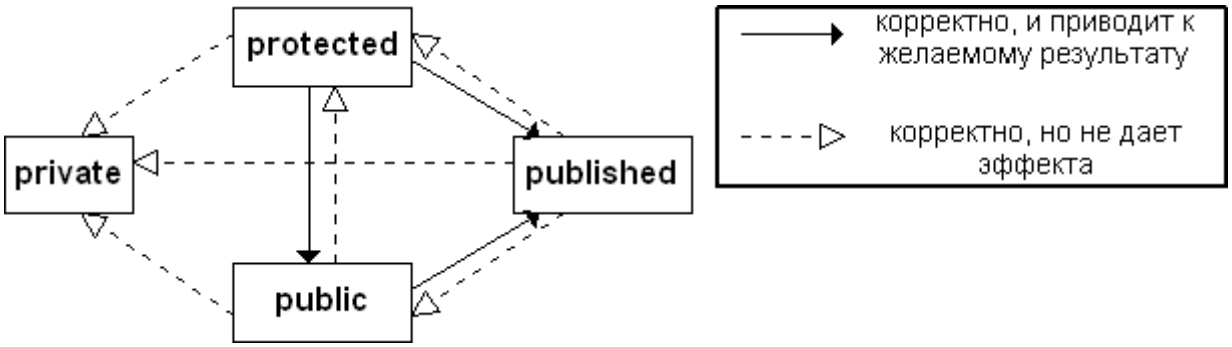

Рис. 2. Возможные изменения ограничения видимости.

Вопросы и упражнения.

1. Какое значение свойства отображается Инспектором объектов по умолчанию – устанавливаемое конструктором или указанное после `default`?

2. Проверьте, можно ли в классе-потомке уменьшить видимость членов по сравнению с классом-предком.

3. *Сериализацией* (serialization) называется процесс преобразования данных объекта в линейную последовательность байтов для сохранения. Можно утверждать, что секция `published` призвана обеспечить сериализацию объектов класса. Реализуйте класс с собственными методами для сериализации и *десериализации* (восстановления) объектов.

15. Исключительные ситуации

Термин *исключительная ситуация* (exception) обозначает любую ошибку или ошибочное условие, возникающие в процессе выполнения программы. Примерами исключительных ситуаций являются деление на 0, запись в файл, открытый только для чтения, ошибки при преобразовании типов. Традиционный подход для контроля и обработки подобных ошибок заключается в использовании условных операторов. Однако они загромождают программу, делают ее менее ясной.

Object Pascal предлагает следующий подход: при исключительной ситуации, возникшей в подпрограмме, информация об этой исключительной ситуации при помощи объекта передается специальному внешнему обработчику, написанному программистом.

Для передачи объекта исключительной ситуации из подпрограммы применяется предложение `raise <объект исключительной ситуации>`. Вот как выглядит создание и передача объекта исключительной ситуации:

```
raise EOutOfMemory.Create('Мало памяти');
```

Каждому типу исключительной ситуации обычно соответствует некий класс. В составе объектной библиотеки VCL входит иерархия классов исключительных ситуаций, корнем которой является класс `Exception`. Приведем некоторые классы и соответствующие им исключительные ситуации:

`EAbort` – «безмолвная» исключительная ситуация, без сообщений; генерируется вызовом процедуры `Abort`;

`EDivByZero` – деление целого числа на 0;

`EInOutError` – ошибка при доступе к файлу;

`EOutOfMemory` – недостаток памяти;

`EMathError` – общая ошибка при работе с вещественными числами;

`EZeroDivide` – деление вещественного числа на 0;

`EConvertError` – ошибка при преобразовании типов.

Полный список классов содержит справочная система Delphi. Основные классы исключительных ситуаций описаны в модуле `SysUtils`.

Подчеркнем важный нюанс: при помощи оператора `raise` можно передать из подпрограммы любой объект. Программист может выбирать, использовать ли ему готовые классы исключительных ситуаций или писать собственные.

Приведем пример функции, генерирующей исключительную ситуацию:

```
function StrToPersent(const S: string): Integer;
begin
  Result := StrToInt(S);
  if (Result < 0) or (Result > 100) then
    raise EConvertError.Create(S + 'is not a valid value')
end;
```

А теперь тот же пример, но используется собственный класс:

```
type TMyClass = class
       MessageToSend: string. . .
     end;
```

```
function StrToPersent(const S: string): Integer;
var Obj: TMyClass;
begin
  Result := StrToInt(S);
  if (Result < 0) or (Result > 100) then
    begin
      Obj := TMyClass.Create;
      Obj.MessageToSend := S + 'is not a valid value';
      raise Obj
    end
end;
```

Обработчики исключительных ситуаций реализуются при помощи *защищенного блока* `try-except`. Синтаксис объявления блока следующий:

```
try
  //операторы, которые могут вызвать исключительную ситуацию
except
  //операторы обработки исключительных ситуаций
end;
```

При нормальном ходе программы выполняются операторы, размещенные между `try` и `except`, а затем – размещенные после `end`. Если какой–либо из операторов между `try` и `except` вызвал исключительную ситуацию, управление сразу передается в часть `except-end`. Предполагается, что в ней сосредоточена обработка исключительной ситуации. После обработки исключительной ситуации выполняются операторы, размещенные за `end`. Конструкции `try-except` могут быть вложенными.

Для распознавания исключительной ситуации в части `except-end` служит последовательность блоков обработки вида `on <класс исключительной ситуаций> do <оператор>`. К примеру:

```
try
  //работа с вещественными числами
except
  on EZeroDivide do . . . ; //обрабатываем ошибку деления на 0
  on EMathError do . . . ;  //обработка других матем. ошибок
end;
```

Если для класса исключительной ситуации существует блок обработки, выполняется часть блока после `do` и управление передается за блок `try-except`. Порядок блоков обработки имеет значение: вначале проводится распознавание частных исключительных ситуаций, затем – более общих (как и в приведенном примере).

Если необходимо, чтобы нераспознанные исключительные ситуации были как-то обработаны, можно поместить операторы их обработки в секцию `except-end` после слова `else`:

```
except
  on EZeroDivide do . . . ; //обрабатываем ошибку деления на 0
  on EMathError do . . . ;  //обработка других матем. ошибок
  else . . .    //обработка нематематических ошибок
```

```
end;
```

Наша функция StrToPersent может генерировать исключительную ситуацию класса EConvertError. В следующей функции она распознается и обрабатывается:

```
function IncPercent(const S: string): string;
begin
  try
    Result := IntToStr(StrToPercent(S) + 1);
  except
    on EConvertError do Result := '0';
  end
end;
```

Если мы хотим возложить часть обработки исключительной ситуации на некий внешний блок, то исключительную ситуацию можно сгенерировать повторно, используя raise:

```
on EZeroDivide do
begin
  . . .   // частично обработали мы
  raise;  // пусть обрабатывает внешний обработчик
end;
```

Если необходимо получить доступ к объекту, описывающему исключительную ситуацию, то используется следующая форма блока обработки: on <идентификатор объекта>: <класс исключительной ситуации> do <оператор>;

```
on E: EConvertError do
begin
  Result := '0';
  ShowMessage(E.Message)
end;
```

Освобождение созданного объекта исключительной ситуации происходит автоматически. Переменную, используемую в блоке on (в примере это E), нигде описывать не надо.

Для корректного освобождения ресурсов приложения часто используется блок try-finally. Синтаксис блока следующий:

```
try
  // операторы, которые могут вызвать исключительную ситуацию
finally
  // эти операторы выполняются всегда
end;
```

При возникновении исключительной ситуации в части try-finally управление передается на часть finally-end. При нормальной работе выполняются все операторы между try и end.

Если возникшая исключительная ситуация программистом не обработана, вызывается обработчик события OnExcept объекта Application. Если этот об-

работчик не установлен, выводится стандартное окно с текстовым сообщением об ошибке.

Вопросы и упражнения.

1. В классе TArray из параграфа 4 метод ReadElement возвращал значение 0 при несоответствии параметра Ind диапазону индексов массива:

```
function TArray.ReadElement(Ind: Integer): double;
begin
  if (Ind > 0) and (Ind <= fLength) then Result := fData[Ind]
  else Result := 0
end;
```

Модифицируйте данный метод так, чтобы в этом случае генерировалась исключительная ситуация. Для типа исключительной ситуации опишите собственный класс, который позволяет передавать значение неправильного индекса в качестве параметра. Продемонстрируйте обработку данной исключительной ситуации.

16. Иерархия классов библиотеки VCL.

Система Delphi поставляется с обширной библиотекой классов, призванной упростить создание пользовательских приложений. Эта библиотека носит название Visual Component Library. В библиотеке классов VCL реализована так называемая *модель PME* (property, method, event). Любой класс из VCL можно полностью охарактеризовать, указав его свойства, доступные методы и события, на которые он может реагировать. Благодаря библиотеке VCL пользователь практически никогда не разрабатывает приложение абсолютно «с нуля». Даже в самом простом приложении для Windows задействовано десятки классов из VCL.

Объектная библиотека VCL насчитывает около 200 классов, составляющих сложное дерево иерархии. Основными классами, «стволом дерева», являются следующие:

```
TObject-TPersistent-TComponent-TControl-(TGraphicControl,TWinControl)
```

Класс TObject является общим предком всех классов. Он содержит конструктор Create, виртуальный деструктор Destroy, метод Free для освобождения объекта, методы для работы с сообщениями системы Dispatch и DefaultHandler и большое число классовых методов для работы с информацией о классе. Эти методы позволяют узнать предок класса (ClassParent), имя класса (ClassName), адрес метода, размещенного в секции published, по имени метода (MethodAddress), размер объекта данного класса (InstanceSize) и многое другое. Виртуальные методы TObject хранятся по отрицательным смещениям таблицы VMT, так что «первым» методом в VMT будет первый виртуальный метод производного класса.

Особенностью класса TPersistent является то, что он содержит методы для сохранения и чтения данных своих published-членов. TPersistent содержит также методы для копирования данных из полей одного объекта в поля

другого. Так как переменные-объекты являются всего лишь указателями на данные в памяти, то код

```
var A, B: TSomeClass;
. . .
A := B;
```

приведет к тому, что и A, и B будут ссылаться на один участок памяти, а данные объекта A будут потеряны. Чтобы этого не произошло, следует использовать методы Assign, который позволяет объекту присваивать данные, связанные с другим объектом, и AssignTo, копирующий данные объекта в другой объект. На уровне TPersistent данные методы объявлены как абстрактные, классы-потомки должны переопределить их. Класс TPersistent скомпилирован с директивой $M+, а значит, у него и всех его потомков члены класса по умолчанию имеют видимость published.

Класс TComponent происходит от TPersistent и является предком визуальных и невизуальных компонент. Объекты класса TComponent и его потомков называют *компонентами*. На уровне TComponent реализована поддержка возможности редактирования свойств компонент в Инспекторе объектов. Класс TComponent вводит новый вид отношения между объектами – *отношение владения*. Если один объект владеет другими, то при уничтожении объекта-владельца автоматически уничтожается все его «имущество». Некоторые из свойств класса TComponent перечислены ниже.

Owner – компонент, который владеет данным,

ComponentState – текущее состояние компонента (редактируется, загружается и т.п.),

ComponentCount – количество компонентов, принадлежащих данному,

Components – массив принадлежащих компонентов,

ComponentIndex – текущая позиция в этом массиве,

Name – имя компонента, используется как идентификатор поля в класс формы,

Tag – число типа Integer.

Класс TComponent определяет виртуальный конструктор Create с одним параметром – объектом-владельцем для создаваемого компонента (допустимым является значение nil).

Методы TComponent позволяют найти компонент по имени в массиве Components (FindControl), получить список всех компонент, владельцем которых является данный и тому подобное. Большинство классов невизуальных компонент порождены непосредственно от TComponent.

От класса TComponent происходит класс TControl – общий предок всех визуальных компонент. Класс TControl содержит свойства позиционирования компонента (Top, Left, Widht, Height, Align), свойства клиентской области (ClientRect, ClientWidth, ClientHeight), свойства внешнего вида (Visible, Enabled, Font, Color), строковые свойства (Caption, Text, Hint), свойства мыши (Cursor, DragCursor, DragMode). На уровне класса TControl появляются обработчики событий: мыши (OnClick, OnDblClick, OnMouseDown, OnMouseMove, OnMouseUp) и событий перетаскивания Drag&Drop (OnDragOver,

OnDragDrop, OnEndDrag). Каждый компонент класса TControl содержит в свойстве Parent компонент класса TWinControl, отвечающий за отображение компонента. Если свойство Parent не установлено, компонент не показывается.

Класс TWinControl является предком для компонент, инкапсулирующих стандартные элементы управления, которые могут получать фокус (поля ввода, кнопки, списки и другие). Компоненты класса TWinControl могут быть родительскими (Parent) для других визуальных компонент. Основные свойства класса предназначены для управления внешним видом компонента и фокусом ввода.

Handle – дескриптор, целое число для идентификации объекта системой Windows,

Ctrl3D – определяет, выглядит ли компонент объемным,

HelpContext – номер раздела справочной системы для оперативной подсказки

Controls – список элементов управления, для которых компонент является родительским

TabStop, TabOrder – управляют перемещениями по компонентам на форме при нажатии клавиши Tab.

Класс TWinControl содержит методы управлением фокусом (Focused, CanFocus, SetFocus), методы позиционирования и выравнивания, методы создания и уничтожения окна (CreateParams, CreateWnd, DestroyWnd). Метод ControlAtPos возвращает дочерний компонент для указанной точки, метод FindNextControl позволяет найти компонент, получающий фокус после данного.

Компоненты класса TWinControl поддерживают обработку событий клавиатуры (OnKeyDown, OnKeyUp, OnKeyPress) и событий управления фокусом (OnEnter, OnExit).

Класс TGraphicControl используется для визуальных компонент, которые отображаются (рисуются) системой Delphi, а не Windows. Это позволяет экономно расходовать системные ресурсы. Потомки класса TGraphicControl (например, TLabel) не могут получать фокус. Класс TGraphicControl содержит свойство Canvas – «холст» для рисования компонента и виртуальный метод Paint, содержащий операторы для рисования компонента и переопределяемый потомками класса.

Вопросы и упражнения.

1. Как создать компонент во время выполнения программы? Напишите код, который делает это.

2. Найдите в документации по Delphi описание класса TCustomControl. Для чего предназначен этот класс, какие его основные свойства и методы?

17. Создание и использование DLL в Delphi.

Библиотека динамической компоновки (dynamic link library, DLL) – выполняемый модуль Windows, код и ресурсы которого могут использоваться приложениями и другими DLL. Файлы библиотек динамической компоновки обычно

имеют расширение DLL. Применение DLL оправдано, если приложение разрабатывается на нескольких языках программирования, программа содержит национальные ресурсы (надписи и тому подобное), существует несколько приложений, использующих сходный набор функций.

Структура исходного кода DLL схожа со структурой обычной программы. Код начинается с ключевого слова library, за которым следует имя библиотеки:

```
library MyDLL;
```

Все подпрограммы, которые будут вызываться из других приложений, объявляются в DLL с директивой stdcall. Вызов подпрограмм, объявленных без директивы stdcall, может быть затруднен в приложениях, написанных не в Delphi.

```
procedure MyProc(A: Integer; B: Char);stdcall;
```

Подпрограммы, экспортируемые из DLL, должны быть перечислены в специальном разделе exports исходного кода.

```
exports MyProc, MyFunct;
```

Таких разделов может быть несколько и в любом месте исходного кода DLL, главное, чтобы экспортируемые подпрограммы были описаны перед упоминанием в разделе exports.

Каждая экспортируемая из DLL подпрограмма идентифицируется для распознавания двумя ключами: *числовым индексом* и *именем экспорта*. По умолчанию имя экспорта совпадает с именем подпрограммы (большие и малые буквы в имени экспорта различаются), а индекс – с порядковым номером подпрограммы в разделе exports. Однако такой порядок можно обойти, явно указав имя экспорта и (или) числовой индекс.

```
exports
  MyProc1,
  MyProc2 name 'MyDinProc',
  MyProc3 index 3, //целое число от 1 до 2,147,483,647
  MyProc4 index 4 name 'MyPr@1';
 //в имени могут быть спецсимволы
```

Будьте внимательны при явном указании индекса. Его уникальность компилятором не отслеживается.

Рассмотрим пример DLL. Для получения некой «заготовки» DLL в IDE Delphi можно использовать следующую последовательность команд: File|New...|DLL. Далее наполняем эту «заготовку» содержимым:

```
library MathLib;
uses SysUtils; //модуль Classes нами не используется
function Max(X,Y: Integer): Integer;stdcall;
begin
  if X < Y then Result := Y else Result := X
end;
function Min(X,Y: Integer): Integer;stdcall;
begin
```

```
  if X < Y then Result := X else Result := Y
end;
exports Min, Max;
begin
end.
```

После компиляции этого кода (Ctrl+F9) в рабочем каталоге получим файл Mathlib.dll.

Между begin и end в коде DLL можно поместить операторы инициализации. Они выполняющиеся при загрузке библиотеки DLL в память.

Различают следующие виды импорта подпрограмм DLL – *статический* и *динамический*.

При статическом импорте DLL загружается в память в момент старта приложения и находится там до окончания его работы. При этом в исходном коде приложения размещены явные ссылки на подпрограммы DLL. Следующие примеры демонстрируют три возможных вида этих ссылок.

1. По имени подпрограммы в исходном коде DLL:

```
procedure MyProc1;stdcall;external 'MYLIB.DLL'
```

2. По имени экспорта:

```
procedure MyProc2; stdcall; external 'MYLIB.DLL'
                            name 'MyDinProc2'
```

3. По числовому индексу:

```
procedure MyProc3; stdcall; external 'MYLIB.DLL' index 3
```

Рекомендуется использовать импорт по имени экспорта, он более нагляден, хотя и работает немного медленнее, чем импорт по числовому индексу.

При динамическом импорте подпрограмм DLL загружается в память и выгружается самим приложением по необходимости. Динамическая загрузка происходит следующим образом. Вызывается функция LoadLibrary[1](LibFileName: PChar): HModule, где LibFileName – имя файла DLL. Если указано имя файла без маршрута, то DLL ищется в каталоге, из которого запущено приложение, затем в текущем каталоге, системном каталоге (Windows\System32), каталоге Windows, каталогах, указанных в системной переменной PATH. Функция возвращает системный дескриптор DLL (число) или код ошибки от 0 до 32:

```
var LibHandle: THandle;      //встроенный тип для дескрипторов
. . .
LibHandle := LoadLibrary('MathLib.dll');
if LibHandle < 32 then . . . //обработка ошибки
```

После загрузки DLL для получения адресов подпрограмм используется функция GetProcAddress(Module: HModule; ProcName: PChar): Pointer, где ProcName – имя подпрограммы. Если мы хотим произвести поиск подпрограмм по индексу, то два младших байта указателя PChar должны содержать

[1] Функции LoadLibrary, GetProcAddress, FreeLibrary объявлены в модуле Windows.

индекс, а два старшие – ноль. Если подпрограмма не найдена, функция GetProcAddress возвращает nil[1].

При динамическом импорте DLL в вызывающей программе необходимо объявить соответствующие процедурные типы. Приведем пример динамического импорта:

```
type TMin = function (X,Y: Integer): Integer;stdcall;
. . .
var Min: TMin
. . .
@Min := GetProcAddress(LibHandle, 'Min');
A := Min(B,C);
```

После завершения работы с библиотекой она освобождается вызовом функции FreeLibrary(LibModule: HModule): BOOL, где LibModule – дескриптор DLL.

Достоинствами динамического импорта DLL являются более эффективное использование ресурсов ОЗУ и возможность контролировать и обрабатывать ситуации, когда необходимой подпрограммы в DLL не оказалось.

DLL может содержать не только подпрограммы, но и ресурсы, в частности, описание одной или нескольких форм. Для помещения формы в DLL во время разработки DLL в IDE необходимо выполнить команду File|New form. Затем новая форма настраивается необходимым образом (такие формы обычно представляют диалоговые окна). Далее в исходный файл DLL добавляется процедура, которая показывает эту форму.

```
procedure ShowForm(AOwner: TComponent);stdcall;
begin
  Form1 := TForm1.Create(AOwner);
  Form1.ShowModal;
  Form1.Free
end;
. . .
exports ShowForm;
```

Эта процедура используется в основной программе:

```
procedure ShowForm(AOwner: TComponent); stdcall;
                                        external 'MYDLL.DLL';
. . .
ShowForm(Form1);
```

Рассмотрим некоторые нюансы при использовании динамических библиотек. Если DLL экспортирует подпрограммы, использующие в качестве параметров значения типа Variant, длинные строки и динамические массивы, эта DLL и использующее ее приложение должны подключать модуль ShareMem (причем первым в списке uses) и иметь доступ к DELPHIMM.DLL или BORLANDMM.DLL (поставляются в комплекте с Delphi).

---

[1] При использовании индекса в случае ошибки значение GetProcAddress может быть отлично от nil.

Если в подпрограмме DLL возникла исключительная ситуация, то объект исключительной ситуации передается в вызывающее приложение. Обработка исключительной ситуации возможна, если DLL использует модуль `SysUtils`.

Если в DLL объявлены свои глобальные переменные и константы, то доступ к ним из вызывающей программы возможен только через процедурный интерфейс.

Для главной программы, использующей DLL, обычно создается дополнительный интерфейсный модуль. В секции `interface` модуля указываются требуемые подпрограммы с директивой `stdcall`, а в секции `implementation` перечисляются с директивой `external` экспортируемые из DLL процедуры.

Вопросы и упражнения.

1. Проведите анализ проблем, которые могут возникнуть при размещении в DLL пользовательского класса. Для их иллюстрации используйте примеры.

2. Напишите с использованием DLL приложение, которое может работать с подключаемыми модулями (плагинам). Попытайтесь сделать работу с такими модулями максимально объектно-ориентированной.

18. Пакеты.

Пакет (package) – это расширенный вариант DLL, который может использоваться как программой, так и средой разработки Delphi. Фактически, пакет – это несколько скомпилированных модулей (`*.DCU`) и, возможно, некоторые ресурсы, специальным образом объединенные. Файлы пакетов имеют расширение `BPL`.

Исходный файл пакета имеет расширение `DPK`. Это обычный текстовый файл. Рассмотрим его структуру на примере:

```
package Sample;
{$R 'COMP.DCR'}    // подключение ресурсов (значка компонента)
{$DESCRIPTION 'Sample Components'}  // описание пакета
requires
  vcl50; //для Delphi 5
contains
  Comp in 'Comp.pas';
end.
```

Ключевое слово `package` обозначает начало исходного файла пакета. Директива `requires` обозначает начало списка пакетов, требуемых для компиляции исходного. Директива `contains` обозначает начало списка имен модулей, из которых состоит пакет. Имена модулей разделяются запятыми, каждый модуль представлен либо именем, либо именем модуля и именем файла в виде строки.

Как правило, редактирование исходного текста пакета вручную не производится. Для этого применяется Менеджер пакетов из IDE Delphi.

При помощи директивы компилятора `{$DesignOnly On}`, помещенной в исходный файл пакета, он может быть помечен как *пакет разработки* (design time packages). Такой пакет нельзя присоединить к приложению, а можно ис-

пользовать только при разработке приложения в IDE. По умолчанию директива выключена. При помощи директивы компилятора {$RunOnly On} пакет помечается как *пакет времени выполнения* (runtime packages). Такой пакет нельзя загрузить в IDE. Директивы не являются взаимоисключающими. По умолчанию установлено {$DesignOnly Off} и {$RunOnly Off} (пакет можно использовать и при разработке приложения и при выполнении).

Управление пакетами можно осуществить, вызвав диалоговую панель Packages (Project|Options|Packages или Component|Install Packages). Панель содержит две части. В первой находится список используемых IDE пакетов дизайна. Мы можем временно отключить использование пакета, а, значит, и входящих в него компонент, убрать пакет из IDE (кнопка Remove), или добавить новый скомпилированный пакет с компонентами (кнопка Add). Состав компонентов, входящих в пакет, можно изучить при помощи кнопки Components.

Вторая часть диалоговой панели управляет генерацией кода с подключением пакетов времени выполнения. Установка флага Build with runtime packages позволяет сгенерировать приложение, которое не включает в себя код используемых компонент и классов, а берет его во время выполнения из пакетов, перечисленных в диалоговом окне строкой ниже. Такое приложение будет малым по размеру (простейшее – около 18 Кбайт), но для своей работы оно будет требовать указанных в строке диалога пакетов. Главный пакет, содержащий основные компоненты, называется в Delphi 5 VCL50.BPL и имеет размер около 3,5 Мбайт.

Вопросы и упражнения.

1. Поместите собственный класс в пакет. Используйте данный пакет в IDE при создании произвольного приложения.

2. Возможна ли динамическая загрузка пакетов во время работы приложения? Изучите этот вопрос при помощи документации по Delphi и проиллюстрируйте примерами.

19. Создание простейших пользовательских компонент.

Одной из ключевых особенностей Delphi, хорошо иллюстрирующей преимущества объектно-ориентированного программирования, является возможность создания и использования новых пользовательских компонент. При этом благодаря наследованию, процесс создания компонента никогда не начинается «с нуля», а благодаря полиморфизму компонент органично встраивается в IDE.

Создание компонента – невизуальный процесс. Он включает следующие стадии.

1. Написание программного кода компонента.

2. Создание значка компонента для Палитры компонентов и файла справки компонента (необязательный этап).

3. Инсталляция компонента.

Проиллюстрируем этапы создания компонента на примере компонента TNewLabel – надписи, которая имеет отдельное свойство для изменения цвета шрифта.

Для получения модуля с «заготовкой» программного кода будущего компонента служит команда IDE File|New|Component. Заполним в появившемся диалоге следующие поля: Ancestor Type – TLabel, Class Name – TNewLabel, Pallete Page – Samples, Unit File Name – 'c:\work\newlabel'.

Дадим необходимые комментарии. Ancestor Type – класс предка. Если новый компонент дополняет возможности существующего, указывается класс существующего компонента (наш случай). Если компонент по содержанию радикально новый, то указываем: TComponent – для невизуальных компонент, TCustomControl – для компонент, которые должны получать фокус, TGraphicControl – для рисуемых компонент, TCustomXXX (TCustomLabel) – если компонент дополняет возможности существующего, но желательно уменьшить количество свойств и событий, видимых в Инспекторе объектов. Class Name – имя класса нового компонента. Pallete Page – страница Палитры компонентов для размещения нового компонента. Можно указать как существующую, так и новую страницу, которая будет автоматически создана. Unit File Name – имя файла с программным кодом компонента.

Модуль-заготовка компонента, создаваемый IDE, содержит каркас описания класса компонента и процедуру RegisterComponents для регистрации компонента в IDE. Наша задача – наполнить этот модуль содержимым.

Наш компонент должен содержать новое свойство, ответственное за цвет надписи. Назовем это свойство LabelColor. Его тип, очевидно, TColor. Так как оно должно быть видимо в Инспекторе объектов, объявим его в секции published. Для установки и чтения свойства используем методы SetLabelColor и GetLabelColor, описанные в секции private:

```
unit NewLabel;
interface
uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls;
type TNewLabel = class(TLabel)
     private
       procedure SetLabelColor(AValue: TColor);
       function GetLabelColor: TColor;
     published
       property LabelColor: TColor read GetLabelColor
                                    write SetLabelColor;
     end;
procedure Register;
implementation
procedure TNewLabel.SetLabelColor;
begin
  if AValue <> LabelColor then Font.Color := AValue;
end;
function TNewLabel.GetLabelColor;
begin
  Result := Font.Color;
```

```
end;
procedure Register;
begin
  RegisterComponents('Samples', [TNewLabel]);
end;
end.
```

Обратите внимание на синтаксис реализации методов и на отнесение методов по разделам видимости.

Разобраться с членами классов компонент, а также с разделами их видимости помогает Браузер объектов. Вызвать его можно после компиляции проекта, используя команду `View|Browser`. Браузер показывает дерево классов в левой части, правая часть содержит сгруппированные по разделам видимости атрибуты текущего класса. Для изучения компонентов используется также исходный код библиотеки VCL, поставляемый с редакцией Delphi Enterprise.

Установим компонент `TNewLabel` в IDE Delphi. Для этого требуется создание DCU–файла для модуля компонента, включение этого файла в существующий пакет или создание нового пакета, компиляция и инсталляция полученного пакета. Совокупность этих действия выполняется при вызове команды `Component|Install Component`. После выполнения команды в появившемся диалоговом окне можно выбрать инсталляцию в новый или существующий пакет. Выберем инсталляцию в новый пакет (`Into new package`), укажем имя модуля (`Unit file name`), имя пакета (`Package file name`, должно отличаться от имени модуля), описание пакета (`Package description`, заполнять не обязательно). После закрытия окна произойдет немедленная компиляция и установка пакета. На Палитре компонентов появится новый компонент.

Более сложный путь предполагает создание пакета вручную. Для этого выполняется команда `File|New|Package`. Появившееся окно содержит кнопки для добавления или удаления модулей в пакет, компиляции пакета, установки пакета и компонент на палитру.

Если компонент активно редактируется в процессе создания, желательно тестировать его, не инсталлируя. Пусть компонент храниться в файле `c:\work\NewLabel.pas`. Создадим маленький тестовый проект, содержащий форму и кнопку, при нажатии на которую на форме появляется компонент. Для этого достаточно в секции `uses` модуля формы подключить файл `NewLabel.pas`, а в обработчике нажатия кнопки создать и отобразить компонент:

```
procedure TForm1.Button1Click(Sender: TObject);
var L: TNewLabel;
begin
  L := TNewLabel.Create(self);     // создаем компонент
  L.Caption := 'Hi-Hi';            // настраиваем его
  L.LabelColor := clRed;
  L.Parent := Form1;               // теперь отображаем
end;
```

Рассмотрим еще один пример. Создадим компонент «Цифровые часы» (`TClock`) на основе текстовой метки. Чтобы не перегружать Инспектор объек-

тов свойствами, выберем в качестве класса-предка нашего компонента класс TCustomLabel. У класса TCustomLabel нет видимых в Инспектор объектов событий и очень мало видимых свойств. Полезные свойства и события в классе TCustomLabel скрыты в секции protected, в классе-наследнике их нужно перенести в секцию published.

Работа компонента TClock будет опираться на компонент TTimer. Основные свойства этого компонента: Enabled – включение таймера, Interval – через сколько миллисекунд происходит событие таймера OnTimer.

Наш компонент будет иметь поле для хранения объект класса TTimer. Создавать этот объект, настраивать его свойства и событие мы будем в конструкторе TClock.Create. Кроме этого, к модулю компонента необходимо подключить модуль extctrls, содержащий класс TTimer. Компонент TClock будет обладать собственным событием OnChange, которое происходит каждую секунду при включенном таймере:

```
unit Clock;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs, StdCtrls, extctrls;
type
  TClock = class(TCustomLabel)
  private
    fTimer: TTimer;           // содержит объект-таймер
    fOnChange: TNotifyEvent;  // хранит обработчик события
    // метод TimerTick обновляет показания часов
    procedure TimerTick(Sender: TObject);
    procedure SetEnabled(AValue: Boolean);
  public
    constructor Create(AOwner: TComponent);override;
  published
    property Enabled: Boolean read fTimer.Enabled
                              write SetEnabled default True;
    property Font;   // перенесено из protected в TCustomLabel
    property OnChange: TNotifyEvent read FOnChange
                                    write FOnChange;
  end;
. . .
constructor TClock.Create;
begin
  inherited;
  fTimer := TTimer.Create(self);      // создаем таймер
  fTimer.Interval := 1000;            // настраиваем его
  fTimer.OnTimer := TimerTick;
  fTimer.Enabled := True;
  Caption := TimeToStr(Time);         // обновим показания
end;
procedure TClock.TimerTick;
begin
  Caption := TimeToStr(Time);         // выведем текущее время
  if Assigned(fOnChange) then fOnChange(self);
```

```
end;
procedure TClock.SetEnabled;
begin
  if AValue <> fTimer.Enabled then
    begin
      fTimer.Enabled := AValue;       // добираемся до таймера
      if fTimer.Enabled then Caption := TimeToStr(Time);
    end
end;
```

Этот пример продемонстрировал перенос свойств из одной области видимости в другую и создание собственных обработчиков событий.

20. Компоненты, являющиеся наследниками TGraphicControl.

Рассмотрим пример рисуемого компонента, порожденного от класса `TGraphicControl`. Напомним, что класс `TGraphicControl` имеет виртуальный метод `Paint`. Его можно перекрывать для собственного рисования компонента. В методе `Paint` доступно свойство класса `Canvas`. Опишем компонент `TArrow` для представления горизонтальной стрелки:

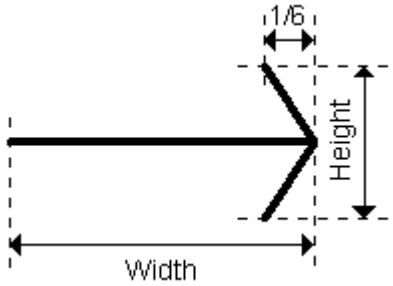

Рис. 3. Внешний вид и размеры стрелки.

```
unit Arrow;
. . .
type
  TDirection = (drLeft, drRight);
  TArrow = class(TGraphicControl)
  private
    fDirection: TDirection;
    procedure SetDirection(ADirection: TDirection);
  protected
    procedure Paint;override;
  public
    constructor Create(AOwner: TComponent);override;
  published
    property Height default 20;
    property Width default 60;
    property Color;
    property Direction: TDirection read fDirection
                                   write SetDirection
                                   default drRight;
  end;
. . .
```

```
constructor TArrow.Create;
begin
  inherited;
  Height := 20;
  Width := 60;
  Direction := drRight
end;
procedure TArrow.SetDirection;
begin
  if ADirection <> fDirection then
    begin
      fDirection := ADirection;
      Invalidate
    end
end;
procedure TArrow.Paint;
var dx, dy: Integer;
begin
  inherited;
  dx := Width div 6;
  dy := Height div 2;
  with Canvas do begin
    if csDesigning in ComponentState then begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle(0,0,Width,Height)
    end;
    Pen.Style := psSolid;
    Pen.Color := Color;
    case Direction of
      drRight: PolyLine([Point(Width-dx,0), Point(Width,dy),
                         Point(Width-dx,Height),
                         Point(Width,dy), Point(0,dy)]);
      drLeft: PolyLine([Point(dx,0), Point(0,dy),
                        Point(dx,Height),
                        Point(0,dy), Point(Width,dy)]);
    end
  end
end;
```

Обратите внимание на следующие моменты. Для задания направления стрелки введен перечисляемый тип `TDirection`. Работа с перечисляемыми типами полностью поддерживается Инспектором объектов, мы сможем выбирать значения свойства из выпадающего списка значений. Свойства `Width` и `Height` перенесены из секции `protected` в секцию `published` с указанием нового значения по умолчанию. В методе `Paint` использовано свойство `ComponentState` класса `TComponent`. Это свойство позволяет контролировать, в каком режиме находится компонент. В данном случае `ComponentState` используется для рисования пунктирной линии на границе компонента в режиме проектирования формы.

ЧАСТЬ II. ЯЗЫК VISUAL BASIC .NET.

1. Платформа .NET.

Задача *платформы .NET* (*.NET Framework*) – предоставить программистам более эффективную и гибкую среду разработки традиционных и Web-приложений. Одна из наиболее важных особенностей .NET Framework – способность обеспечить совместную работу кода, написанного на различных языках программирования. На рис. 4 показана структура платформы .NET на самом высоком уровне.

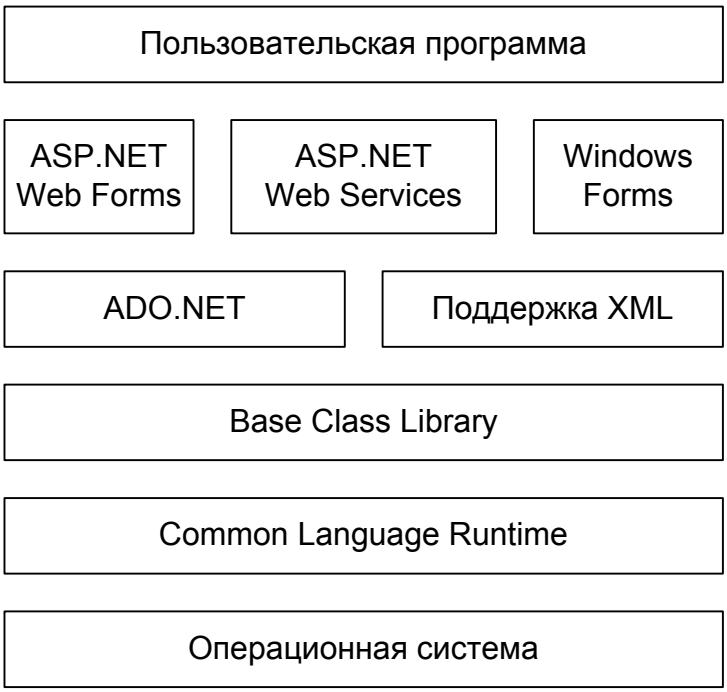

Рис. 4. Общая структура .NET Framework

Базой платформы является *общеязыковая среда исполнения* (*Common Language Runtime*, *CLR*). CLR является «прослойкой» между операционной системой и кодом приложений для .NET Framework. Такой код называется *управляемым* (*managed code*). Более подробно роль CLR обсуждается далее.

В состав платформы .NET входит библиотека классов *Framework Class Library* (*FCL*). Элементом этой библиотеки является базовый набор классов *Base Class Library* (*BCL*). В BCL входят классы для работы со строками, коллекциями данных, поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML, базами данных (ADO.NET), создания Windows-приложений и Web-приложений (ASP.NET).

В стандартную поставку .NET Framework включены компиляторы для платформы. Это компиляторы языков C#, Visual Basic.NET, J#. Благодаря открытым спецификациям компиляторы для .NET предлагаются различными сто-

ронними производителями (не Microsoft). На данный момент количество компиляторов измеряется десятками.

Рассмотрим подробнее компоненты и роль CLR. Любой компилятор для .NET позволяет получить из исходного текста программы двоичный исполняемый файл или библиотеку кода. Однако эти файлы по своей структуре и содержанию не имеют ничего общего с традиционными исполняемыми файлами операционной системы. Двоичные файлы для платформы .NET называются *сборками* (*assembly*). Сборка состоит из следующих частей:

1. *Манифест* (*manifest*) – описание сборки: версия, ограничения безопасности, список внешних сборок и файлов, необходимых для работы данной сборки.

2. *Метаданные* – специальное описание всех пользовательских типов данных, размещенных в сборке.

3. *Код на промежуточном языке Microsoft Intermediate Language* (*MSIL* или просто *IL*). Данный код является независимым от операционной системы и типа процессора, на котором будет выполняться приложение. В процессе работы приложения он компилируется в машинно-зависимый код специальным компилятором (*Just-in-Time compiler*, *JIT compiler*).

Основная задача CLR – это манипулирование сборками: загрузка сборок, трансляция кода IL в машинно-зависимый код, создание окружения для выполнения сборок. Важной функцией CLR является управление размещением памяти при работе приложение и выполнение *автоматической сборки мусора*, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET верификацию типов, управление политиками безопасности при доступе к коду и некоторые другие функции.

Кроме упомянутых элементов, выделим еще две части платформы .NET:

- *Система типов данных* (*Common Type System*, *CTS*) – базовые, не зависящие от языка программирования примитивные типы, которыми может манипулировать CLR.
- *Набор правил* для языка программирования (*Common Language Specification*, *CLS*), соблюдение которых обеспечивает создание на разных языках программ, легко взаимодействующих между собой.

В заключение хотелось бы подчеркнуть, что любой компилятор для .NET является верхним элементом архитектуры. Библиотека классов FCL, имена ее элементов не зависят от языка программирования. Специфичным элементом языка остается только синтаксис, но не работа с внешними классами. Это упрощает межъязыковое взаимодействие, перевод текста программы с одного языка на другой. С другой стороны, тесная связь с CLR неизбежно находит свое отражение в синтаксических элементах языка программирования.

2. Примитивные и пользовательские типы Visual Basic.NET.

Язык Visual Basic.NET (далее VB.NET) является одним из языков программирования для платформы .NET. Это объектно-ориентированный язык, соответствующий CTS и CLS.

Основой VB.NET является развитая система типов. Проведем ее классификацию. Все типы, присутствующие в языке, можно разделить на *примитивные* и *пользовательские*. Примитивным назовем тот тип, множество значений которого определено и постоянно. Примитивный тип присутствует в объектной библиотеке и не описывается пользователем (примитивный тип – это основа для конструирования пользовательского типа). Обычно поля пользовательских типов имеют значения одного из примитивных типов, примитивные типы также используются для локальных переменных в программных блоках. Примитивные типы включают числовые типы, логический тип, тип для представления даты и времени, символьный и строковый типы. Информация о примитивных типах сведена в таблицу 1:

Таблица 1

Примитивные типы языка VB.NET

Имя типа	**Размер переменной (байт)**	**Диапазон значений**
`Byte`	1	$0..2^8-1$
`Short`	2	$-2^{15}.. 2^{15}-1$
`Integer`	4	$-2^{31}.. 2^{31}-1$
`Long`	8	$-2^{63}.. 2^{63}-1$
`Single`	4	Точность – 8 знаков после запятой
`Double`	8	Точность – 16 знаков после запятой
`Decimal`	16	28 цифр с произвольной позицией десятичной запятой [1]
`Boolean`	4	`True` или `False`[2]
`Date`	8	От 01.01.0001 до 31.12.9999 (и время)
`Char`	2	0..65535 (коды Unicode)
`String`	4 (указатель)	до 2 Гб символов Unicode

Ключевое слово, имеющиеся в VB.NET для обозначения примитивного типа, является коротким синонимом имени типа из CTS (например, тип `Date` – это синоним типа `System.DateTime`, тип `Short` – синоним типа `System.Int16`).

Пользовательские типы названы так потому, что их компоненты описываются пользователем. Можно утверждать, что любая программа на языке VB.NET представляет собой набор пользовательских типов. Перечислим возможные пользовательские типы и опишем ту функциональность, которой они обладают.

1. **Класс** – тип, который поддерживает всю функциональность ООП, включая наследование и полиморфизм. Компонентами класса могут являться

[1] Один бит хранит знак, следующие 96 бит хранят значения числа, следующие 8 бит – степень числа 10, на которое делится 96-разрядное число (может быть в диапазоне от 0 до 28). Остальные биты не используются.

[2] Внутреннее представление значения `True` – это константа `-1` (`&HFFFFFFFF` в шестнадцатеричной системе счисления); внутреннее представление значения `False` – константа `0`.

поля, методы, константы, свойства, события. Класс также может включать описания некоторых других пользовательских типов.

2. **Структура** – тип, обеспечивающий всю функциональность ООП, кроме наследования. Структура в VB .NET очень похожа на класс, за исключением метода размещения в памяти и отсутствия поддержки наследования.

3. **Модуль** – пользовательский тип, все члены которого являются разделяемыми, то есть доступными без создания переменных типа. Модуль не поддерживает наследование. Модуль обычно является «обрамляющим» пользовательским типом в программе. Таким образом, программа, как правило, представляет собой модуль, который содержит такие компоненты, как поля (в традиционном программировании они соответствуют глобальным переменным), методы (в традиционном программировании – подпрограммы) и вложенные описания пользовательских типов (классов, структур и т. п.).

4. **Интерфейс** – абстрактный тип, реализуемый другими типами для обеспечения оговоренной функциональности.

5. **Массив** – пользовательский тип для представления упорядоченного набора значений некоторых (примитивных или пользовательских) типов.

6. **Перечисление** – тип, содержащий в качестве членов именованные константы.

7. **Делегат** – пользовательский тип для представления ссылок на методы-обработчики событий.

С точки зрения размещения переменных типов в памяти все типы можно разделить на *структурные* и *ссылочные*. Переменная ссылочного типа является неким аналогом динамической переменной. Перед использованием она должна быть особым образом инициализирована и размещена в динамической памяти. Все переменные ссылочного типа фактически являются указателями. Переменные структурного типа соответствуют обычным переменным и в особой инициализации не нуждаются. Все примитивные типы, за исключением `String`, являются структурными. Структура – это также структурный тип. Строковый тип, класс, модуль, делегат, массив и интерфейс – ссылочные типы (напомним, что переменные модуля никогда не создаются).

3. Основные концепции синтаксиса Visual Basic .NET.

Опишем базовые концепции синтаксиса языка Visual Basic .NET. Исходный код программы на VB .NET размещается в одном или нескольких текстовых файлах (стандартное расширение файлов – `*.vb`). Такой файл представляет собой набор пустых и интерпретируемых строк. Максимальная длина одной интерпретируемой строки в спецификации языка не задана. Интерпретируемые строки можно разделить на строки кода и строки комментариев. Обычно одна строка кода содержит одну команду языка. Строчные и прописные символы при записи идентификаторов и ключевых слов не различаются. Если для размещения команды одной строки недостаточно, то команду можно продолжить в следующей строке. При этом предыдущая строка завершается пробелом и символом подчеркивания. Таким образом, переход на новую строку обычно является признаком конца команды, специальных символов для разделения команд

нет. Несколько команд можно разместить в одной строке, разделив их двоеточием, однако обычно так не поступают. Количество пробелов в начале строки, в конце строки и между элементами строки значения не имеет, чем обычно пользуются для улучшения структурированности кода.

Программа может содержать комментарии, игнорируемые при компиляции. Комментарии начинаются с апострофа (') и продолжаются до конца строки, либо вводятся командой `Rem`.

В каждом исходном файле программы можно выделить необязательную заголовочную секцию. Эта секция находится в начале файла и содержит опции компилятора, распространяющиеся на файл и инструкции импортирования пространств имен. Пространства имен будут подробно рассмотрены далее, их можно воспринимать как средство группировки типов. Инструкции импортирования позволяют сократить запись обращения к членам пространств имен.

Любая программа на VB .NET имеет специальную точку входа, с которой начинается выполнение приложения. Такой точкой входа является метод `Main()`, объявленный в неком модуле или классе. Только один модуль или класс может иметь метод `Main()`.

В качестве примера рассмотрим «классическую» программу «`Hello World`», написанную на VB .NET и реализованную в виде консольного приложения (исходный код разместим в файле `MyModule.vb`):

```
Module MyModule
    Sub Main()
        System.Console.WriteLine("Hello World!")
        System.Console.ReadLine()
    End Sub
End Module
```

Опции компилятора и инструкции импортирования в данной программе не используются. Программа представляет собой модуль `MyModule` (пользовательский тип), который содержит единственный метод-процедуру `Main()`. Метод `Main()` включает два вызова методов класса `Console` из пространства имен `System`. Метод `WriteLine()` выводит строку на консоль, метод `ReadLine()` ожидает ввода с консоли, завершающегося нажатием «Enter».

4. Идентификаторы и литералы.

Идентификатор – это пользовательское имя для переменной, константы, метода или типа. В VB.NET идентификатор – это произвольная последовательность букв, цифр и символов подчеркивания общей длиной до 16383, начинающаяся с буквы. Идентификатор должен быть уникальным внутри области использования. Он не может совпадать с ключевым словом языка. Однако если заключить ключевое слово в квадратные скобки, то получившийся *escape-идентификатор* является допустимым. Примеры допустимых идентификаторов: `Temp`, `Some_Variable`, `[Module]`.

В VB.NET *литерал* – это последовательность символов, которая может интерпретироваться как значение одного из примитивных типов. Так как язык VB.NET является языком со строгой типизацией, иногда необходимо явно ука-

зать, к какому типу относится последовательность символов, определяющая данные. Для этого используется либо один из суффиксов, либо некие обрамляющие символы[1].

Таблица 2

Признаки литералов

Тип данных	Признак типа в литерале	Пример
Byte	Нет	
Short	Суффикс S	237S
Integer	Суффикс I или %	237%
Long	Суффикс L или &	237L
Single	Суффикс F или !	3F
Double	Суффикс R или #	0.5#
Decimal	Суффикс D или @	123@
Boolean	Два возможных литерала: True и False	True
Date	Обрамляющие # и #	#01/30/2002#
Char	Обрамляющие кавычки (") и суффикс C	"H"C
String	Обрамляющие кавычки	"Alex"

Если указано целое число без суффикса, то подразумевается тип Integer. Любой целочисленный литерал можно записать в шестнадцатеричной системе счисления, использовав префикс &H, или в восьмеричной системе счисления, использовав префикс &O.

Если в числе с десятичной точкой не указан суффикс, то подразумевается тип Double. Число с плавающей точкой может быть записано в научном формате: 3.5E-6, -7E10, .6E+7.

Если одиночный символ просто поместить в кавычки, без указания суффикса C, то это будет означать строку из одного символа. Для обозначения символа " (кавычки) в строковом или символьном литерале этот символ дублируют (""). Так как символьный тип хранит коды Unicode, то для определения символа по коду используют специальные функции, например Microsoft.VisualBasic.Chr(): Microsoft.VisualBasic.Chr(&H53) это символ S.

Литерал для дат может содержать дату и/или время. Дата записывается в формате месяц/число/год или в формате месяц-число-год. Время записывается в формате часы:минуты:секунды, возможно с суффиксами AM или PM (минуты и секунды могут отсутствовать).

5. Уровни доступа к компонентам пользовательских типов.

При написании программ часто возникает необходимость разграничить доступ к отдельным членам пользовательского типа. Например, скрыть поля класса, ограничить доступ к методу, описанному в некотором модуле и тому

[1] Использование небуквенных суффиксов (%, &, !, #, @) поддерживается только для обеспечения обратной совместимости с Visual Basic 6.

подобное. Для этих целей применяют модификаторы доступа, указываемые непосредственно перед объявлением типа или члена типа. Возможно использование следующих модификаторов.

`Private`. Элемент с данным модификатором (тип или члена типа) видим только в том типе, в котором определен (например, поле видно только в описываемом классе). Для полей и констант пользовательских типов `Private` является модификатором по умолчанию. С этим модификатором можно объявить класс и структуру (если они вложены в другой тип), поле, метод, константу, свойство, событие. К модулям данный идентификатор не применим.

`Public`. Элемент доступен без ограничений как в той сборке, где описан, так и в других сборках, к которым подключается сборка с элементом.

`Friend`. Элемент доступен без ограничений, но только в той сборке, где описан. Этот модификатор применяется по умолчанию для методов класса, структуры и модуля. Так же это модификатор по умолчанию для модуля в целом.

`Protected`. Элемент с данным модификатором видим только в типе, в котором определен, и в наследниках данного типа (даже если эти наследники расположены в других сборках). Данный модификатор может применяться только для типов, поддерживающих наследование, то есть для классов.

`Protected Friend`. Комбинация модификаторов `Protected` и `Friend`. Элемент виден в сборке без ограничений, а вне сборки – только в наследниках типа.

Для локальных переменных методов и программных блоков не используются все выше перечисленные модификаторы доступа.

Подчеркнем, что модификаторы `Private` и `Protected` могут применяться только к *членам* пользовательского типа. Для «внешних» (обрамляющих) пользовательских типов данные модификаторы не приемлемы.

6. Объявление переменных, полей и констант.

Команда для объявления переменных в языке VB.NET может использоваться в двух контекстах. В первом контексте данная команда используется на уровне подпрограммы и объявляет *локальную переменную* подпрограммы. Во втором контексте она используется на уровне описания пользовательского типа (класса, структуры или модуля). В этом случае правильнее было бы говорить об объявлении *поля* типа. Для объявления переменных и полей в VB .NET используется команда `Dim`. В простейшем случае формат этой команды следующий[1]:

```
Dim <имя переменной или поля> As <тип> [= <нач. значение>]
```

где `Dim` и `As` – ключевые слова, `<имя переменной или поля>` – допустимый идентификатор, `<тип>` – определяет тип переменной, `<нач. значение>` – литерал, соответствующий типу переменной и задающий начальное значение. Если

[1] В дальнейшем используются следующие соглашения по записи синтаксиса. Элементы в квадратных скобках [и] являются необязательными. Элементы в фигурных скобках, разделенные вертикальной чертой – обязательные элементы, чертой разделены возможные значения элемента.

начальное значение переменной не задано, то переменная получает значение 0 для числовых типов, False для логического типа, символ с кодом 0 для символьного типа, 0 часов 0 минут 1 января 1 года для типа Date. Любой ссылочный тип, включая строки и массивы, получает специальное значение Nothing[1]. Начальное значение может быть задано как для полей пользовательских типов, так и для локальных переменных методов.

Если для поля пользовательского типа используется модификатор доступа, то он может применяться или перед командой Dim, или вместо нее. Локальные переменные методов и программных блоков объявляются только при помощи команды Dim.

Если необходимо объявить несколько переменных или полей одного типа, то идентификаторы переменных можно перечислить через запятую между Dim и As, однако начальная инициализация переменных при этом выполняться не может.

Рассмотрим примеры объявления переменных:

```
Dim Age As Integer             'Простейший вариант объявления
Dim Age As Integer = 20        'Объявления с инициализацией
Dim A, B, C As Double          'Объявление нескольких переменных
Private Dim Age As Integer     'Объявлено поле с модификатором
Private Age As Integer         'Dim для полей можно не писать
```

Одной командой Dim можно объявить переменные или поля нескольких типов, повторив список переменных с ключевым словом As необходимое количество раз:

```
Dim Age As Integer = 20, A, B, C As Double
```

В отличие от языка Pascal, VB.NET разрешает использовать локальные переменные без объявления. Но так поступать не рекомендуется во избежание ошибок разнотипного присваивания. Для того чтобы запретить или разрешить использование переменных без предварительного объявления, служит следующая опция компилятора:

```
Option Explicit {On|Off}
```

Если в заголовочной секции файла установлено Option Explicit On, то использование переменных без объявления запрещено. Такое значение установлено по умолчанию. Option Explicit Off разрешает использование необъявленных переменных.

Для локальных переменных методов (и только для локальных переменных) возможно использование специального модификатора Static, определяющего *статическую переменную*. Такая переменная сохраняет свое значение после окончания работы метода до следующего вызова этого метода. Если указать для статической переменной начальное значение, инициализация будет проведена только при первом вызове метода.

```
'Это объявление статической переменной
Static Dim Count As Integer
```

[1] Некий аналог nil из Object Pascal.

```
'Это тоже, но с инициализацией
Static Count2 As Integer = 100
```

Как и другие языки программирования, VB.NET позволяет описать в пользовательском типе или методе константы. Синтаксис объявления константы следующий:

```
Const <имя константы> [As <тип константы>] = <выражение>
```

Тип константы – это любой примитивный тип. Если тип константы не указан, он распознается по типу выражения, задающего константу. При использовании опции компилятора `Option Strict On` тип константы указывать обязательно. `<Выражение>` может быть литералом или результатом действий с другими константами. Примеры объявления констант:

```
Const Pi As Double = 3.1415926
Const Pi2 As Double = Pi + 2
```

Для полей пользовательских типов возможно применение модификатора `ReadOnly`, который фактически превращает их в константу. Однако в отличие от констант, тип такого поля может быть любым:

```
ReadOnly Dim Age As Integer = 20
```

Для полей классов, которые объявлены с модификатором `ReadOnly`, начальное значение может устанавливать только конструктор (но оно может быть указано и при объявлении поля). Использование модификатора `ReadOnly` для локальных переменных запрещено.

7. Операции.

В VB.NET существует стандартный набор операций над данными и переменными примитивных типов. Можно выделить следующие типы операций: *математические*, *логические* и *операцию конкатенации* для строк.

Набор математических операций стандартен: сложение (+), вычитание (-), перемена знака (унарный минус -), умножение (*), деление (/), возведение в степень (^). Результатом деления всегда является значение типа `Double`, даже если тип операндов целый. Для целых типов определены операции целочисленного деления (\) (результат – целого типа), нахождения остатка (`Mod`) и побитовые логические операции (`And`, `Or`, `Xor`, `Not`). Если в качестве операндов выступают значения разных типов, то тип результата операции – это «наибольший» тип операнда. Деление на 0 для вещественных типов не вызывает ошибку – результатом являются специальные значения `infinity` или `NaN`[1].

Логические операции представлены традиционным набором `And`, `Or`, `Xor`, `Not`. Существуют аналоги операций `And` и `Or` для вычисления логического выражения по *сокращенной схеме*: `AndAlso` и `OrElse` (например, если первый операнд при использовании `AndAlso` имеет значение `False`, то второй не вычисляется).

[1] То есть «бесконечность» при делении на ноль и «не число» (not a number), если ноль делится на ноль.

VB.NET поддерживает стандартный набор операций отношения: <, >, <=, >=, =, <>. Специальная операция Like, используемая в форме <строка> Like <шаблон>, позволяет установить соответствие строки некоему шаблону и в результате совпадения возвращает True. *Шаблон* – это строка, которая может включать следующие специальные символы:

? – на этом месте в строке может быть любой символ;

* – на этом месте в строке может быть любая группа символов;

– на этом месте в строке может быть любая цифра;

[charlist] – любой символ из группы charlist, элементы которой разделяются запятой или символом "-";

[!charlist] – любой символ вне группы charlist.

Приведем несколько примеров использования Like:

```
"Alex" Like "A*x" = True
"Alex" Like "A[!a-d,k,m]ex" = True
"Alex" Like "A[a-z]#x" = False
```

Имеется специальная опция компилятора, которая влияет на выполнения операции Like. При использовании Option Compare Text строчные и прописные буквы при сравнении не различаются. Использование Option Compare Binary, установленной по умолчанию, ведет к различию строчных и прописных букв.

Операция отношения Is позволяет установить, указывают ли объектные ссылки на один и тот же объект, и используется при объектно-ориентированном программировании.

Для строк возможно использование единственной операции конкатенации (сцепления). Символом операции является &. Можно использовать символ +, однако это не принято:

```
"Hello " & "World!" = "Hello World!"
```

При выполнении нескольких операций порядок определяется скобками, а при их отсутствии – приоритетом операции. Сведения о приоритете операций представлены в таблице 3.

Таблица 3

Приоритет операций в VB.NET

Высший приоритет	^	=	Not
	- (унарный)	<>	
	*, /	<, >	And, AndAlso
	\	<=	
	Mod	>=	
	+, -	Like	Or, OrElse, Xor
Низший приоритет	&	Is	

8. Оператор присваивания. Преобразование типов.

Оператор присваивания – один из самых распространенных в любом языке. Синтаксис оператора присваивания в VB.NET следующий:

```
<имя переменной> =<выражение,литерал,переменная,вызов функции>
```

Допустима комбинация бинарных математических операций (кроме `Mod`) и операции конкатенации с оператором присваивания: `^=`, `*=`, `/=`, `\=`, `-=`, `+=`, `&=` (запись `i += 1` означает `i = i+1`).

При проведении различных операций (арифметических, присваивания) возникает проблема совмещения значений разных типов. В VB.NET предусмотрен режим жесткой проверки совместимости типов, включаемый опцией компилятора `Option Strict On`. Данный режим рекомендуется включать в заголовочной секции любого файла. По умолчанию режим является выключенным (`Option Strict Off`). Включение режима жесткой проверки типов неявно ведет к включению опции `Option Explicit On`[1].

При активизации режима жесткой проверки типов любые преобразования типов, которые могут привести к потере точности, запрещены. Разрешены только так называемые *расширяющие преобразования*. Для понимания логики расширяющих преобразований можно использовать схему на рис. 5[2]:

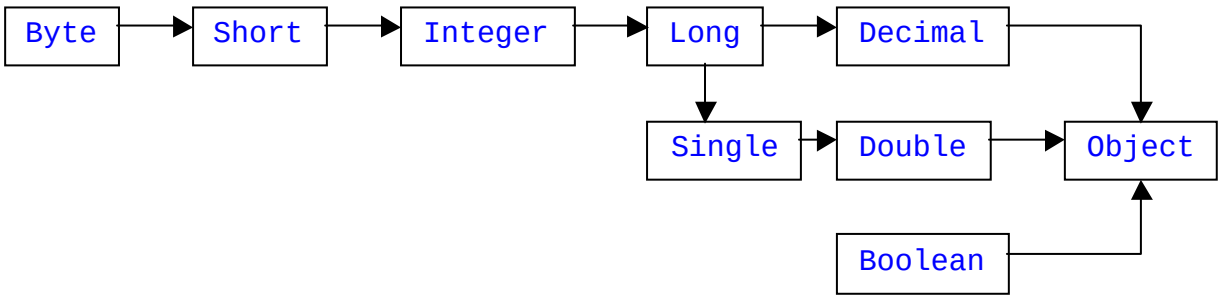

Рис. 5. Схема расширяющих преобразований типов

Преобразование типа `A` в тип `B` возможно, если на схеме существует путь от `A` к `B`. Обратите внимание, что значения типа `Decimal` нельзя преобразовывать в значения вещественных типов и наоборот.

Если режим жесткой проверки типов включен, но необходимо выполнить не расширяющее преобразование, можно использовать *явное преобразование* типов, использовав одну из перечисленных в табл. 4. встроенных функций.

Таблица 4

Встроенные функции преобразования типов

Функция	Тип значения	Тип аргумента
`CBool`	`Boolean`	Любой числовой тип, `String`, `Object`
`CByte`	`Byte`	Любой числовой тип, `Boolean`, `String`, `Object`
`CShort`	`Short`	Любой числовой тип, `Boolean`, `String`, `Object`
`CInt`	`Integer`	Любой числовой тип, `Boolean`, `String`, `Object`

[1] Всего в языке VB.NET три опции компилятора: `Option Explicit`, `Option Compare` и `Option Strict`.

[2] Преобразование типа `Long` к типу `Single` все-таки может привести к потере точности!

CLng	Long	Любой числовой тип, Boolean, String, Object
CSng	Single	Любой числовой тип, Boolean, String, Object
CDbl	Double	Любой числовой тип, Boolean, String, Object
CDec	Decimal	Любой числовой тип, Boolean, String, Object
CDate	Date	String, Object
CChar	Char	String, Object
CStr	String	Любой числовой тип, Boolean, Char, массив Char(), Date, Object
CObject	Object	Любой тип

Встроенная функция CType имеет два аргумента: первый – это преобразовываемая переменная или литерал, второй – имя целевого типа. Результат работы функции CType имеет указанный целевой тип. Если выполнить преобразование с помощью вышеуказанных функций невозможно, возникает контролируемая исключительная ситуация.

С отключенным режимом жесткой проверки типов можно использовать объявление переменной без указания типа. В этом случае делается попытка распознать тип переменной по типу выражения, задающего ее начальное значение. Если такого выражения нет, переменная получает тип Object (универсальный тип, базовый тип для любых других):

```
'Установлено Option Strict Off
Dim Age = 27 'Age имеет тип Integer
Dim SomeVar 'SomeVar имеет тип Object
```

9. Операторы переходов.

В языке Visual Basic .NET описания методов, свойств, синтаксические конструкции операторов ветвления и циклов образуют в тексте *программные блоки*. Для принудительного выхода из программного блока используется оператор Exit. Данный оператор существует в особых вариантах для каждого программного блока:

```
Exit {Select|For|While|Do|Sub|Function|Property|Try}
```

Рассмотрим оператор безусловного перехода. Его выполнение ведет к передаче управления на указанную строку внутри блока кода (передавать управление из одного метода в другой не разрешено). Синтаксис оператора следующий:

```
GoTo <метка строки>
```

Для использования оператора GoTo строка кода, на которую передается управления, должна быть помечена. *Метка* – это идентификатор, заканчивающийся двоеточием. Метка должна предшествовать командам, размещенным в строке, или располагаться на отдельной строке. Отдельно описывать метки нигде не надо:

```
GoTo Lab1
. . .
Lab1:
A = 100
```

Использование операторов безусловного перехода не приветствуется, так как усложняет структуру и понимание программы.

Оператор условного перехода существует в VB.NET в нескольких формах. Простейшая форма называется *однострокoвой*. Ее синтаксис следующий:

```
If <условие> Then <команды-1> [Else <команды-2>]
```

где `<условие>` – это некое логическое выражение. Ветвь `Else` является необязательной.

Если при ветвлении необходимо выполнить несколько команд, обычно используется *блочная* форма оператора:

```
If <условие> [Then]
    [<команды-1>]
[Else
    [<команды-2>]]
End If
```

Обратите внимание, что при использовании блочной формы ключевое слово `Then` указывать не обязательно. Если имеется несколько вложенных операторов условного перехода, используется синтаксис с ключевым словом `ElseIf`, обозначающим вложенное условие:

```
If <условие> [Then]
    [<команды-1>]
[ElseIf <условие-2> [Then]
    [<команды-2>]]
[Else
    [<команды-3>]]
End If
```

Рассмотрим пример использования оператора условного перехода:

```
If A < 0 Then
    Console.WriteLine("Отрицательное число")
ElseIf A > 0 Then
    Console.WriteLine("Положительное число")
Else
    Console.WriteLine("Ноль")
End If
```

Оператор выбора `Select Case` выполняет одну из групп инструкций в зависимости от значения тестируемого выражения. Синтаксис оператора выбора следующий:

```
Select [Case] <тестируемое выражение>
    [Case <список-1>
         [<команды-1>]]
. . .
    [Case <список-n>
         [<команды-n>]]
    [Case Else
         [<команды-n+1>]]
End Select
```

Ключевое слово `Case` в заголовке оператора можно не указывать. После каждой ветви `Case` идет список, элементы которого разделены запятыми. Каждый элемент списка может иметь следующий вид:

`<выражение>` – любое числовое или строковое выражение;

`<выражение-1> To <выражение-2>` – диапазон значений (`<выражение-1>` должно быть меньше `<выражения-2>`);

`Is <оператор сравнения> <выражение>` – диапазон значений.

Приведем пример использования оператора `Select Case`:

```
Select Case Nomer
    Case 1, 3, 5, 7
         Console.WriteLine("Нечетное")
    Case 10 To 99
         Console.WriteLine("Двузначное")
    Case Is < 0, 100 To 999
         Console.WriteLine("Отрицательное или трехзначное")
End Select
```

Если списки, определенные в разных ветвях `Case`, задают пересекающиеся диапазоны, то выполняется первая ветвь с подходящим диапазоном. При использовании в списке сравнения конструкции с `Is` недопустима операция `Like`.

10. Операторы организации циклов.

VB.NET предоставляет разнообразный набор операторов для организации циклов. Для циклов с определенным числом итераций используется оператор `For - Next`. Его синтаксис:

```
For <счетчик> = <нач. знач.> To <кон. знач.> [Step <шаг>]
    [<команды>]
Next [<счетчик>]
```

В качестве счетчика цикла возможно использование переменной числового типа. Если шаг цикла не задан, он полагается равным 1. Возможен досрочный выход из цикла при помощи конструкции `Exit For`. После ключевого слова `Next` не обязательно указывать имя счетчика.

Если начальное значение, конечное значение и шаг задаются выражениями, то эти выражения вычисляются один раз при организации цикла. Внутри цикла допустимо изменять значение счетчика, однако делать это не рекомендуется.

Если используются вложенные циклы `For - Next`, то их можно завершить единственным ключевым словом `Next`, указав через запятую значения счетчиков циклов. При этом необходимо, чтобы счетчики в списке шли по порядку от внутренних циклов к внешним. Вложенные циклы должны иметь уникальные переменные-счетчики.

Формально считается, что значение переменной-счетчика после выхода из цикла не определено. Однако фактически оно равно последнему значению счетчика в цикле.

Для организации циклов, число повторов в которых заранее не известно, можно использовать оператор `While`. Он выполняет последовательность команд, пока условие имеет истинное значение:

```
While <условие>
    [<команды>]
End While
```

Оператор `Do - Loop` позволяет, как и оператор `While`, организовывать циклы с заранее неизвестным количеством повторов. Он может использоваться в следующих синтаксических модификациях:

```
Do {While|Until} <условие>
    [<команды>]
Loop
```

или

```
Do
    [<команды>]
Loop {While|Until} <условие>
```

Данные формы различаются местом проверки условия цикла. В первом случае оно проверяется до входа в цикл (цикл может не выполниться ни разу), во втором случае условие проверяется после выполнения тела цикла (хотя бы один раз тело цикла выполнится). Если при условии используется ключевое слово `While`, цикл выполняется пока условие истинно, если ключевое слово `Until` – пока условие ложно. В любой момент из цикла можно выйти, употребив конструкцию `Exit Do`.

Рассмотрим пример использования оператора `Do - Loop`:

```
A = 3
Do
    Console.WriteLine(A)
    A += 2
Loop While A < 10
```

В операторе `Do - Loop` условие `While|Until` может отсутствовать. В этом случае цикл выполняется бесконечно:

```
Do
    'Бесконечный цикл. Выход - командой Exit Do
Loop
```

Для оператора условного перехода, оператора выбора и операторов цикла справедливо следующее замечание. Каждый из этих операторов содержит блок или несколько блоков кода. Можно объявить в таких блоках переменные. Однако за пределами блока переменные будут недоступны:

```
Do Until X < 100
    Dim Var As Integer
    . . .
Loop While A < 10
Var = 100 'Вызовет ошибку, Var недоступна вне блока цикла
```

Компилятор не допускает, чтобы локальная переменная блока имела такое же имя, как и переменная вне блока:

```
Dim Var As Integer
Do Until X < 100
    Dim Var As Integer 'Вызовет ошибку компиляции
    . . .
```

Если объявление переменной в блоке цикла содержит указание начального значения, то переменная инициализируется при каждом проходе цикла:

```
For I = 1 To 10
    Dim Var As Integer = 100
    Console.WriteLine(Var) 'Всегда выводит 100
    Var += 1
Next
```

11. Ввод и вывод данных в консольных приложениях. Примеры программ.

Опишем, как осуществляется ввод и вывод данных в консольных приложениях на VB.NET.

Язык Visual Basic .NET не имеет специальных операторов для ввода и вывода данных. Эти действия реализуются посредством методов некоторых классов. В частности, методы для ввода и вывода данных на консоль содержит класс `Console` из пространства имен `System`. Это методы `Read()` и `ReadLine()` (для ввода данных) и методы `Write()` и `WriteLine()` (для вывода данных). Особенность этих методов является то, что они являются *разделяемыми методами* класса, то есть для их вызова экземпляр класса не создается.

Метод `WriteLine()` служит для вывода данных на консоль и перевода позиции вывода на следующую строку. Данный метод является *перегруженным*, то есть существует несколько версий метода с разной сигнатурой. В частности, имеются версии метода `WriteLine()` для вывода одиночных значений всех примитивных типов. Вызов `WriteLine()` без параметров означает просто переход на новую строку. Данный метод также позволяет осуществлять *форматированный вывод*. Для этого первым параметром метода указывается строка со *спецификаторами формата*, затем следует набор выводимых значений. Спецификаторы формата строятся по следующей схеме. В простейшем случае в фигурных скобках указывается номер того значения из набора, которое подставляется вместо спецификатора в строку (нумерация начинается с нуля). Например, в результате выполнения

```
Console.WriteLine("This is {0}, {2} and {1}", 10, 100, 1000)
```

получим на консоли строку

```
This is 10, 1000 and 100
```

Если после номера значения из набора через запятую указать натуральное число n, то значение выводится в поле из n пробелов, начиная справа. Например,

```
System.Console.WriteLine("This is {0,10}", 33)
```

даст на консоли строку в виде

```
This is         33
```

Более сложный вид спецификатора предполагает указание форматирующего символа (после номера значения через двоеточие). Возможные форматирующие символы сведены в табл. 5.

Таблица 5

Форматирующие символы

Символ	Описание	Параметры `WriteLine()`	Результат
`C` или `c`	Вывод значений в денежном формате (используются региональные установки системы)	`"x={0:C}", 8765.4321`	`x=8 765,43р.`
`D` или `d`	Вывод целых значений. Можно указать количество позиций цифр.	`"x={0:D7}", 98765`	`x=0098765`
`N` или `n`	Вывод числовых значений с разделением разрядов. Можно указать точность (по умолчанию – 2 цифры после запятой)	`"x={0:N3}", 9875.4329`	`x=9 875,433`
`E` или `e`	Использование экспоненциального формата. Можно указать точность (по умолчанию – 6 цифр после запятой)	`"x={0:E3}", 9875.4329`	`x=9,875E+003`
`F` или `f`	Вывод вещественного числа. Можно указать количество цифр после запятой (по умолчанию – 2)	`"x={0:F3}", 9875.4329`	`x=9875,433`
`G` или `g`	Общий формат для вывода чисел	`"x={0:G}", 9875.4329`	`x=9875,4329`
`X` или `x`	Выводит целое значение в шестнадцатеричной системе счисления	`"x={0:X}", 9875`	`x=2693`

Метод `Write()` по функциональности аналогичен методу `WriteLine()`, но не переводит позицию вывода на следующую строку.

Для ввода данных в программу можно использовать метод `ReadLine()`. Данный метод является функцией, которая возвращает строку, введенную с консоли (ввод строки завершается нажатием Enter). Обычно веденная строка преобразуется к необходимому типу при помощи функций приведения:

```
Dim A As Integer, B As Double
A = CInt(System.Console.ReadLine())
B = CDbl(System.Console.ReadLine())
```

Метод `Read()` является функцией, которая возвращает Unicode-код очередного символа строки вода (значение типа `Integer`) или -1, если строка ввода закончилась. Рассмотрим пример использования данного метода:

```
Dim i As Integer
Dim c As Char
While True
    i = System.Console.Read()
    If i = - 1 Then Exit While
    c = Microsoft.VisualBasic.Chr(i)
```

```
        System.Console.WriteLine("Echo: {0}", c)
    End While
```

При вызове, метод `Read()` не возвращает значение до тех пор, пока строка не завершится вводом Enter. Затем возвращаются все символы строки, включая символы перевода строки и возврата каретки[1].

В программах нередко возникает необходимость в использовании математических функций. Пространство имен `System` содержит класс `Math`, разделяемые методы которого представляют набор таких функций. Тип аргументов и возвращаемого значения большинства методов данного класса – `Double`. Наиболее часто используемые методы класса `Math` приведены в таблице 6.

Таблица 6

Методы класса `Math`

Имя метода	Описание
`Abs(x)`	Модуль числа x
`Ceiling(x)`	Округление до ближайшего целого, не меньшего, чем x
`Cos(x)`	Косинус x (x – в радианах)
`Exp(x)`	Экспонента, e^x
`Floor(x)`	Округление до ближайшего целого, не большего, чем x
`Log(x)`	Натуральный логарифм x
`Max(x, y)`	Максимальное из двух чисел[2]
`Min(x, y)`	Минимальное из двух чисел[3]
`Pow(x, y)`	Степень x^y
`Sin(x)`	Синус x (x – в радианах)
`Sqrt(x)`	Квадратный корень из x
`Tan(x)`	Тангенс x (x – в радианах)

Рассмотрим некоторые примеры программ на языке Visual Basic .NET.

1. Получить факториал введенного пользователем числа.

```
Option Strict On
Imports System
Module Factorial
    Sub Main()
        Dim N As Integer
        Dim Res As Integer = 1
        Console.Write("Введите N: ")
      N = CInt(Console.ReadLine())
        Dim I As Integer
        For I = 1 To N
            Res *= I
        Next
        Console.WriteLine("Факториал {0} равен {1}", N, Res)
    End Sub
End Module
```

[1] Таким образом, приведенный в качестве примера фрагмент программы будет выполняться бесконечно.

[2] Имеются перегруженные версии данного метода для аргументов типов `Single`, `Integer` и `Long`.

[3] См. примечание 2.

Дадим комментарии к приведенной программе. Прежде всего, обратите внимание на начало программы. Если в программе устанавливаются какие-либо опции компилятора, то их установка должна производиться в самом начале. Вторая строка программы содержит команду `Imports`. Данная команда предназначена для импортирования пространств имен, что позволяет сократить запись обращения к классу из пространства имен в программе (мы пишем `Console.WriteLine`, а не `System.Console.WriteLine`). Переменная-счетчик цикла объявлена непосредственно перед записью цикла. В цикле использован комбинированный (с операцией `*`) оператор присваивания.

Данная программа считает факториалы чисел до 12 включительно, большие числа вызывают ошибку переполнения. Изменим программу, определив тип переменной `Res` как `Decimal`. Это позволит считать факториалы до числа 27 включительно. Также используем для инициализации переменной `N` ввод пользователя.

```
Option Strict On
Imports System
Module Factorial
    Sub Main()
        Dim Res As Decimal = 1
        Console.Write("Введите N: ")
        Dim N As Integer = CInt(Console.ReadLine())
        Dim I As Integer
        For I = 1 To N
            Res *= I
        Next
        Console.WriteLine("Факториал {0} равен {1}", N, Res)
    End Sub
End Module
```

Попробуйте изменить тип переменной `Res` на `Double`. На сколько большие факториалы (пусть даже приблизительно) сможет посчитать программа?

2. Получить таблицу значений функции $y=\cos x$, используя для вычисления функции метод `Math.Cos()` и ряд Тейлора. Границы и шаг изменения x вводятся пользователем.

```
Imports System
Module CosTable
Const eps As Double = 0.001
    Sub Main()
        Console.Write("Введите левую границу a: ")
        Dim a As Double = CDbl(Console.ReadLine())
        Console.Write("Введите правую границу b: ")
        Dim b As Double = CDbl(Console.ReadLine())
        Console.Write("Введите шаг h: ")
        Dim h As Double = CDbl(Console.ReadLine())
        If h <= 0 Then
             Console.WriteLine("Шаг должен быть больше 0")
             Exit Sub
        End If
        Dim x As Double
```

```
            For x = a To b Step h
                Dim s As Double = 0
                Dim ds As Double = 1
                Dim k As Integer = 1
                Do
                    s += ds
                    ds *= -(x^2)/(2*k*(2*k-1))
                    k += 1
                Loop Until Math.Abs(ds) < eps
      Console.WriteLine("x={0:G},y={1:G},s={2:G}",x,Math.Cos(x),s)
            Next x
        End Sub
    End Module
```

3. Вводятся начальное и конечное значение временного интервала (часы и минуты), а также временной шаг (минуты и секунды). Вывести список отсчетов времени от начального до конечного значений с заданным шагом.

```
    Imports System
    Module TimeSlaps
        Sub Main()
            Console.Write("Введите начальные часы: ")
            Dim HS As Integer = CInt(Console.ReadLine())
            Console.Write("Введите начальные минуты: ")
            Dim MS As Integer = CInt(Console.ReadLine())
            Console.Write("Введите конечные часы: ")
            Dim HF As Integer = CInt(Console.ReadLine())
            Console.Write("Введите конечные минуты: ")
            Dim MF As Integer = CInt(Console.ReadLine())
            Console.Write("Введите минуты шага: ")
            Dim MStep As Integer = CInt(Console.ReadLine())
            Console.Write("Введите секунды шага: ")
            Dim SStep As Integer = CInt(Console.ReadLine())
            Dim A As Integer = (HS*60+MS)*60  'Переводим в секунды
            Dim B As Integer = (HF*60+MF)*60
            Dim S As Integer = MStep*60+SStep
            Dim I As Integer
            For I = A To B Step S
                Dim Hours As Integer = I\3600
                Dim Min As Integer = (I - Hours*3600)\60
                Dim Sec As Integer = I - Hours*3600 - Min*60
           Console.WriteLine("{0:D2}:{1:D2}:{2:D2}",Hours,Min,Sec)
            Next
        End Sub
    End Module
```

4. Вводится месяц и число 2003 года. Проверить корректность исходных данных и вывести, какой это день недели, если 1 января 2003 года была среда.

```
    Imports System
    Module DayOfWeek
        Sub Main()
            Console.Write("Введите месяц: ")
```

```
            Dim M As Integer = CInt(Console.ReadLine())
            If (M > 12) OrElse (M < 1)
               Console.WriteLine("Неверно задан месяц!")
               Exit Sub
            End If
            Dim UBound As Integer
            Select UBound
            Case 1, 3, 5, 7, 8, 10, 12
                UBound = 31
            Case 4, 6, 9, 11
                UBound = 30
            Case Else
                UBound = 28
            End Select
            Console.Write("Введите день месяца: ")
            Dim D As Integer = CInt(Console.ReadLine())
            If (D > UBound) OrElse (D < 1)
               Console.WriteLine("Неверно задан день месяца!")
               Exit Sub
            End If
            Dim NumberOfDays As Integer = D
            Dim I As Integer
            For I = 1 To M - 1
                Select I
                Case 1, 3, 5, 7, 8, 10, 12
                     NumberOfDays += 31
                Case 4, 6, 9, 11
                     NumberOfDays += 30
                Case Else
                     NumberOfDays += 28
                End Select
            Next
            Select NumberOfDays Mod 7
            Case 1
                Console.WriteLine("Среда")
            Case 2
                Console.WriteLine("Четверг")
            Case 3
                Console.WriteLine("Пятница")
            Case 4
                Console.WriteLine("Суббота")
            Case 5
                Console.WriteLine("Воскресенье")
            Case 6
                Console.WriteLine("Понедельник")
            Case 0
                Console.WriteLine("Вторник")
            End Select
        End Sub
    End Module
```

Конечно, данная программа выглядит достаточно громоздко. Более короткое решение задачи можно получить, если использовать массивы. Это решение будет рассмотрено ниже.

5. Вводятся два натуральных числа A и B. Вывести все натуральные числа из отрезка [A,B], в двоичном разложении которых ровно 4 нуля.

```
Imports System
Module FourNulls
    Sub Main()
        Console.Write("Введите нижнюю границу интервала: ")
        Dim A As Integer = CInt(Console.ReadLine())
        Console.Write("Введите верхнюю границу интервала: ")
        Dim B As Integer = CInt(Console.ReadLine())
        If (B < A) OrElse (A < 1)
           Console.WriteLine("Неверно заданы исходные данные")
           Exit Sub
        End If
        Dim I As Integer
        For I = A To B
            Dim NumOfNulls As Integer = 0
            Dim C As Integer = I
            Do While C > 1
                If C Mod 2 = 0 Then NumOfNulls += 1
                C \= 2
            Loop
            If NumOfNulls = 4 Then Console.WriteLine(I)
        Next
    End Sub
End Module
```

12. Массивы в Visual Basic .NET.

Рассмотрим начальные сведения о работе с массивами в языке Visual Basic .NET. Опишем синтаксис объявления массивов. Объявление массива схоже с объявлением переменной, однако после идентификатора массива располагается пара круглых скобок, в которой указывается верхняя граница индекса массива:

```
Dim Data(9) As Integer
Dim Coeff(1000) As Double
```

Обратите внимание: указывается верхняя граница массива, а не число элементов. В VB.NET нижняя граница массива всегда равна нулю[1].

Для доступа к элементу массива используются круглые скобки:

```
Data(5) = 10 'Работаем с шестым элементом
Coeff(i) = 2*Coeff(i-1)
```

Если необходимо объявить массив из нескольких размерностей, верхние границы индексов каждой перечисляются в объявлении через запятую:

```
Dim Matrix(100,100) As Integer
```

[1] Границы индексов массива можно изменить, но особыми методами.

```
Matrix(50,50) = 100000
```

Все массивы в VB.NET являются динамическими. Во время работы программы их можно переобъявлять с новыми размерами. Для этого служит команда `ReDim`:

```
Dim Data(9) As Integer
ReDim Data(99)          'Теперь в массиве 100 элементов
```

Команда `ReDim` переобъявляет массив, уничтожая его содержимое. Вариант команды `ReDim Preserve` изменяет размер массива, сохраняя содержимое. Тип массива и количество размерностей командой `ReDim` изменить нельзя.

Массив может быть объявлен и без указания индекса:

```
Dim Data() As Integer
Dim Matrix(,) As Integer     'Двумерный массив
```

В этом случае до первого обращения к элементу массива должна быть выполнена команда `ReDim`. Также в этом случае можно использовать синтаксис объявления массива, при котором круглые скобки располагаются после идентификатора типа:

```
Dim Data As Integer()
Dim Matrix As Integer(,)
```

Массивы могут инициализироваться при объявлении. Для этого их элементы перечисляются в фигурных скобках через запятую. Если объявлен массив без указания индекса, и он инициализирован, то работать с ним можно без предварительного вызова команды `ReDim`:

```
Dim Data() As Integer = {1,3,5,7,9}
Dim Matrix(,) As Integer = {{0,2,4},{10,20,40}}
```

Обратите внимание на синтаксис инициализации двумерного массива. Если для таких массивов не указаны верхние границы индексов, то нужно задавать одинаковое значение элементов по размерностям.

Для инициализации массива можно использовать вызов *конструктора массива*, так как любой массив ведет себя сходно с произвольной ссылочной переменной (подробнее о конструкторах будет рассказано ниже):

```
Dim A() As Integer
A = New Integer(){10, 20, 40}
```

Можно задать верхнюю границу массива, но она должна соответствовать количеству элементов в списке инициализации:

```
Dim A() As Integer
A = New Integer(2){10, 20, 40}
```

В VB.NET существует возможность работать с так называемыми *jagged-массивами*, то есть массивами непрямоугольной формы. Для этого используется следующий синтаксис:

```
'Объявляем массив размером 2 с элементами-массивами
Dim A(1)() As Integer
'Инициализируем первый элемент-массив
```

```
A(0) = New Integer(){10, 20, 40}
'Инициализируем второй элемент-массив
A(1) = New Integer(){100, 200}
```

Приведем пример объявления трехмерного jagged-массива[1]:

```
Dim B()()() As Integer
B = New Integer()()(){ _
New Integer()(){New Integer(){1,2},New Integer(){10,20,30}}, _
New Integer()(){New Integer(){100,200}, _
                                        New Integer(){1,2,3,4}}}
```

Команда `Erase` позволяет освободить память, выделенную под массив при завершении работы с ним:

```
Erase Data 'Указывается только имя массива, без скобок
```

Для массивов, у которых базовый тип является примитивным, возможно использовать специальный оператор цикла, перебирающий все элементы. Это оператор `For Each`, его синтаксис:

```
For Each <переменная> In <имя массива>
    [<команды>]
Next [<переменная>]
```

В следующем фрагменте программы производится суммирование элементов массива:

```
Dim Data() As Integer = {1,3,5,7,9}
Dim Sum, Element As Integer
For Each Element In Data
    Sum += Element
Next
```

В операторе `For Each` возможно перемещение только в одном направлении, при этом попытки присвоить значение элементу массива игнорируются.

```
Dim Data() As Integer = {1,3,5,7,9}
Dim Element As Integer
For Each Element In Data
    'Компилируется, но элементы Data сохранят значение
    Element = 100
Next
```

Одним из нюансов использования массивов является то, что переменные типа массив фактически являются указателями. Следовательно, даже при передаче массивов в подпрограммы по значению, можно изменить содержимое массива.

Рассмотрим программу, которая позволяет ввести и отсортировать массив целых чисел.

```
Imports System
Module ArraySort
```

[1] Обратите внимание: объявление `Dim M(,) As Integer` задает двумерный массив чисел; `Dim M()() As Integer` описывает одномерный массив массивов.

```
    Sub Main()
        Dim M() As Integer
        Console.Write("Введите размер массива: ")
        Dim Size As Integer = CInt(Console.ReadLine())
        ReDim M(Size-1)
        Dim I As Integer
        For I = 0 To Size-1
            Console.Write("Введите {0}-й элемент массива: ",I)
            M(I) = CInt(Console.ReadLine())
        Next I
        Console.WriteLine("Исходный массив:")
        For Each I In M
            Console.Write("{0,6}", I)
        Next I
        Console.WriteLine()
        Dim J,Dop As Integer
        For I = 0 To Size-2
            For J = I+1 To Size-1
                If M(I) > M(J) 'Сортируем по возрастанию
                    Dop = M(I)
                    M(I) = M(J)
                    M(J) = Dop
                End If
        Next J,I
        Console.WriteLine("Отсортированный массив:")
        For Each I In M
            Console.Write("{0,6}", I)
        Next I
    End Sub
End Module
```

Решим задачу из предыдущего параграфа при помощи массивов. Напомним ее условие. Вводится месяц и число 2003 года. Проверить корректность исходных данных и вывести, какой это день недели, если 1 января 2003 года была среда.

```
Imports System
Module DayOfWeek
    Sub Main()
        Dim MaxDays() As Integer = _
              {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
        Dim Names() As String = {"Вторник", "Среда", _
                        "Четверг", "Пятница", "Суббота", _
                        "Воскресенье", "Понедельник"}
        Console.Write("Введите месяц: ")
        Dim M As Integer = CInt(Console.ReadLine())
        If (M > 12) OrElse (M < 1)
           Console.WriteLine("Неверно задан месяц!")
           Exit Sub
        End If
        Console.Write("Введите день месяца: ")
        Dim D As Integer = CInt(Console.ReadLine())
        If (D > MaxDays(M-1)) OrElse (D < 1)
```

```
            Console.WriteLine("Неверно задан день месяца!")
            Exit Sub
        End If
        Dim NumberOfDays As Integer = D
        Dim I As Integer
        For I = 1 To M - 1
            NumberOfDays += MaxDays(I-1)
        Next
        Console.WriteLine(NumberOfDays)
        Console.WriteLine(Names(NumberOfDays Mod 7))
    End Sub
End Module
```

13. Объявление и вызов методов.

Язык VB.NET содержит два вида организации методов: представление методов в виде процедур и в виде функций.

Рассмотрим «облегченный» синтаксис описания метода-процедуры (не используются некоторые модификаторы, в частности, модификаторы доступа):

```
Sub <имя процедуры>([<список аргументов>])
    [<команды>]
End Sub
```

В процедуре можно использовать команду `Exit Sub` для выхода из процедуры.

Опишем структуру списка аргументов метода-процедуры. Прежде всего, заметим, что даже если список аргументов является пустым, то после имени метода при описании и при вызове желательно наличие пары скобок `()`. Элементы непустого списка аргументов разделяются запятыми и имеют следующий синтаксис:

```
[Optional] [ByVal|ByRef] [ParamArray] <имя аргумента>[()] _
                        [As <тип аргумента>] [= <нач. значение>]
```

`Optional` – ключевое слово, указывает, что параметр является необязательным. При использовании этого ключевого слова все последующие элементы списка параметров также должны быть необязательными и описанными с `Optional`. Для всех необязательных параметров необходимо указать начальное значение, принимаемое ими по умолчанию (в случае отсутствия при вызове). Если используется ключевое слово `ParamArray`, применение `Optional` не допускается.

`ByVal` – ключевое слово, указывает, что параметр передается в метод по значению. Передача по значению используется по умолчанию.

`ByRef` – ключевое слово, указывает, что параметр передается по ссылке.

`ParamArray` – может использоваться только для последнего элемента в списке параметров. Это ключевое слово позволяет задать при вызове произвольное количество параметров. Такие параметры всегда передаются в метод по значению, использование вместе с `ParamArray` ключевых слов `ByRef` и `Optional` не допускается. При использовании `ParamArray` необходим синтак-

сис описания параметров-массивов. Работа в подпрограмме с параметрами, описанными через ParamArray, происходит как работа с элементами массива.

<имя аргумента> – любой допустимый идентификатор. Синтаксис <имя аргумента>() используется при описании параметров-массивов.

<тип аргумента> – тип аргумента. Допустимо указание любого примитивного и пользовательского типа, за исключением модуля. При включенной опции Option Strict On тип аргумента указывать обязательно, иначе аргумент получает тип Object.

<нач. значение> – литерал или выражение, дающее константу. Используется только вместе с параметром Optional. Если указан тип Object, единственное значение по умолчанию – это значение Nothing.

Рассмотрим примеры описания заголовков методов-процедур:

```
Sub Proc1()                 'Процедура без параметров
Sub Proc2(A As Integer)     'Один параметр, передача по значению
Sub Proc3(A As Char, B As Char)    'Два параметра
Sub Proc4(ByRef A As Char, Optional ByVal B As Byte = 2)
   'Первый параметр передается по ссылке, второй – по значению
Sub Proc5(ParamArray A() As Integer)
   'Произвольное количество параметров целого типа
```

Одной из распространенных ошибок при описании заголовков свойств является попытка объявить несколько параметров одного типа «скопом»:

```
Sub SomeProc(A, B As Integer)
```

В данной ситуации *на самом деле* происходит объявление первого параметра с типом Object, а только второго параметра с типом Integer. Если же включена опция компилятора Option Strict On, то приведенный пример вообще не скомпилируется. Правильным является следующее объявление:

```
Sub SomeProc(A As Integer, B As Integer)
```

Заметим, что в VB.NET не допускается вложенность методов – объявить один метод внутри другого нельзя.

Для вызова метода-процедуры можно использовать либо имя метода, либо оператор Call в форме Call <имя метода-процедуры>. В любом случае указываются фактические параметры, подставляемые в процедуру на место формальных аргументов:

```
Proc2(100) 'Параметр A равен 100
Call Proc3("H"C, "i"C)
Proc4(A) 'Второй параметр по умолчанию равен 2
Proc5(1,2,3,4) 'Произвольное количество параметров
```

Отметим одну особенность. Если в Object Pascal на месте фактических параметров, передаваемых по ссылке, могли находиться только переменные, то в VB.NET допустимо использовать как переменную, так и литерал. Передача по ссылке в этом случае фактически означает создание невидимой временной переменной-«обертки» для параметра-литерала. Естественно, в этом случае измененное в методе значение аргумента в вызывающую программу не попадет.

Кроме использовавшегося в примерах способа передачи параметров, называемого *передача по позиции*, возможен способ передачи параметров *по имени*. В этом случае при вызове подпрограммы в списке параметров указывается имя формального параметра, затем двоеточие и знак равенства (:=), затем фактическое значение параметра. Использование передачи по имени позволяет вводить фактические параметры в любом порядке. При передаче параметров можно также смешивать два подхода. Рассмотрим пример:

```
Proc3("H"C, "i"C) 'Передача по позиции
Proc3(B := "H"C, A := "i"C)'Передача по имени, порядок изменен
Proc3("H"C, B := "i"C) 'Смешанный способ передачи
```

Синтаксис описания метода-функции сходен с синтаксисом описания метода-процедуры:

```
Function <имя>([<список аргументов>]) As <тип возвр. значения>
    [<команды>]
End Function
```

Для списка аргументов функции справедливо все то, что было отмечено для списка аргументов процедуры. Команда `Exit Function` вызывает немедленный выход из функции.

Для возврата значения функции в теле функции возможно применение двух подходов. Первый заключается в использовании имени функции в левой части оператора присваивания, а возвращаемого значения – в правой. После такого оператора присваивания выхода из функции не происходит, ее выполнение продолжается. Второй подход – использование оператора `Return` в виде `Return <возвращаемое значение>`. При этом, в отличие от первого подхода, происходит немедленный выход из функции.

Тип возвращаемого значения функции может быть любым[1], за исключением модуля. Если произошел выход из функции до определения возвращаемого значения, то функция принимает значение типа по умолчанию.

При вызове функция обычно фигурирует в правой части оператора присваивания или в выражении, где используется возвращаемое ей значение. Однако это значение можно проигнорировать. Для этого достаточно использовать оператор `Call` или просто записать вызов функции как вызов процедуры.

Одной из возможностей VB.NET является описание *перегруженных* методов. Перегруженные методы называются одинаково, но выполняют различные действия. Для того чтобы различить перегруженные методы при вызове, они должны иметь разные сигнатуры. При этом если параметры метода различаются только способом передачи (`ByVal|ByRef`), то считается, что их сигнатуры совпадают.

Для того чтобы подчеркнуть, что какой-то метод является перегруженным, возможно при описании метода указать ключевое слово `Overloads` (перед `Sub` или `Function`):

```
Overloads Sub Proc(A As Integer)
```

[1] `Function F() As Integer()` – такая функция может вернуть любой одномерный массив целых чисел.

```
. . .
Overloads Sub Proc(A As Double)
```

Однако если перегружаемые методы находятся в одном пользовательском типе, то ключевое слово `Overloads` можно не указывать.

В заключение рассмотрим пример программы для вычисления чисел Фибоначчи.

```
Option Strict On
Imports System
Module FibonachiCalculus
    Dim N As Integer
    Function Fibonachi(N As Integer) As Integer ‘Метод-функция
        If (N = 1) Or (N = 2)
            Return 1
        Else
            Return Fibonachi(N-1) + Fibonachi(N-2) ‘Рекурсия
        End If
    End Function
    Sub Main()
        Console.Write("Введите N ")
        N = CInt(Console.ReadLine())
        Console.Write("{0}-е число = {1}", N, Fibonachi(N))
    End Sub
End Module
```

14. Синтаксис объявления классов.

Как упоминалось ранее, VB.NET является языком, полностью поддерживающим концепции ООП. Основным объектным типом является *класс*. Синтаксис объявления класса следующий:

```
Class <имя класса>
    [<члены класса>]
End Class
```

Здесь `<имя класса>` – любой уникальный идентификатор. Допустимы следующие члены класса.

1. Поле. Поля класса описываются как обычные переменные, как правило, с указанием модификатора доступа. Поле может иметь любой тип, за исключением модуля.

```
Class CSomeClass
. . .
    Dim Field1 As Integer
    Private Field2 As Integer
    Public Field3 As String
. . .
End Class
```

Если для поля не указан модификатор доступа, то по умолчанию подразумевается модификатор `Private`. Полям класса можно придавать начальные значения.

2. Константа. Объявление константы, помещенное в класс, обычно используется для того, чтобы сделать код более читабельным. Модификатор доступа к константам по умолчанию - `Private`. Если объявлена открытая (`Public` или `Friend)` 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 по умолчанию для методов является `Friend`.

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

5. Конструктор. Задача конструктора – начальная инициализация объекта или класса. Конструкторы также будут описаны далее.

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

7. Вложенный пользовательский тип. Описание класса может содержать описание другого пользовательского типа – класса, структуры, интерфейса. Обычно вложенные типы выполняют дополнительные функции для основного типа и явно вне основного типа не используются.

При описании класса допустимо указать для него следующие модификаторы доступа – `Public` или `Friend` (применяется по умолчанию). Если класс является элементом другого пользовательского типа, то его можно объявить с модификаторами `Private` или `Protected`. При этом если класс объявлен с модификатором `Friend`, то даже члены класса, которые объявлены с модификатором `Public`, не будут видны за пределами сборки.

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

```
Dim <имя объекта> As <имя класса>
```

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

```
<имя объекта> = New <имя класса>()
```

Инициализацию объекта можно совместить с объявлением объекта, указав оператор `New` и конструктор после ключевого слова `As`:

```
Dim <имя объекта> As New <имя класса>()
```

Доступ к членам класса через объект осуществляется по синтаксису `<имя объекта>.<имя члена>`.

Приведем пример описания класса. Класс будет содержать два поля:

```
Class CPet
    Public Age As Integer
    Public Name As String
End Class
```

Проиллюстрируем описание и использование объектов класса `CPet`:

```
Dim Dog As CPet             'Просто объявление
Dim Cat As New CPet()       'Объявление с инициализацией
Dog = New CPet()            'Инициализация объекта
Dog.Age = 10                'Доступ к полям
Cat.Name = "123Y"
```

Добавим в класс CPet методы. Заметим, что для устранения конфликта имен «имя члена класса = имя параметра метода» возможно использование ключевого слова Me, представляющего собой ссылку на текущий объект класса.

```
Class CPet
    Public Age As Integer
    Public Name As String
    Sub SetAge(Age As Integer)
        Me.Age = Age
    End Sub
    Function GetName() As String
        Return Name
    End Function
End Class
```

Чтобы упростить работу с членами объекта некоторого класса в программе, можно использовать специальный оператор With. Синтаксис данного оператора следующий:

```
With <объект>
    [<команды>]
End With
```

Для доступа к членам объекта внутри блока With используется синтаксис .<имя члена> (точку указывать обязательно):

```
With Dog
    .SetAge(10)
    S = .GetName()
End With
```

Разрешена вложенность блоков With, если это не вызывает многозначности ссылок на члены объектов. Крайне не рекомендуется (хотя и возможно) организовывать программный код так, чтобы осуществлялся принудительный выход из блока With или переход на операторы внутри блока извне.

Рассмотрим следующий пример. Опишем класс для представления стека, состоящего из произвольных объектов.

```
Option Strict On
Imports System
Class CStack
     Private Data() As Object
     Private CurrentIndex As Integer = -1
     Sub Push(X As Object)
         CurrentIndex += 1
         ReDim Preserve Data(CurrentIndex)
         Data(CurrentIndex) = X
     End Sub
```

```
        Function Pop() As Object
            If Not IsEmpty() Then
                Pop = Data(CurrentIndex)
                CurrentIndex -= 1
                ReDim Preserve Data(CurrentIndex)
            End If
        End Function
        Function IsEmpty() As Boolean
            Return CurrentIndex = -1
        End Function
    End Class

    Module StackUseExample
        Sub Main()
            Dim Stack As New CStack()
            Do
                Console.Write("Введите элемент (-1 - окончание):")
                Dim A As Integer = CInt(Console.ReadLine())
                If A = - 1 Then Exit Do
                Stack.Push(A)
            Loop
            Console.WriteLine("Данные стека")
            With Stack
                Do Until .IsEmpty()
                    Console.WriteLine(.Pop())
                Loop
            End With
        End Sub
    End Module
```

Обратите внимание на размещение класса и модуля в исходном коде. В данном примере можно использовать вариант размещения класса в модуле:

```
Option Strict On
Imports System
Module StackUseExample
    Class CStack
        . . .
    End Class
    Sub Main()
        . . .
End Module
```

В таком случае один пользовательский тип – модуль – включает описание другого пользовательского типа – класса. Размещать описание пользовательских типов внутри методов (в частности, внутри метода `Main()`) не разрешается.

15. Свойства.

Свойства класса призваны предоставить защищенный доступ к полям. Как и в большинстве объектно-ориентированных языков, в VB .NET непосредственная работа с полями не приветствуется. Поля класса обычно объявляются с

модификатором `Private`, а для доступа к ним используются свойства. Свойства позволяют при обращении к полю выполнить некоторые действия (проверку значений, форматирование данных и тому подобное). Рассмотрим синтаксис описания свойства:

```
Property <имя свойства>[()] As <тип свойства>
    Get
        [<команды>]
        [Return <возвращаемое значение>]
        [<команды>]
    End Get
    Set (ByVal <значение> As <тип>)
        [<команды>]
    End Set
End Property
```

Тип свойства обычно совпадает с типом того поля, для обслуживания которого создается свойство. Если не указать тип свойства, подразумевается тип `Object`. Часть `Get` – `End Get` отвечает за возвращаемое свойством значение и работает как функция, часть `Set` – `End Set` работает как процедура, устанавливающая значение свойства[1]. Команда `Exit Property` служит для выхода из подпрограмм, описывающих свойство.

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

```
Class CPet
    Private fAge As Integer 'Теперь поля закрыты
    Private fName As String
    Property Age As Integer
        Get
            Return fAge
        End Get
        Set (ByVal Value As Integer)
            If Value > 0 Then fAge = 0 Else fAge = Value
        End Set
    End Property
    Property Name As String
        Get
            Return "My name is " & fName
        End Get
        Set (ByVal Value As String)
            fName = Value
        End Set
    End Property
End Class
```

[1] Если в качестве параметра этой процедуры будет использоваться переменная с именем `Value`, то все, что написано в круглых скобках после `Set`, можно опустить:

```
Property Age As Integer
    . . .
    Set
        If Value > 0 Then fAge = 0 Else fAge = Value
    End Set
End Property
```

Модификатором доступа для свойства по умолчанию служит `Friend`.

Можно объявить свойство только для чтения или только для записи. Для этого используются модификаторы `ReadOnly` (свойство только для чтения) и `WriteOnly` (свойство только для записи). У таких свойств отсутствует секция `Set - End Set` или `Get - End Get` в зависимости от модификатора. Модификаторы `ReadOnly` и `WriteOnly` указываются сразу после модификаторов доступа к свойству.

Свойство может зависеть от индекса, исполняя роль массива свойств. Обычно такое свойство обслуживает поле-массив класса. Для определения *свойства-массива* необходимо после имени свойства указать в круглых скобках список индексов. Этот список напоминает список аргументов подпрограммы, однако, каждый индекс должен задаваться с модификатором `ByVal`[1].

В качестве примера объявления свойства-массива рассмотрим следующий класс:

```
Class SmartArray
    Private fData(10) As Integer 'Это объявление поля-массива
    Property Data(ByVal Index As Integer) As Integer
        Get
          If Index < 11 Then Return fData(Index) Else Return 0
        End Get
        Set (ByVal Value As Integer)
            If Index < 11 Then fData(Index)=Value
        End Set
    End Property
End Class
```

Свойство-массив (и только его) можно сделать *свойством по умолчанию*. Тогда при обращении к свойству его имя можно опустить. Для задания свойства по умолчанию используется ключевое слово `Default`, размещаемое перед модификатором доступа к свойству. Изменим свойство `Data`, сделав его свойством по умолчанию:

```
Class SmartArray
    Private fData(10) As Integer  'Это объявление поля-массива
    Default Property Data(ByVal Index As Integer) As Integer
. . .
End Class
. . .
Dim SA As New SmartArray()
SA.Data(1) = 1000   'Используем индексированное свойство Data
SA(2) = 2000        'Вспоминаем, что оно свойство по умолчанию
```

Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются два метода[2] со специальными именами `Get_`*`Name`* и `Set_`*`Name`*, где *`Name`* – это имя свойства. Побочным эффектом данного преобразования является тот факт, что методы с данными именами не допус-

[1] Напомним, что данный модификатор применяется по умолчанию.

[2] Или один метод, если свойство объявлено только для чтения или только для записи.

тимы в классе (даже если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству).

```
Class CPet
    . . .
    Property Age As Integer
        Get
            Return fAge
        End Get
        Set (ByVal Value As Integer)
            If Value > 0 Then fAge = 0 Else fAge = Value
        End Set
    End Property
    Sub Set_Age(ByVal Value As Integer)    'Ошибка компиляции!
    . . .
```

Подстановка методов вместо свойств в скомпилированном коде означает, в частности, что свойства можно перегружать подобно методам. Перегрузку нельзя осуществить для свойств, которые отличаются только типом возвращаемого значения, но ее можно использовать для того, чтобы описать в классе несколько одноименных свойств-массивов с разными типами индексов (или с разным количеством индексов). Рассмотрим пример перегрузки свойств:

```
Class CPropertyOverloadExample
    WriteOnly Property Item As Integer
        Set
            Console.WriteLine(Value)
        End Set
    End Property
    WriteOnly Property Item(X As Integer) As Integer
        Set
            Console.WriteLine("X={0}, Value={1}", X, Value)
        End Set
    End Property
    WriteOnly Property Item(X As Integer,Y As Integer) _
                                                    As Integer
        Set
            Console.WriteLine("X={0}, Y={1}, Value={2}", _
                                                X, Y, Value)
        End Set
    End Property
    WriteOnly Property Item(I As Double) As Integer
       Set
           Console.WriteLine("I={0}, Value={1}", I, Value)
       End Set
    End Property
End Class
```

В случае перегружаемых свойств нельзя использовать директиву `Default` ни с одним из них.

16. Объявление и использование конструкторов.

Конструкторы используются для начальной инициализации объектов пользовательского типа. Конструкторы похожи на методы, но в отличие от методов конструкторы не наследуются и вызов конструктора в виде `<имя объекта>.<имя конструктора>` после инициализации объекта запрещен. Различают несколько видов конструкторов – *конструкторы по умолчанию*, *пользовательские конструкторы*, *разделяемые конструкторы*.

Конструктор по умолчанию автоматически создается компилятором, если пользователь 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. Если начальное значение для поля не задано, поле гарантированно принимает свое значение по умолчанию («обнуляется»).

```
Class CPet                          'Класс не содержит конструктора
    Public Age As Integer
    Public Name As String = "I'm a pet"
End Class
. . .
Dim Pet As CPet
Pet = New CPet()                    'Вызов конструктора по умолчанию
Console.WriteLine(Pet.Age)          'Выводит 0
Console.WriteLine(Pet.Name)         'Выводит I'm a pet
```

Пользовательский конструктор описывается в классе как метод-процедура с именем `New`. Пользовательский конструктор может получать параметры, необходимые для инициализации объекта. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой. При определении нескольких пользовательских конструкторов не используется ключевое слово `Overloads`.

Опишем два пользовательских конструктора в классе `CPet`:

```
Class CPet
    Public Age As Integer
    Public Name As String
    Public Sub New()
        Age = 0
        Name = "CPet"
    End Sub
    Public Sub New(ByVal X As Integer, ByVal Y As String)
        Age = X
        Name = Y
    End Sub
End Class
```

При вызове пользовательского конструктора в момент инициализации объекта фактические параметры конструктора указываются после имени класса в скобках. Если конструкторов у класса несколько, подходящий выбирается по сигнатуре:

```
Dim Cat As New CPet()           'Это наш первый конструктор
```

```
Dim Dog As New CPet(5,"Buddy") 'Это второй конструктор
```

Пользовательские конструкторы класса могут использоваться для начальной инициализации `ReadOnly` - полей класса. Напомним, что такие поля ведут себя как константы, однако в отличие от констант могут иметь произвольный тип. Таким образом, `ReadOnly` - поля доступны для записи, но только в конструкторе класса.

Обратите внимание: если в классе определен хотя бы один пользовательский конструктор, конструктор по умолчанию не создается. Если мы изменим предыдущий класс, оставив там только один пользовательский конструктор

```
Class CPet
    Public Age As Integer
    Public Name As String
    Public Sub New(ByVal X As Integer, ByVal Y As String)
        Age = X
        Name = Y
    End Sub
End Class
```

то строка `Dim Cat As New CPet()` будет вызывать ошибку компиляции. Код, который выполняет присваивание полям значений по умолчанию, добавляется компилятором автоматически в начало кода любого пользовательского конструктора.

Для доступа к другим конструкторам класса в теле конструктора необходимо использовать имя `New` с необходимыми параметрами. Вызов других конструкторов должен предшествовать всем остальным действиям в теле конструктора.

Разделяемые конструкторы используются для начальной инициализации разделяемых полей класса (см. ниже). Разделяемый конструктор объявляется при помощи ключевого слова `Shared`. Разделяемый конструктор – это всегда конструктор без параметров. Область видимости у разделяемых конструкторов не указывается.

```
Class CAccount
    Public Shared Tax As Double
    Shared Sub New()
        Tax = 0.1
    End Sub
End Class
```

Разделяемые конструкторы вызываются исполняемой средой в следующих случаях

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

В теле разделяемого конструктора возможна работа только с разделяемыми полями класса, разделяемые конструкторы не могут вызывать другие конструкторы класса.

Рассмотрим некоторые аспекты использования конструкторов. Частным случаем пользовательских конструкторов являются *закрытые конструкторы*, то есть конструкторы, объявленные с модификатором видимости `Private`. Если класс содержит объявление закрытого конструктора (и больше никаких конструкторов), то это означает, что объекты данного класса создать невозможно. Подобные классы обычно представляют собой сгруппированный набор разделяемых методов, что и не предполагает создание объекта (примером могут служить классы `System.Math` и `System.Console`).

Классы являются ссылочными типами. Иногда для копирования информации, содержащейся в одном объекте, в другой объект целесообразно создать в классе *копирующий конструктор*. Такой конструктор получает в качестве параметра объект своего класса и производит копирование полей этого объекта в создаваемый объект. Рассмотрим пример:

```
Class Cloneable
    Public Info As String
    Public Info2 As Integer
    Sub New(X As String, Y As Integer)
        Info = X
        Info2 = Y
    End Sub
    Sub New(Sourse As Cloneable)
        Info = Sourse.Info
        Info2 = Sourse.Info2
    End Sub
    Sub Print()
        Console.WriteLine("Info={0}, Info2={1}", Info, Info2)
    End Sub
End Class
```

Пусть имеется программа, использующая данный класс:

```
Module MainModule
    Sub Main()
        Dim A,B As Cloneable
        A = New Cloneable("First",1)
        B = New Cloneable(A)
        A.Print()
        B.Print()
    End Sub
End Module
```

Эта программа выведет на экран две одинаковые строки, то есть объект `A` является полной копией объекта `B`. Отметим, что в Visual Basic .NET предусмотрены и другие способы копировать содержимое объектов, эти способы будут рассмотрены ниже.

17. Разделяемые члены класса.

Разделяемые (shared) поля, методы и свойства предназначены для работы с классом, а не с объектом класса. Соответственно, для вызова разделяемого члена класса обычно используется имя класса (хотя возможно использование и

имени объекта). Разделяемые поля хранят информацию, общую для всех объектов, разделяемые методы не могут работать с полями объекта, а только с разделяемыми полями класса. Иногда разделяемые члены класса называют *статическими*.

Для того чтобы объявить разделяемый член класса, используется ключевое слово `Shared`, указываемое после модификатора доступа (для свойств – после модификатора доступа и модификаторов `ReadOnly` и `WriteOnly`).

Приведем пример класса с разделяемыми членами:

```
Class Account
    Public Shared Tax As Double = 0.1
    Shared Function GetTax() As Double
        Return Tax * 100
    End Function
End Class
. . .
Dim A1 As New Account()
Dim A2 As New Account()
Console.WriteLine(A1.GetTax())    'Используем для вызова объект
Console.WriteLine(Account.GetTax())  'Используем класс
A1.Tax = 0.3                    'Изменили в классе, не в объекте!
Console.WriteLine(A2.GetTax())       'Выводит 0.3
```

При конфликте вида «имя разделяемого поля – имя параметра метода» в разделяемых методах требуется использовать имя класса для ссылки на поле класса.

```
Class Account
    Public Shared Tax As Double = 0.1
    . . .
    Shared Sub SetTax(Tax As Double)
        Account.Tax = Tax
    End Sub
End Class
```

Можно утверждать, что такие члены класса как константа или вложенный пользовательский тип ведут себя как разделяемые, так как для доступа к ним нет нужды создавать объект класса (используется имя класса).

Использование разделяемых методов позволяет оформить исходную программу на VB.NET в виде класса. Для этого точка входа в программу – метод `Main()` – объявляется в классе как разделяемый:

```
Class Example
    Shared Sub Main()
        System.Console.WriteLine("Hello!")
    End Sub
End Class
```

В качестве одно из примеров использования разделяемых методов и закрытых конструкторов опишем класс `CSingleton`. Особенностью этого класса является то, что в приложении можно создать только один объект данного класса.

```
Class CSingleton
    Private Shared Instance As CSingleton
    Public Info As String
    Private Sub New()
    End Sub
    Shared Function Create() As CSingleton
       If Instance Is Nothing Then Instance = New CSingleton()
       Return Instance
    End Function
End Class
```

Так как в классе `CSingleton` конструктор объявлен как закрытый, то единственный способ создать объект этого класса – вызвать функцию `Create()`. Логика работы этой функции организована так, что она всегда возвращает ссылку на один и тот же объект. Обратите внимание: поле `Instance` хранит ссылку на объект и описано как разделяемое. Это сделано для того, чтобы с ним можно было работать в разделяемом методе `Create()`. Так как это поле описано как `Private`, то только в этом методе с ним и можно работать. Пример программы, использующей класс `CSingleton`:

```
Module Example
    Sub Main()
        Dim A As CSingleton = CSingleton.Create()
        Dim B As CSingleton = CSingleton.Create()
        A.Info = "Information"
        System.Console.WriteLine(B.Info)
    End Sub
End Module
```

Объекты `A` и `B` представляют собой одну сущность, то есть на консоль выведется строка `"Information"`.

Класс `CSingleton` соответствует так называемому *шаблону* (*паттерну*) *проектирования* `Singleton`. Шаблоны проектирования призваны упростить процесс создания сложных программных проектов.

18. Наследование классов.

Язык VB.NET полностью поддерживает объектно-ориентированную концепцию наследования. Чтобы показать, что один класс является наследником другого, используется ключевое слово `Inherits` в следующем синтаксисе:

```
Class <имя наследника>
    Inherits <имя базового класса>
```

Наследник обладает всеми полями, методами и свойствами предка, однако члены предка с модификатором `Private` не доступны в наследнике. Конструкторы предка не переносятся в наследник.

При наследовании классов нельзя расширить область видимости класса: `Friend` – класс может наследоваться от `Public` – класса, но не наоборот.

Для обращения к членам непосредственного предка класс-наследник может использовать ключевое слово `MyBase` в формате `MyBase.<имя члена ба-`

зового класса>. Обычно так поступают в конструкторах для вызова конструктора базового класса:

```
Class Point
    Public x, y As Double
    Public Sub New()
        x = 1
        y = 1
    End Sub
End Class
Class Point3D
    Inherits Point
    Public z As Double    'Поля x и y пришли из базового класса
    Public Sub New()
        MyBase.New()      'Инициализируем x и y
        z = 1
    End Sub
End Class
```

Для конструкторов производного класса справедливо следующее замечание: первая строка в теле конструктора должна совершать вызов другого конструктора в виде `Me.New(...)` или `MyClass.New(...)` или `MyBase.New(...)`. Если строка вызова конструктора опущена, компилятор автоматически подставляет в тело конструктора вызов `MyBase.New()`. Если в базовом классе нет конструктора без параметров, происходит ошибка компиляции.

В VB.NET существует базовый класс, от которого наследуются все остальные. Это класс `System.Object`. Наследование от двух и более классов в VB.NET запрещено.

Для классов можно указать два модификатора, связанных с наследованием. Модификатор `NotInheritable` задает класс, который наследовать нельзя. Модификатор `MustInherit` задает абстрактный класс, у которого обязательно должны быть наследники. Объект абстрактного класса создать нельзя, хотя статические члены класса можно вызывать, используя имя класса. Модификаторы наследования указываются непосредственно перед ключевым словом `Class`:

```
Public NotInheritable Class FinishedClass
. . .
MustInherit Class AbstractClass
```

19. Замещение методов. Реализация полиморфизма.

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

```
Class CPet
    Public Sub Speak()
        Console.WriteLine("I'm a pet")
    End Sub
End Class
Class CDog
```

```
    Inherits CPet
    Public Sub Speak()
        Console.WriteLine("I'm a dog")
    End Sub
End Class
. . .
Dim Pet As New CPet(), Dog As New CDog()
. . .
Pet.Speak()
Dog.Speak()
```

При компиляции данного фрагмента будет получено предупреждающее сообщение о том, что метод `CDog.Speak()` конфликтует с методом базового класса `CPet.Speak()`. Чтобы подчеркнуть, что метод класса-наследника замещает метод базового класса, используется ключевое слово `Shadows`:

```
Class CDog
    Inherits CPet
    Public Shadows Sub Speak() 'Компиляция без предупреждений
        Console.WriteLine("I'm a dog")
    End Sub
End Class
```

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

Замещение методов не является полиморфным по умолчанию. Следующий фрагмент кода печатает две одинаковые строки (отметим попутно, что для объектов в VB .NET действует стандартное в ООП правило – объекту базового класса можно присваивать объект класса-наследника, но не наоборот):

```
Dim Pet, Dog As CPet()
Pet = New CPet()
Dog = New CDog() 'Так можно по правилам присваивания
Pet.Speak()      'Печатает "I'm a pet"
Dog.Speak()      'Так же печатает "I'm a pet"
```

Для организации полиморфного вызова методов применяется пара ключевых слов `Overridable` и `Overrides`. `Overridable` указывается для метода базового класса, который мы хотим сделать полиморфным. `Overrides` прописывается для методов производных классов. Эти методы должны совпадать по имени и сигнатуре с перекрываемым методом класса-предка.

```
Class CPet
    Public Overridable Sub Speak()
        Console.WriteLine("I'm a pet")
    End Sub
End Class
Class CDog
    Inherits CPet
    Public Overrides Sub Speak()
        Console.WriteLine("I'm a dog")
    End Sub
```

```
End Class
Dim Pet, Dog As CPet()
Pet = New CPet()
Dog = New CDog()
Pet.Speak()       'Печатает "I'm a pet"
Dog.Speak()       'Теперь печатает "I'm a dog"
```

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

```
Class CDog
    Inherits CPet
    Public NotOverridable Sub Speak()
```

Для методов абстрактных классов (классов с модификатором `MustInherit`) возможно задать модификатор `MustOverride`, который говорит о том, что данная подпрограмма не реализуется в классе, а должна обязательно переопределяться в классе-наследнике.

```
MustInherit Class AbstractClass
    'Реализации в классе нет
    Public MustOverride Sub AbstractMethod()
End Class
```

Отметим, что наряду с виртуальными методами в Visual Basic .NET можно описать виртуальные свойства (свойство транслируется в методы). Разделяемые члены класса не могут быть виртуальными.

Рассмотрим один из нюансов использования виртуальных методов при наследовании. Пусть имеется класс, содержащий два метода, первый из которых – виртуальный, а второй метод вызывает первый. Пусть также имеется наследник этого класса, перекрывающий первый виртуальный метод:

```
Class Base
    Overridable Sub Proc1()
        Console.WriteLine("This is Proc1 from Base")
    End Sub
    Sub Proc2()
        Console.WriteLine("This is Proc2 from Base")
        Proc1()
    End Sub
End Class
Class Inheritor
    Inherits Base
    Overrides Sub Proc1()
        Console.WriteLine("This is Proc1 from Inheritor")
    End Sub
End Class
```

Следующая программа

```
Module MainModule
    Sub Main()
        Dim X As New Inheritor()
```

```
            X.Proc2()
        End Sub
    End Module
```

выведет на экран строки

```
This is Proc2 from Base
This is Proc1 from Inheritor
```

Это ожидаемый эффект при использовании виртуальных методов. Ранее было упомянуто ключевое слово `Me`, которое обозначало текущий объект. Строка `Proc1()` из второго метода может быть заменена эквивалентной строкой `Me.Proc1()`. Однако для обращения к членам класса в языке Visual Basic .NET можно использовать и ключевое слово `MyClass`. Если при обращении к методу указать ключевое слово `MyClass`, то даже в случае перекрытия этого метода классом-наследником, будет вызываться метод базового класса. Таким образом, если переписать метод `Proc2()` как

```
Sub Proc2()
    Console.WriteLine("This is Proc2 from Base")
    MyClass.Proc1()
End Sub
```

то описанный в модуле код выведет на консоль строки

```
This is Proc2 from Base
This is Proc1 from Base
```

Различия между ключевыми словами `Me` и `MyClass` проявляются только в ситуациях, эквивалентных описанной (использование полиморфизма). Во всех других случаях использование этих ссылок дает одинаковый эффект.

20. Делегаты.

Делегат в VB .NET исполняет роль указателя на метод. В делегате инкапсулируется указатель на объект и адрес метода. Хотя делегаты, по сути, являются наследниками класса `System.MulticastDelegate`, они объявляется не как производные классы, а с использованием ключевого слова `Delegate`. При этом указывается тип подпрограммы (процедура или функция), имя делегата и сигнатура подпрограммы:

```
Delegate Function Y(ByVal X As Double) As Double
Delegate Sub IntegerSub(ByVal I As Integer)
```

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

После объявления класса делегата можно объявить переменные этого типа:

```
Dim Y1 As Y, SomeSub As IntegerSub
```

Переменные делегата инициализируются конкретными адресами подпрограмм при использовании ключевого слова `AddressOf` в формате `AddressOf`

<имя метода объекта>. При этом подпрограмма должна обладать подходящей сигнатурой. Компилятор автоматически вычисляет адрес объекта по имени объекта, определенному для метода. Если учитывать, что делегаты являются классами, то для инициализации делегата необходимо вызвать конструктор с одним параметром – адресом подпрограммы:

```
'Строка должна быть в том же типе, что и объявление MyFunction
Y1 = New Y(AddressOf MyFunction)
SomeSub = New IntegerSub(AddressOf Object1.Sub1)
```

Инициализация может выполняться и следующим оператором, в котором конструктор делегата вызывается неявно:

```
Y1 = AddressOf MyFunction
```

После того как делегат инициализирован, инкапсулированная в нем подпрограмма вызывается методом Invoke класса System.Delegate, который можно и опускать, указывая параметры непосредственно после имени переменной-делегата:

```
Y1.Invoke(0.5)
Y1(0.5)        'Такой способ верен, и он более короткий
```

Приведем пример использования делегатов. Опишем класс, инкапсулирующий процедуру вывода массива целых чисел.

```
Class ArrayPrint
    Shared Sub Print(A() As Integer, P As PrintMethod)
        Dim Element As Integer
        For Each Element In A
            P(Element)
        Next Element
    End Sub
End Class
```

Тип PrintMethod является делегатом, который описан следующим образом:

```
Delegate Sub PrintMethod(X As Integer)
```

Теперь можно описать модуль, который работает с приведенным классов и делегатом:

```
Module Test
    Sub ConsolePrint(I As Integer)
        System.Console.WriteLine(I)
    End Sub
    Sub FormatPrint(I As Integer)
        System.Console.WriteLine("Element is {0}", I)
    End Sub
    Sub Main()
        Dim A As Integer() = {1,20,30,100}
        Dim D As PrintMethod
        D = AddressOf ConsolePrint
        ArrayPrint.Print(A, D)
        D = AddressOf FormatPrint
```

```
        ArrayPrint.Print(A, D)
    End Sub
End Module
```

В результате работы данной программы на экран будут выведены следующие строки:

```
1
20
30
100
Element is 1
Element is 20
Element is 30
Element is 100
```

Обратите внимание, что в данном примере переменная `D` инициализировалась разделяемыми методами. Делегат не делает различие при работе между обычными методами объекта и разделяемыми методами класса.

Ключевой особенностью делегатов в языке Visual Basic .NET является то, что они могут инкапсулировать не один метод, а несколько. Для этого используются так называемые *групповые делегаты*. При этом при вызове группового делегата срабатывает вся цепочка инкапсулированных в нем методов.

Групповой делегат объявляется таким же образом, как и обычный. Затем создается несколько объектов делегата, и все они связывается с некоторыми методами. После этого используется метод `System.Delegate.Combine`, который получает в качестве параметров два объекта делегата и возвращает групповой делегат, являющийся объединением параметров. Модифицируем код модуля из предыдущего примера следующим образом:

```
Module Test
. . .
    Sub Main()
        Dim A As Integer() = {1,20,30,100}
        Dim First, Second, Result As PrintMethod
        First = AddressOf ConsolePrint
        Second = AddressOf FormatPrint
        Result = System.Delegate.Combine(First, Second)
        ArrayPrint.Print(A, Result)
    End Sub
End Module
```

Теперь результат работы программы выглядит следующим образом:

```
1
Element is 1
20
Element is 20
30
Element is 30
100
Element is 100
```

Если требуется удалить некий метод из цепочки группового делегата, то используется метод `System.Delegate.Remove`. Он имеет два параметра – имя группового делегата и имя удаляемого из цепочки делегата. Результатом работы метода является делегат с модифицированной цепочкой. Если из цепочки удаляют последний метод, результат вызова `System.Delegate.Remove` – значение `Nothing`. Следующий код удаляет метод `First` из цепочки группового делегата `Result`:

```
Result = System.Delegate.Remove(Result, First)
```

Как видим, делегаты являются мощным средством создания программного кода и используют преимущества объектно-ориентированной технологии.

21. Обработка событий.

События представляют собой способ описания связи одного объекта с другими. Работу с событиями можно условно разделить на три этапа:

1. объявление события в классе (*publishing*);
2. регистрация получателя события (*subscribing*);
3. генерация события (*raising*).

Событие можно объявить в пределах класса, структуры, модуля или интерфейса. Различают *явное* и *неявное* объявление событий. При явном объявлении события сигнатура события описывается в месте объявления. Синтаксис явного объявления события следующий:

```
Event <имя события>([<список аргументов события>])
```

Ключевое слово `Event` указывает на объявление события. Объявление события может предваряться модификаторами доступа (по умолчанию принимается модификатор `Friend`). Синтаксис объявления списка аргументов события, которые будут передаваться его обработчику, совпадает с синтаксисом задания аргументов в подпрограммах. Однако в списке должны отсутствовать ключевые слова `Optional` и `ParamArray` и выражения значений по умолчанию для аргументов. Если событие в классе реализует какое-либо событие интерфейса, то после имени события и списка аргументов можно поместить команду `Implements`.

Приведем пример класса с явным объявлением события:

```
Class CEventClass
    Private Data As Integer
    Event OnWrongData(Value As Integer)
    . . .
End Class
```

Фактически, события являются полями типа делегатов. Явное объявление события транслируется компилятором в следующий набор объявлений в классе:

a. в классе объявляется вложенный делегат с именем *Name*`EventHandler`, где *Name* - это имя события; вложенный делегат объявляется с переопределенным конструктором, методами `Invoke`, `BeginInvoke`, `EndInvoke`;
b. в классе объявляется `Private`-поле с именем *Name*`Event` и типом *Name*`EventHandler`;

с. в классе объявляются два метода с именами `add_Name` и `remove_Name` для добавления и удаления обработчиков события.

Явное объявление события обладает следующим очевидным недостатком: если в классе есть несколько событий с одинаковой сигнатурой, для *каждого* события будет создан свой тип делегата. Выходом является неявное объявление события, при котором вместо сигнатуры события указывается тип делегата, объявленного отдельно. Модифицируем представленный класс `CEventClass` для неявного объявления события:

```
Class CEventClass
    Private Data As Integer
    Delegate EventDelegat(Value As Integer)
    Event OnWrongData As EventDelegat
    . . .
End Class
```

Как видим, при неявном объявление сразу после имени события указывается при помощи `As` тип делегата.

Рассмотрим этап генерации события. Процесс генерации события производится оператором `RaiseEvent` в виде

```
RaiseEvent <имя события>([<список фактических аргументов>])
```

Генерация события может происходить в одном из методов того класса, в котором объявлено событие. Генерировать в одном классе события других классов нельзя.

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

```
Class CExampleClass
    Private FInt As Integer
    Public Event OnErrorEvent(I As Integer)
    Public Sub SeTField(I As Integer)
        FInt = I
        If I < 0 Then RaiseEvent OnErrorEvent(I)
    End Sub
End Class
```

Рассмотрим этап регистрации получателя события. Для того чтобы отреагировать на событие, его надо ассоциировать с *обработчиком события*. Обработчиком события может быть процедура, но не функция. Возможны два подхода к ассоциации. При *динамическом подходе* во время выполнения программы можно подключать и отключать процедуры, обрабатывающие событие. Для этого используются ключевые слова `AddHandler` и `RemoveHandler`:

```
AddHandler <имя объекта>.<событие>, AddressOf <обработчик>
RemoveHandler <имя объекта>.<событие>, AddressOf <обработчик>
```

Приведем пример динамической ассоциации:

```
Dim Obj As New CExampleClass()
```

```
Public Sub MyErrorHandling(ByVal A As Integer)
    Console.WriteLine(A)
End Sub
. . .
AddHandler Obj.OnErrorEvent, AddressOf MyErrorHandling
Obj.DoSomething(-10)
RemoveHandler Obj.OnErrorEvent, AddressOf MyErrorHandling
```

При *статическом подходе* объект класса, содержащего события, объявляется с ключевым словом `WithEvents`, помещаемым сразу после `Dim`:

```
Dim WithEvents Obj As New CExampleClass()
```

Ключевое слово `WithEvents` применяется только для полей класса или модуля. Ни для локальных переменных, ни для полей структуры это ключевое слово применить нельзя.

Та процедура, которая будет являться обработчиком для события объекта, должна содержать в заголовке после списка аргументов ключевое слово `Handles`, после которого указано имя события в некотором объекте.

```
Public Sub MyErrHandling(ByVal A As Integer) Handles _
                                                Obj.OnErrorEvent
    Console.WriteLine(A)
End Sub
. . .
Obj.DoSomething(-10)
```

Даже если обработчик события задан статически, его можно убрать командой `RemoveHandler` или назначить командой `AddHandler`. Нужно иметь в виду, что `AddHandler` добавляет обработчик к списку обработчиков, а не замещает старый обработчик. Таким образом, используя `AddHandler` и `RemoveHandler`, можно динамически составлять список обработчиков одного события, выполняющихся последовательно.

22. Интерфейсы.

В языке VB.NET запрещено множественное наследование классов. Тем не менее, в VB.NET существует концепция, позволяющая имитировать множественное наследование. Эта концепция *интерфейсов*. Интерфейс представляет собой набор объявлений свойств, методов и событий. Класс или структура могут *реализовывать* определенный интерфейс. В этом случае они берут на себя обязанность предоставить полную реализацию членов интерфейса (хотя бы пустыми методами). Можно сказать так: интерфейс – это контракт, пункты которого суть свойства, методы и события; если пользовательский тип реализует интерфейс, он берет на себя обязательство выполнить этот контракт.

Объявление интерфейса схоже с объявлением класса. Для интерфейса используется ключевое слово `Interface`. Интерфейс содержит только заголовки методов, свойств и событий:

```
Interface IBird
    Sub Fly()
    Property Speed As Double
```

```
End Interface
```

Обратите внимание – в определении членов интерфейса отсутствуют модификаторы уровня доступа. Для методов возможно использование только модификатора `Overloads`, для свойств – модификатора `Default`.

Если класс собирается реализовать интерфейс `IBird`, то он обязуется содержать процедуру без параметров и свойство, доступное и для чтения и для записи, имеющее тип `Double`. Имена при этом не существенны, главное это реализация сигнатуры.

Чтобы показать, что класс реализовывает некий интерфейс, используется команда `Implements <имя интерфейса>`, располагаемая в отдельной строке после имени класса и, возможно, команды `Inherits`. Кроме этого, ключевое слово `Implements` используется при описании членов класса, реализующих соответствующие члены интерфейса. При этом используется синтаксис `Implements <имя интерфейса>.<имя члена интерфейса>`:

```
Class CFalcon
    Implements IBird
    Private FS As Double
    Sub DoSomeThing()
        Console.WriteLine("Falcon Flys")
    End Sub
    Sub FalconFly() Implements IBird.Fly
        Console.WriteLine("Falcon Flys")
    End Sub
    Property FalconSpeed As Double Implements IBird.Speed
        Get
            Return FS
        End Get
        Set (Value As Double)
            FS = Value
        End Set
    End Property
End Class
```

При реализации в классе членов некоторого интерфейса запрещается использование модификатора `Shared`, так как члены интерфейса должны принадлежать конкретному объекту, а не классу в целом.

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:

```
'Объявили переменную интерфейсного типа
Dim Bird As IBird
'Инициализировали объектом подходящего класса
Bird = New CFalcon()
'Фактически вызывается CFalcon.FalconFly()
Bird.Fly()
'Ошибка компиляции! В IBird нет такого метода!
Bird.DoSomeThing()
```

Если в программе необходимо проверить, реализует ли объект `Obj` некоего класса интерфейс `Inter`, то можно воспользоваться встроенной функцией `TypeOf` и оператором `Is` в форме

```
'Выражение равно True, если Obj реализует Inter
TypeOf(Obj) Is Inter . . .
```

Один класс может реализовывать несколько интерфейсов, при этом имена интерфейсов перечисляются через запятую в команде `Implements`:

```
Interface ISwimable
    Sub Swim()
End Interface

Class CDuck
    Implements IBird, ISwimable
    Public Sub DuckFly() Implements IBird.Fly
        Console.WriteLine("Duck Flys")
    End Sub
    Public Sub DuckSwim() Implements ISwimable.Swim
        Console.WriteLine("Duck Swims")
    End Sub
. . .
End Class
```

Один метод может реализовывать несколько методов из разных интерфейсов, при этом методы перечисляются через запятую после `Implements`:

```
Public Sub SomeSub() Implements _
                          Inter1.Sub1, Inter1.Sub2, Inter2.Sub3
```

Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом, в отличие от классов, наследование интерфейсов может быть множественным.

```
Interface IWaterfowl
    Inherits ISwimable, IBird
    Sub Flush() 'Взлетать – для птиц
End Interface
```

Иерархия интерфейсов позволяет построить (подобно иерархии классов) деление множества формализуемых понятий на подмножества. Интерфейсы позволяют выделить в отдельную иерархию характеристики действий рассматриваемых объектов.

Библиотека .NET Framework содержит несколько полезных интерфейсов, которые могут использоваться в пользовательских классах. Рассмотрим некоторые из них, определенные в пространстве имен `System`.

Интерфейс `ICloneable` обеспечивает стандартный механизм копирования как ссылочных, так и структурных объектов. Данный интерфейс определяет метод `Clone()`:

```
Function Clone() As Object
```

Переопределяя метод `Clone()` в собственном классе, программист записывает свой метод копирования данных одного экземпляра класса в другой.

Интерфейс `IDisposable` реализуется теми пользовательскими типами, которые должны удаляться из памяти особым образом. Реализуя интерфейс `IDisposable`, объект сообщает своим пользователям, что перед окончанием работы с объектом они должны вызвать метод `Dispose()`. Если пользователь объекта этого не сделает, это может помешать нормальной деинициализации объекта.

Для того чтобы поддерживалось сравнение объектов пользовательских типов, тип должен реализовывать интерфейс `IComparable`. Данный интерфейс содержит единственный метод-функцию `CompareTo`, который получает в качестве параметра произвольный объект и должен вернуть положительное значение, если текущий экземпляр «больше» переданного объекта, отрицательное значение, если экземпляр «меньше» объекта и ноль, если экземпляр и объект равны.

23. Структуры.

Структура – это пользовательский тип, поддерживающий всю функциональность класса, кроме наследования. Тип структуры, определенный в языке VB. NET, в простейшем случае являются аналогом типа «запись» языка Pascal, то есть позволяет инкапсулировать несколько полей различных типов. Но членами структуры в VB. NET могут быть не только поля, а и методы, свойства, события, константы. Синтаксис определения структуры следующий:

```
Structure <имя структуры>
    <члены структуры>
End Structure
```

Структура должна содержать, по крайней мере, одно поле или событие. Поля структуры не могут инициализироваться при объявлении, инициализация возможна только для констант. Если в качестве поля структуры выступает массив, то для него нельзя задать начальный размер при объявлении.

Как и класс, структура может содержать конструкторы. В структуре можно объявить только пользовательский конструктор с параметрами.

Рассмотрим пример структуры для представления комплексных чисел:

```
Structure Complex
    Public Re, Im As Double
    Sub New(ByVal X As Double, ByVal Y As Double)
        Re = X
        Im = Y
    End Sub
    Function Add(ByVal Z As Complex) As Complex
        Add.Re = Me.Re + Z.Re
        Add.Im = Me.Im + Z.Im
    End Function
End Structure
```

Переменные структур определяются как обычные переменные:

```
Dim Z1, Z2 As Complex
```

Содержимое полей переменных Z1 и Z2 будет считаться неопределенным, пока полям не присвоят какое-либо значение. Если в структуре описан конструктор, его можно вызвать для начальной инициализации:

```
Dim Z3 As New Complex(1.0, 2.0)
```

Доступ к членам структуры осуществляется так же, как к членам объекта класса:

```
Z1.Re = 10.0
Z1.Im = 5.0
Z2 = Z3.Add(Z1)
```

Структуры, как и классы, могут реализовывать интерфейсы.

Напомним, что структуры являются структурными переменными, а не ссылочными. Переменные структур размещаются в стеке приложения. Структурные переменные можно присваивать друг другу, при этом выполняется копирование данных структуры на уровне полей.

24. Перечисления.

Перечисление – это тип, содержащий в качестве членов только именованные целочисленные константы. Рассмотрим синтаксис определения перечисления:

```
Enum <имя перечисления> [As <тип перечисления>]
<элемент перечисления 1> [= <значение элемента>]
. . .
<элемент перечисления N> [= <значение элемента>]
End Enum
```

Перечисление может предваряться модификатором доступа. Если задан тип перечисления, то он определяет тип каждого элемента перечисления. Типами перечислений могут быть только Byte, Short, Integer, Long. По умолчанию принимается тип Integer. Для элементов перечисления область видимости указать нельзя. Значением элемента перечисления должна быть целочисленная константа. Если для какого-либо элемента перечисления значение опущено, то в случае, если это первый элемент, он принимает значение 0, иначе элемент принимает значение на единицу большее предыдущего элемента. Заданные значения элементов перечисления могут повторяться.

Приведем примеры определения перечислений:

```
Enum Seasons
    Winter
    Spring
    Summer
  Autumn
End Enum
Public Enum ErrorCodes As Byte
    First = 1
    Second = 2
    Fourth = 4
End Enum
```

После описания перечисления можно объявить переменную соответствующего типа:

```
Dim S As Seasons, EC As ErrorCodes
```

При помощи данной переменной можно получить доступ к элементам перечисления:

```
Console.WriteLine(S.Spring)    'Печатает 1
```

Для этой же цели можно использовать и имя перечисления (чаще поступают именно так):

```
Console.WriteLine(Seasons.Summer)     'Печатает 2
```

С другой стороны, переменной типа перечисления можно присваивать значения, как и обычной переменной:

```
Dim S As Seasons
S = Seasons.Spring
Console.WriteLine(S)    'Печатает 1
```

Если включена жесткая проверка типов (`Option Strict On`), то переменной перечисления можно присвоить значения только элементов перечисления. В противном случае такой переменной можно присвоить любое значение базового типа перечисления.

```
Dim S As Seasons
S = 1000      'Это строка компилируется, если Option Strict Off
Console.WriteLine(S)    'Печатает 1000
```

Перечисления фактически являются наследниками типа `System.Enum`. При компиляции проводится простая подстановка соответствующих значений для элементов перечислений. Для элементов перечислений можно использовать арифметические операторы и операторы сравнения.

25. Пространства имен.

Пространства имен служат для логической группировки пользовательских типов. Применение пространств имен обосновано в крупных программах для снижения риска конфликта имен и улучшения структуры объектных библиотек.

Синтаксис описания пространства имен следующий:

```
Namespace <имя пространства имен>
[<компоненты пространства имен>]
End Namespace
```

Компонентами пространства имен могут быть модуль, классы, делегаты, перечисления, структуры и другие пространства имен. Само пространство имен может быть вложено только в другое пространство имен. Это означает, что строгая логическая структура одного исходного файла программы должна выглядеть следующим образом:

```
[опции компилятора]
[секция импортирования]
[атрибуты уровня пространства имен]
```

```
Namespace <имя пространства имен>
'Здесь помещаются пользовательские типы: модули, классы,
'перечисления, делегаты, интерфейсы и другие пространства имен
End Namespace
```

Если в разных местах программы (возможно, в разных входных файлах) определены несколько пространств имен с одинаковыми именами, компилятор собирает компоненты из этих пространств в общее пространство имен. Для этого только необходимо, чтобы одноименные пространства имен находились на одном уровне вложенности в иерархии пространств имен.

Для доступа к компонентам пространства имен используется синтаксис `<имя пространства имен>.<имя компонента>`. Для компилируемых входных файлов имя пространства имен по умолчанию (если в файле нет обрамляющего пространства имен) можно задать специальной опцией компилятора.

Для использования в программе некоего пространства имен служит команда `Imports`. Ее синтаксис следующий:

```
Imports [<имя псевдонима> =] <имя пространства>[.<элемент>]
```

Импортирование пространства имен позволяет сократить трудозатраты программиста при наборе текстов программ. Псевдоним, используемый при импортировании, это обычно короткий идентификатор для ссылки на пространство имен в тексте программы. Импортировать можно пространства имен из текущего проекта, а также из подключенных к проекту сборок и других подключаемых проектов.

26. Обработка и генерация исключительных ситуаций.

Опишем возможности обработки и генерации исключительных ситуаций в языке Visual Basic .NET.

Рассмотрим синтаксис генерации исключительной ситуации. Для генерации исключительной ситуации используется оператор `Throw` со следующим синтаксисом:

```
Throw <объект класса исключительной ситуации>
```

Обратите внимание: объект, указанный после `Throw`, должен обязательно быть объектом класса исключительной ситуации. Таким классом является класс `System.Exception` и все его наследники.

Рассмотрим пример программы с генерацией исключительной ситуации:

```
Imports System
Class CExample
    Private fX As Integer
    Sub SetFx(X As Integer)
        If X > 0 Then
            fX = X
        Else
            Throw New Exception() 'Объект создается «на месте»
        End If
    End Sub
End Class
```

```
Module M1
    Sub Main()
        Dim A As New CExample()
        A.SetFx(-3)    'Строка вызовет исключительную ситуацию
    End Sub
End Module
```

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

Класс `System.Exception` является стандартным классом для представления исключительных ситуаций. Основными членами данного класса является свойство только для чтения `Message`, содержащее строку с описанием ошибки, и перегруженный конструктор с одним параметром-строкой, записываемой в свойство `Message`. Естественно, библиотека классов .NET Framework содержит большое число разнообразных классов, порожденных от `System.Exception` и описывающих конкретные исключительные ситуации.

Пользователь может создать собственный класс для представления информации об исключительной ситуации. Единственным условие является прямое или косвенное наследование этого класса от класса `System.Exception`.

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

```
Class MyException
    Inherits Exception
    Public Info As Integer
End Class
Class CExample
    Private fX As Integer
    Sub SetFx(X As Integer)
        If X > 0 Then
            fX = X
        Else
            Dim E As New MyException()
            E.Info = X
            Throw E
        End If
    End Sub
End Class
```

Опишем возможности по обработке исключительных ситуаций. Для перехвата исключительных ситуаций служит блок `Try - Catch`. Синтаксис блока следующий:

```
Try
    [<операторы, способные вызвать исключительную ситуацию>]
[<один или несколько блоков Catch>]
[Finally
<операторы из секции завершения>]
End Try
```

Операторы из части `Finally` (если она присутствует) выполняются всегда, вне зависимости от того, произошла исключительная ситуация или нет. Если один из операторов, расположенных между `Try` и блоком `Catch` вызвал исключительную ситуацию, управление немедленно передается на блоки `Catch`. Синтаксис отдельного блока `Catch` следующий:

```
Catch [<идентификатор объекта ИС> As <тип ИС>]
```

`<идентификатор объекта ИС>` - это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации. Отдельно описывать эту переменную в секции `Dim` не нужно.

Модифицируем модуль, описанный выше, добавив в него блок перехвата ошибки:

```
Module M1
    Sub Main()
        Dim A As New CExample()
        Try
            Console.WriteLine("Эта строка печатается")
            A.SetFx(-3)
            Console.WriteLine("Не печатается, если ошибка")
        Catch ex As MyException
            Console.WriteLine("Ошибка, если {0}", ex.Info)
        Finally
         Console.WriteLine("Строка печатается - блок Finally")
        End Try
    End Sub
End Module
```

Если используется несколько блоков `Catch`, то обработка исключительных ситуаций должна вестись по принципу «от частного к общему», так как после выполнения одного блока `Catch` управление передается на часть `Finally` (при отсутствии `Finally` – на оператор после `End Try`):

```
. . .
Try
    Console.WriteLine("Эта строка печатается")
    A.SetFx(-3)
    Console.WriteLine("Эта строка не печатается, если ошибка")
Catch ex As Exception
    Console.WriteLine("Общий перехват")
Catch ex As MyException
    Console.WriteLine("Теперь строка не печатается никогда!")
Finally
    Console.WriteLine("Эта строка печатается - блок Finally")
End Try
```

Если в блоке `Catch` не указать идентификатор объекта и тип исключительной ситуации, то такой блок будет обрабатывать любые ошибки (в общем, это почти аналогично объявлению `Catch ex As Exception`).

Заголовок блока `Catch` может содержать ключевое слово `When`, после которого следует булевское выражение. В этом случает блок обрабатывает ошибку,

только если это выражение истинно. Рассмотрим пример, который иллюстрирует применение этой возможности:

```
Option Strict On
Imports System
Module ExampleModule
    Sub Main()
        Dim X, Y As Integer
        Console.WriteLine("Введите A")
        Dim A As Integer = CInt(Console.ReadLine())
        Console.WriteLine("Введите B")
        Dim B As Integer = CInt(Console.ReadLine())
        Try
            X = 10 \ A 'Используется целочисленное деление
            y = 50 \ B
        Catch When A = 0
            Console.WriteLine("Ошибку вызвало деление на A")
        Catch When B = 0
            Console.WriteLine("Ошибку вызвало деление на B")
        End Try
    End Sub
End Module
```

Как видим, применение `When` позволяет более гибко отслеживать ситуации, приводящие к ошибке.

В блоке `Try - Catch` в любом месте возможно использование оператора `Exit Try`, который вызывает немедленный переход на операторы, находящиеся после `End Try`.

27. Жизненный цикл объектных переменных.

Как уже отмечалось выше, все пользовательские типы в языке Visual Basic .NET можно разделить на ссылочные и структурные. Переменные структурных типов создаются выполняющей средой в стеке. «Время жизни» (lifetime) переменных структурного типа обычно ограничено тем блоком кода, в котором они объявляются. Например, если переменная, соответствующая пользовательской структуре, объявлена в некой подпрограмме, то после выхода из подпрограммы память, занимаемая переменной, автоматически освободится.

Переменные ссылочного типа, далее называемые для краткости объектами, размещаются в «куче». В отличие от большинства традиционных языков программирования, «куча» в VB.NET является *управляемой* (managed heap). Объясним значение этого понятия. Если в пользовательской программе превышен некий порог расходования ресурсов (захвачено слишком много памяти), средой выполнения программы выполняется процесс, называемый *сборка мусора*. Сборка мусора заключается в следующем. Среда выполнения отслеживает все используемые объекты и определяет реально занимаемую этими переменными память. После этого вся оставшаяся память освобождается, то есть помечается как свободная для использования. Освобождая память, среда выполнения заново размещает «уцелевшие» объекты в куче, чтобы уменьшить ее фрагментацию. Ключевой особенность сборки мусора является то, что она осуществляет-

ся средой выполнения автоматически и независимо от основного потока инструкций приложения. По этой причине язык VB.NET относят к языкам с *автоматической сборкой мусора*[1].

Обсудим автоматическую сборку мусора с точки зрения программиста, разрабатывающего некий класс. С одной стороны, такой подход к освобождению памяти, занимаемой объектом, имеет свои преимущества. В частности, практически исключаются случайные утечки памяти, которые могут вызвать «забытые» объекты.

```
Class ClassWithManyFields
'Предположим, что данный класс содержит столько полей,
'что они занимают 1 мегабайт памяти
End Class
. . .
Dim A As New ClassWithManyFields() 'Захватили 1 Мб
Dim B As New ClassWithManyFields() 'Захватили еще 1 Мб
A = B 'Невинная строка, но как теперь освободить память,
      'которая была выделена под A?
'К счастью, выполняемая среда освободит ее сама!
```

С другой стороны, размещаемый объект может захватывать некоторые особо ценные ресурсы (например, подключения к базе данных), которые требуется освободить сразу после того, как объект больше не используется. В этой ситуации выходом является написание некого особого метода, который содержит код освобождения ресурсов.

```
Class ClassGetResourses
    . . .
    Sub FreeResourse()
        'Здесь код, который освобождает ресурсы
    End Sub
End Class
. . .
Dim A As New ClassGetResourses()
A.DoSomething()
'Поработали с объектом, освободим ресурсы
A.FreeResourse()
```

Но как *гарантировать* освобождение ресурсов, даже если ссылка на объект была случайно утеряна? Оказывается, в классе `System.Object` существует виртуальная процедура `Finalize` без параметров. Данная процедура имеет уровень доступа `Protected`. Если пользовательский класс при работе резервирует некие ресурсы, он может переопределить `Finalize` для их освобождения. Объекты классов, имеющих реализацию `Finalize` при «сборке мусора» обрабатываются особо. Когда сборщик мусора распознает, что уничтожаемый объект имеет реализацию метода `Finalize`, он откладывает уничтожение объекта. Через некоторое время в отдельном программном потоке происходит вызов метода `Finalize` и фактическое уничтожение объекта.

[1] Фактически, так как автоматическая сборка мусора выполняется средой выполнения, то это свойство всей платформы .NET, а не конкретного языка.

```
Class ClassFinalize
    Sub DoSomething()
        System.Console.WriteLine("I'm working!")
    End Sub
    Protected Overrides Sub Finalize()
        System.Console.WriteLine("Bye-bye!")
    End Sub
End Class
Module M1
    Sub Main()
        Dim A As New ClassFinalize()
        A.DoSomething()
        A = Nothing '"Сдох"? Еще нет!
        System.Console.WriteLine("Main program")
    End Sub
End Module
```

Данная программа выводит следующие строки:

```
I'm working!
Main program
Bye-bye!
```

Метод `Finalize` можно воспринимать как некий аналог традиционного деструктора. Проблема с использованием `Finalize` заключается в том, момент вызова этого метода очень сложно отследить. Альтернативой использованию метода `Finalize` является реализация классом интерфейса `IDisposable`. Напомним, что данный интерфейс имеет единственный метод `Dispose()`, куда помещается завершающий код работы с объектом[1].

Как и всё в языке VB.NET, сборщик мусора является неким классом. За сборку мусора отвечает класс `System.GC`. Метод `Collect()` данного класса вызывает принудительную сборку мусора в программе. Не рекомендуется пользоваться методом `Collect()` часто, так как любая сборка мусор требует значительного расхода ресурсов.

28. Иерархия типов VB. NET.

Как уже было замечено ранее, все типы (и примитивные, и пользовательские) в языке VB.NET образуют особую иерархию. На рис. 6 приведена схема иерархии пользовательских и примитивных типов.

[1] C# имеет специальную обрамляющую конструкцию `using`, которая *гарантирует* вызов метода `Dispose()` для объектов, использующихся в блоке. К сожалению, подобной конструкции в языке Visual Basic.NET нет.

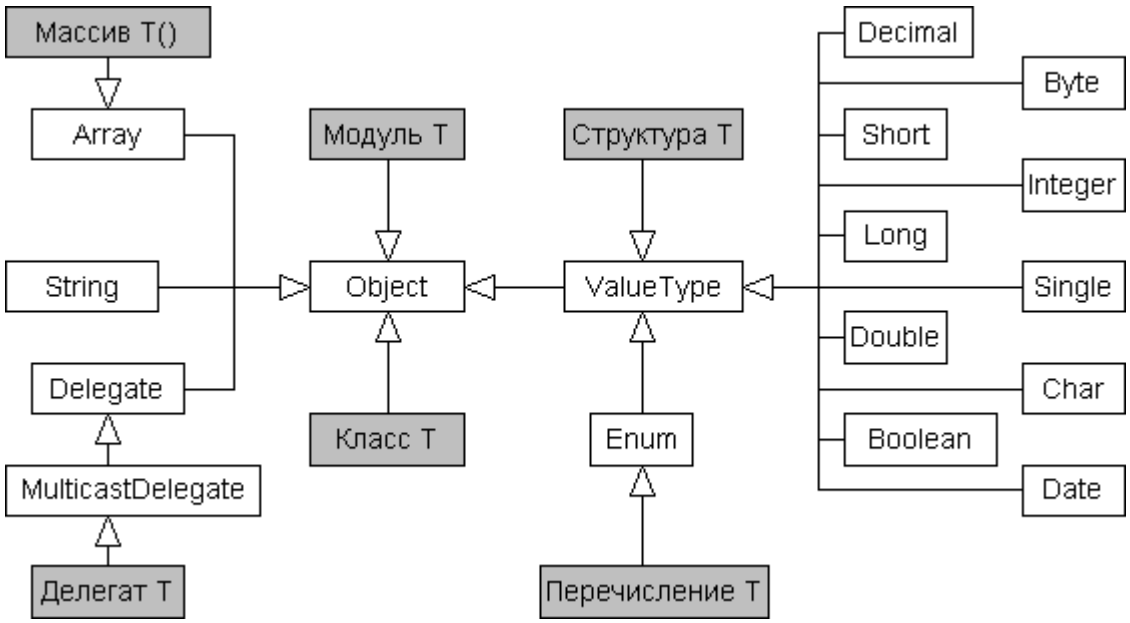

Рис. 6. Иерархия типов в VB.NET

Рассмотрим подробнее классы, составляющие данную иерархию. Любой тип, используемый в VB.NET, является наследником типа `System.Object`. Коротким синонимом этого типа в VB.NET является `Object`. Так как любой тип – это наследник `Object`, то переменная типа `Object` может принимать значения объекта любого типа, и методы `Object` доступны для любой переменной (и даже литерала). Основные методы данного класса приведены в таблице 7.

Таблица 7

Методы типа `Object`

Имя метода	Описание
`Equals`	Данный виртуальный метод проверяет, обладают ли два объекта одинаковыми данными, и может переопределяться в производных классах.
`ToString`	Возвращает представление объекта в строковом формате, допускает переопределение в производных классах. По умолчанию возвращает полное имя типа.
`GetHashCode`	Возвращает хэш-код для значения данного объекта, допускает переопределение, если объекты используются в качестве ключа в хэш-таблице.
`GetType`	Возвращает тип объекта.
`Finalize`	Защищенная виртуальная процедура. Если пользовательский тип при работе резервирует некие ресурсы, он может переопределить `Finalize` для их освобождения.

Рассмотрим пример использования методов класса `Object`. Опишем следующий тип:

```
Structure Person
    Public Name As String
    Public Age As Integer
    Overrides Function ToString() As String
        Return "Name: " & Name & " Age = " & Age.ToString()
    End Function
    Overrides Overloads Function Equals(A As Object) _
```

```
                                                    As Boolean
            Return Name = CType(A, Person).Name
        End Function
    End Structure
```

Следующий фрагмент кода использует описанную структуру:

```
Dim A, B As Person
A.Name = "Mister X"
A.Age = 30
B.Name = "Mister X"
B.Age = 25
Console.WriteLine(A) 'Используется перегруженный WriteLine()
Console.WriteLine(B) 'ToString для представления данных
Console.WriteLine(A.Equals(B))
```

Этот код выводит следующие строки:

```
Name: Mister X Age = 30
Name: Mister X Age = 25
True
```

Обратите внимание, что при переопределении методов `ToString` и `Equals` использовалось ключевое слово `Overrides`, так как эти методы являются виртуальными.

Все массивы, используемые в VB.NET, являются неявными потомками класса `System.Array`. Опишем основные методы и свойства класса `System.Array`.

Таблица 8

Свойства и методы класса `System.Array`

Имя члена	Описание
`Rank`	Свойство только для чтения, возвращает размерность массива.
`Length`	Свойство только для чтения, возвращает общее число элементов массива.
`GetLength`	Метод, который возвращает число элементов в указанном измерении.
`GetLowerBound`	Метод возвращает нижнюю границу для указанного измерения (чаще всего это 0).
`GetUpperBound`	Метод возвращает верхнюю границу для указанного измерения (чаще всего это количество элементов минус 1).
`GetValue`	Метод возвращает значение элемента с указанными индексами.
`SetValue`	Метод устанавливает значение элемента с указанными индексами (значение – первый аргумент).
`Sort`	Разделяемый метод, который сортирует массив, переданный в качестве параметра. Тип элемента массива должен иметь реализацию интерфейса `IComparable`.
`BinarySearch`	Разделяемый метод. Поиск элемента в отсортированном массиве. Тип элемента массива должен иметь реализацию интерфейса `IComparable`.
`IndexOf`	Разделяемый метод. Возвращает индекс первого вхождения своего аргумента в одномерный массив или –1, если элемента в массиве нет.
`LastIndexOf`	Разделяемый метод. Возвращает индекс последнего вхождения своего аргумента в одномерный массив или –1, если элемента в массиве нет.

Reverse	Разделяемый метод. Меняет порядок элементов в одномерном массиве или его части на противоположный.
Copy	Разделяемый метод. Копирует раздел одного массива в другой массив, выполняя приведение типов.
Clear	Разделяемый метод. Устанавливает для диапазона элементов массива значение по умолчанию для типов элементов.
CreateInstance	Разделяемый метод. Динамически создает экземпляр массива любого типа, размерности и длины.

Теперь рассмотрим примеры использования данных методов и свойств. В примерах выводимые данные записаны как комментарии. Вначале использование нескольких простых членов System.Array:

```
Dim M(,) As Integer = {{1,3,5},{10,20,30}}
    Console.WriteLine(M.Rank)      '2
    Console.WriteLine(M.Length)    '6
    Console.WriteLine(M.GetLowerBound(0))  '0
    Console.WriteLine(M.GetUpperBound(1))  '2
```

Продемонстрируем сортировку и поиск в одномерном массиве:

```
Dim M() As Integer = {1, -3, 5, 10, 2, 5, 30}
Console.WriteLine(Array.IndexOf(M, 5))       '2
Console.WriteLine(Array.LastIndexOf(M, 5))  '5
Array.Reverse(M)
Dim I As Integer
For Each I In M
    Console.WriteLine(I)  '30, 5, 2, 10, 5, -3, 1
Next
Array.Sort(M)
For Each I In M
    Console.WriteLine(I)  '-3, 1, 2, 5, 5, 10, 30
Next
Console.WriteLine(Array.BinarySearch(M, 10)) '5
```

Для того чтобы поддерживалась сортировка массивов пользовательских типов, тип должен реализовывать интерфейс IComparable. Рассмотрим пример использования данного интерфейса:

```
Structure Person
    Implements IComparable
    Public Name As String
    Public Age As Integer
    Function Comp(Obj As Object) As Integer _
                                Implements IComparable.CompareTo
        Dim Tmp As Person
        Tmp = CType(Obj, Person)
        If Age < Tmp.Age Then
            Return 1
        ElseIf Age > Tmp.Age Then
            Return -1
        Else Return 0
        End If
    End Function
```

```
End Structure

Dim Data(2) As Person
Dim I As Integer
Data(0).Name = "John"
Data(0).Age = 25
Data(1).Name = "Bob"
Data(1).Age = 15
Data(2).Name = "Marry"
Data(2).Age = 20
Array.Sort(Data)
For I = 0 To Data.Length-1
    Console.Write(Data(I).Name)
    Console.WriteLine(Data(I).Age)
Next I
```

Существует возможность организации нескольких способов сортировки одного массива (реализация интерфейса `IComparer` и использование перегруженного метода `Array.Sort`), однако нами она рассматриваться не будет. Также упомянем о возможности использования оператора `For Each` для собственных типов, содержащих встроенные наборы данных (эта возможность связана с реализацией интерфейсов `IEnumerator` и `IEnumerable`).

Опишем процесс динамического создания массива. Данный способ позволяет задать для массивов произвольные нижние и верхние границы. Допустим, нам необходим двумерный массив из элементов `Decimal`, первая размерность которого представляет годы в диапазоне от 1995 до 2004, а вторая – кварталы в диапазоне от 1 до 4. Следующий код осуществляет создание массива и обращение к элементу массива. Предполагается, что включена опция жесткой проверки типов:

```
Option Strict On
. . .
'Назначение вспомогательных массивов понятно из их названий
Dim LowerBounds() As Integer = {1995,1}
Dim Lengths() As Integer = {10,4}
'"Заготовка" для будущего массива
Dim Target(,) As Decimal
Target = CType(Array.CreateInstance(GetType(Decimal), _
                              Lengths, LowerBounds), Decimal(,))
Target(2000,1) = 10.3D
```

Допустимо было написать следующий код для создания массива:

```
Dim Target As Array
Target = Array.CreateInstance(GetType(Decimal), _
                                          Lengths, LowerBounds)
```

В этом случае для установки и чтения значений элементов необходимо было бы использовать методы `SetValue` и `GetValue`:

```
Target.SetValue(10.3D, 2000, 1)
Console.WriteLine(Target.GetValue(2000, 1))
```

Рассмотрим класс System.String. Он служит для представления строк. Любая строка в VB.NET является строкой произвольной длины, состоящей из символов в формате Unicode. Класс System.String содержит большое число методов и свойств, реализующих типичные строковые функции. Некоторые из членов приведены в таблице 9:

Таблица 9

Свойства и методы класса System.String

Имя члена	Описание
Length	Свойство только для чтения, возвращает число символов строки.
Chars	Свойство-массив только для чтения. Возвращает символ строки по индексу (нумерация начинается с нуля).
IndexOf	Метод, который возвращает индекс первого вхождения символа или подстроки в строку.
LastIndexOf	Метод, который возвращает индекс последнего вхождения символа или подстроки в строку.
IndexOfAny	Метод, который возвращает индекс первого символа строки, имеющегося в массиве заданных символов.
LastIndexOfAny	Метод, который возвращает индекс последнего символа строки, имеющегося в массиве заданных символов.
CompareTo	Метод сравнивает между собой две строки.
ToUpper	Метод преобразует символы строки к верхнему регистру.

Рассмотрим примеры работы со строками:

```
Dim S As String = "This is a string"
Console.WriteLine(S.Length)                 '16
Console.WriteLine(S.Chars(1))               'h
Console.WriteLine(S.IndexOf("i"C))          '2
Console.WriteLine(S.IndexOf("st"))          '10
Console.WriteLine(S.LastIndexOf("i"C))      '13
Console.WriteLine(S.ToUpper())              'THIS IS A STRING
```

Класс System.String реализует интерфейс IEnumerator. Это значит, что для получения последовательности всех символов строки можно использовать цикл For Each:

```
Dim S As String = "This is a string"
Dim C As Char
For Each C In S
    Console.Write(C)            'Выведет все символы строки S
Next
```

Полный перечень методов класса System.String можно найти в справочной системе .NET SDK.

Отметим наличие в VB. NET специальной команды Mid, которая позволяет изменить выбранную часть строки и используется слева от знака присваивания. У данной команды 3 параметра. Первый – это обрабатываемая строка, второй – позиция символа, с которого начинается замена, третий – количество заменяемых символов (символы нумеруются с единицы).

```
Dim Name As String = "Visual Basic. NET"
Mid(Name,4,1) = "X" 'Name = "VisXal Basic. NET"
```

Особенностью класса System.String является то, что объекты этого класса являются неизменными (при изменении строки всегда создается ее новая копия в памяти). Если происходит интенсивная работа со строками, для увеличения производительности рекомендуется использовать класс StringBuilder из пространства имен System.Text.

Классы System.Delegate и System.MulticastDelegate предназначены для создания пользовательских делегатов. Класс MulticastDelegate служит для создания групповых делегатов. Основные методы этих классов уже были рассмотрены ранее. Это такие методы как Invoke, Combine, Remove. Упомянем также методы для асинхронного вызова делегатов: BeginInvoke и EndInvoke. Для групповых делегатов метод GetInvocationList() класса System.MulticastDelegate возвращает массив, элементами которого являются делегаты, составляющие цепочку вызова. Это позволяет вызвать произвольный метод из цепочки.

Класс System.ValueType является основой для построения структурных типов. Пользовательская структура является прямым потомком класса System.ValueType. Альтернативный способ описания пользовательского структурного типа – использование перечислений, каждое из которых неявно порождено от класса System.Enum.

У классов, предназначенных для представления примитивных встроенных типов, можно отметить методы MinValue и MaxValue, позволяющие получить минимальное и максимальное возможное значение переменной типа, а также метод Parse, выполняющий преобразование строки в число.

29. Организация взаимодействия сборок.

Все рассмотренные ранее примеры приложений для платформы .NET были сравнительно простыми. Во-первых, весь код примеров располагался в одном файле, во-вторых, результатом компиляции кода было выполняемое (*.exe) приложение. В процессе разработки реальных приложений при использовании платформы .NET могут возникнуть более сложные сценарии.

Первый сценарий предполагает размещение исходного кода приложения в нескольких входных файлах. Для компиляции такого приложения достаточно вызывать компилятор, перечислив все необходимые входные файлы:

```
vbc.exe file1.vb file2.vb file3.vb
```

В приведенном примере три входных файла скомпилируются в выполняемое приложение file1.exe. Таким образом, имя результирующего файла совпадает с именем первого файла в списке исходных. Впрочем, подобная ситуация легко исправляется при необходимости, использованием специального ключа компилятора:

```
vbc.exe /out:file2.exe file1.vb file2.vb file3.vb
```

Второй сценарий: создание библиотек динамической компоновки. Для получения такой библиотеки не используются средства языка, а указывается специальный ключ компилятора:

```
vbc.exe /target:library file1.vb
```

Обратите внимание: библиотека динамической компоновки, полученная в среде .NET, является обычной сборкой. Таким образом, использовать такую библиотеку можно только в программах для .NET. Библиотека представляет собой набор обычных пользовательских типов. Так как библиотеку нельзя запустить как отдельное приложение, в типах библиотеки может отсутствовать метод `Main()`. (и, как правило, отсутствует, чтобы не «мешать» методу `Main()` основной программы).

Рассмотрим пример создания библиотеки динамической компоновки. Создадим простой класс, который поместим в библиотеку:

```
Public Class CExampleClass
    Private fX As Integer
    Public Sub SetField(NewX As Integer)
        fX = NewX
        System.Console.WriteLine("New field value is {0}", fX)
    End Sub
End Class
```

Обратите внимание, что и для класса и для его метода использовался модификатор доступа `Public`, так как они должны быть видны из других сборок. Скомпилируем файл с исходным текстом программы (пусть его название `MyLibrary.vb`), использовав ключ компилятора `/target:library`:

```
vbc.exe /target:library MyLibrary.vb
```

После компиляции мы получим файл `MyLibrary.dll`.

Чтобы использовать данную библиотеку в других приложениях, при компиляции приложений указывается ссылка на библиотеку.

Напишем небольшое приложение, использующее класс `CExampleClass`:

```
Imports System
Module MyModule
    Sub Main()
        Dim A As New CExampleClass()
        A.SetField(100)
        Console.ReadLine()
    End Sub
End Module
```

Скомпилируем данное приложение (файл `MyModule.vb`):

```
vbc.exe /reference:MyLibrary.dll MyModule.vb
```

Параметр компиляции `/reference:MyLibrary.dll` определяет ссылку на сборку, содержащую наш класс.

Продемонстрируем возможности наследования классов, размещенных в разных сборках. Создадим приложение, которое содержит класс, унаследованный от `CExampleClass`:

```
Imports System
Module MyModule
    Public Class CDescentClass
        Inherits CExampleClass
        Public Sub Hello()
```

```
            Console.WriteLine("Hello!")
        End Sub
    End Class
    Sub Main()
        Dim A As New CDescentClass()
        A.SetField(100)
        A.Hello()
        Console.ReadLine()
    End Sub
End Module
```

Скомпилируем данное приложение (пусть оно содержится в файле `MyModule.vb`):

```
vbc.exe /reference:MyLibrary.dll MyModule.vb
```

Результат работы приложения – 2 строки: `"New field value is 100"` и `"Hello!"`.

Продемонстрируем возможности межъязыкового взаимодействия. Создадим класс, практически аналогичный `CExampleClass`, но написанный на C#:

```
public class CexampleClass {
    int fX;
    public void SetField(int NewX) {
        fX = NewX;
        System.Console.WriteLine("New field is {0}", fX);
    }
}
```

Скомпилируем класс в библиотеку, использовав компилятор командной строки:

```
csc.exe /target:library MyLibrary.cs
```

Перекомпилировав последний вариант файла `MyModule.vb`, мы можем видеть, что платформа .NET с легкостью поддерживает использование и наследование от класса, написанного на другом языке программирования.

Один или несколько исходных файлов на VB .NET можно скомпилировать в отдельные модули. *Модуль*[1] представляет собой фрагмент скомпилированного кода, который может использоваться сборками. Для компиляции файла в модуль используется ключ `/target:module`:

```
vbc.exe /target:module file1.vb
```

Полученный модуль будет иметь расширение `netmodule` (`file1.netmodule`). Для того чтобы использовать модули при создании сборки, используются ключ `/addmodule`:

```
vbc.exe /addmodule:file1.netmodule resultfile.vb
```

Модули не являются полноценными сборками. Это следует учитывать при использовании в модулях директив ограничения видимости.

[1] Не путайте понятие *модуль*, используемое в данном контексте, с понятием модуль, как пользовательский тип VB.NET.

30. Работа с директориями и файлами.

Пространство имен `System.IO` содержит четыре класса, предназначенные для работы с физическими файлами и каталогами на диске. Классы `Directory` и `File` выполняют операции в файловой системе при помощи разделяемых членов, классы `DirectoryInfo` и `FileInfo` обладают схожими возможности, однако для работы требуется создании соответствующих объектов.

Рассмотрим работу с классами `DirectoryInfo` и `FileInfo`. Данные классы являются наследниками абстрактного класса `FileSystemInfo`. Этот класс содержит следующие основные члены, перечисленные в табл. 10.

Таблица 10

Свойства и методы класса `FileSystemInfo`

Имя элемента	Описание
`Attributes`	Свойство позволяет получить или установить атрибуты объекта файловой системы (тип – перечисление `FileAttributes`)
`CreationTime`	Свойство для чтения или установки времени создания объекта файловой системы
`Exists`	Свойство для чтения, проверка существования объекта файловой системы
`Extension`	Свойство для чтения, расширение файла
`FullName`	Свойство для чтения, полное имя объекта файловой системы
`LastAccessTime`	Свойство обеспечивает чтение или установку времени последнего доступа для объекта файловой системы
`LastWriteTime`	Свойство обеспечивает чтение или установку времени последней записи для объекта файловой системы
`Name`	Свойство для чтения, которое возвращает имя файла или каталога
`Delete()`	Метод удаляет объект файловой системы
`Refresh()`	Метод обновляет информацию об объекте файловой системы

Конструктор класса `DirectoryInfo` принимает в качестве параметра строку с именем того каталога, с которым будет производиться работа. Для указания текущего каталога используется точка (строка `"."`). При попытке работать с данными несуществующего каталога генерируется исключительная ситуация. Работу с методами и свойствами класса продемонстрируем в следующем примере:

```
Imports System
Imports System.IO

Class MainClass
    Public Shared Sub Main()
        'Создали объект для директории
        Dim dir As DirectoryInfo = _
                        New DirectoryInfo("C:\Temp\Diagrams")
        'Выводим некоторые свойства директории
        Console.WriteLine("Full Name: {0}", dir.FullName)
        Console.WriteLine("Name: {0}", dir.Name)
        Console.WriteLine("Parent: {0}", dir.Parent)
        Console.WriteLine("Root: {0}", dir.Root)
```

```
            Console.WriteLine("Creation: {0}", dir.CreationTime)
            'На экран будет выведена следующая информация:
            'Full Name: C:\Temp\Diagrams
            'Name: Diagrams
            'Parent: Temp
            'Root: C:\
            'Creation: 01.01.1601 3:00:00

            'Создаем новую поддиректорию в нашей
            dir.CreateSubdirectory("Dir2")
            'Создаем еще одну новую поддиректорию
            dir.CreateSubdirectory("Dir2\SubDir2")
            'Получаем массив объектов, описывающих поддиректории
            Dim subdirs As DirectoryInfo() = dir.GetDirectories
            'Получаем массив объектов, описывающих файлы
            Dim files As FileInfo() = dir.GetFiles
            'Можно использовать маску файлов
            Dim files1 As FileInfo() = dir.GetFiles("*.er1")
            'Перемещаем директорию на новое место
            dir.MoveTo("C:\T")
            'Объявляем и создаем директорию
            Dim new_dir As DirectoryInfo = _
                                       New DirectoryInfo("C:\Temp\D")
            new_dir.Create
        End Sub
    End Class
```

Класс `FileInfo` описывает файл на жестком диске и позволяет производить операции с этим файлом. Наиболее важные члены класса представлены в таблице 11.

Таблица 11

Свойства и методы класса `FileInfo`

Имя элемента	Описание
`AppendText()`	Создает объект `StreamWriter` для добавления текста к файлу
`CopyTo()`	Копирует существующий файл в новый
`Create()`	Создает файл и возвращает объект `FileStream` для работы с файлом
`CreateText()`	Создает объект `StreamWriter` для записи текста в новый файл
`Directory`	Свойство для чтения, каталог файла
`DirectoryName`	Свойство для чтения, полный путь к файлу
`Length`	Свойство для чтения, размер файла в байтах
`Open()`	Открывает файл с указанными правами доступа на чтение, запись или совместное использование
`OpenRead()`	Создает объект `FileStream`, доступный только для чтения
`OpenText()`	Создает объект `StreamReader` для чтения информации из существующего текстового файла
`OpenWrite()`	Создает объект `FileStream`, доступный для чтения и записи

Примеры, иллюстрирующие работу с объектами-потоками, будут рассмотрены ниже. Рассмотрим пример, показывающий создание и удаление файла:

```
Imports System
Imports System.IO
Class MainClass
    Public Shared Sub Main()
        'Создаем объект
        Dim file As FileInfo = New FileInfo("C:\Test.txt")
        'Создаем файл (с потоком делать пока ничего не будем)
        Dim fs As FileStream = file.Create
        'Выводим информацию
        Console.WriteLine("Full Name: {0}", file.FullName)
        Console.WriteLine("Attributes: {0}", _
                                    file.Attributes.ToString)
        Console.WriteLine("Press any key to delete file")
        Console.Read
        'Закрываем поток, удаляем файл
        fs.Close
        file.Delete
    End Sub
End Class
```

Для работы с файлом можно использовать метод `FileInfo.Open()`, который обладает большим числом возможностей, чем метод `Create()`. Рассмотрим перегруженную версию метода `Open()`, которая содержит три параметра. Первый параметр определяет режим запроса на открытие файла. Для него используются значения из перечисления `FileMode`.

Таблица 12

Элементы перечисления `FileMode`

Значение	Описание
`Append`	Открывает файл, если он существует, и ищет конец файла. Если файл не существует, то он создается. Этот режим может использоваться только с доступом `FileAccess.Write`
`Create`	Указывает на создание нового файла. Если файл существует, он будет перезаписан
`CreateNew`	Указывает на создание нового файла. Если файл существует, генерирует исключение `IOException`
`Open`	Операционная система должна открыть существующий файл
`OpenOrCreate`	Операционная система должна открыть существующий файл или создать новый, если файл не существует
`Truncate`	Система должна открыть существующий файл и обрезать его до нулевой длины

Второй параметр метода `Open()` определяет тип доступа к файлу как к потоку байтов. Для него используются элементы перечисления `FileAccess`.

Таблица 13

Элементы перечисления `FileAccess`

Значение	Описание
`Read`	Файл будет открыт только для чтения
`ReadWrite`	Файл будет открыт и для чтения, и для записи
`Write`	Файл открывается только для записи, то есть добавления данных

Третий параметр определяет возможность совместного доступа к открытому файлу и представлен значениями перечисления FileShare.

Таблица 14

Элементы перечисления FileShare

Значение	Описание
None	Совместное использование запрещено, на любой запрос на открытие файла будет возвращено сообщение об ошибке
Read	Файл могут открыть и другие пользователи, но только для чтения
ReadWrite	Другие пользователи могут открыть файл для чтения и записи
Write	Файл может быть открыт другими пользователями для записи

Вот пример кода, использующего метод Open():

```
'Файл создается (или открывается) для чтения и записи,
'без возможности совместного использования
Dim file As FileInfo = New FileInfo("C:\Test.txt")
Dim fs As FileStream = file.Open(FileMode.OpenOrCreate, _
                                 FileAccess.ReadWrite, _
                                 FileShare.None)
```

31. Ввод и вывод в файлы и потоки.

Для поддержки операций, связанных с вводом и выводом информации, библиотека классов платформы .NET предоставляет пространство имен System.IO. Основное понятие, связанное с элементами данного пространства имен, – это поток. *Поток* – абстрактное представление данных в виде последовательности байт. Потоки (в отличие от файлов) могут быть ассоциированы с файлами на диске, памятью, сетью. В пространстве имен System.IO поток представлен абстрактным классом Stream. От данного абстрактного класса порождены классы System.IO.FileStream (работа с файлами как с потоками), System.IO.MemoryStream (поток в памяти), System.Net.Sockets.NetworkStream (работа с сокетами как с потоками), System.Security.Cryptography.CryptoStream (потоки зашифрованных данных).

Рассмотрим основные методы и свойства класса Stream. Свойства для чтения CanRead, CanWrite и CanSeek определяют, поддерживает ли поток чтение, запись и поиск. Если поток поддерживает поиск, перемещаться по потоку можно при помощи метода Seek(). На текущую позицию в потоке указывает свойство Position (нумерация с нуля). Свойство Length возвращает длину потока, которая может быть установлена при помощи метода SetLength(). Методы Read() и ReadByte(), Write() и WriteByte() служат для чтения и записи блока байт или одиночного байта. Метод Flush() записывает данные из буфера в связанный с потоком источник данных. При помощи метода Close() поток закрывается и все связанные с ним ресурсы освобождаются.

Класс Stream вводит поддержку асинхронного ввода/вывода. Для этого служат методы BeginRead() и BeginWrite(). Уведомление о завершении асинхронной операции возможно двумя способами: или при помощи делегата тип System.AsyncCallback, передаваемого как параметр методов BeginRead() и BeginWrite(), или при помощи вызова методов EndRead() и EndWrite(),

которые приостанавливают текущий поток управления до окончания асинхронной операции.

Использование методов и свойств класса `Stream` продемонстрируем во фрагменте кода с классом `FileStream`. Объект класса `FileStream` возвращается некоторыми методами классов `FileInfo` и `File`, кроме этого, данный объект можно создать при помощи конструктора с параметрами, включающими имя файла и режимы доступа к файлу.

```
Imports System
Imports System.IO
Class MainClass
    Public Shared Sub Main()
        'Создаем файл test.dat в текущем каталоге
        Dim fs As FileStream = New FileStream("test.dat", _
                                              FileMode.OpenOrCreate, _
                                              FileAccess.ReadWrite)
        'В цикле записываем туда 100 байт
        Dim i As Byte
        For i = 0 To 100
            fs.WriteByte(i)
        Next

        'Мы можем записывать информацию из массива байт
        Dim info As Byte() = {1, 2, 3, 4, 5, 6, 7, 8, 9}
        'Первый параметр - массив, второй - смещение в нем,
        'третий - количество записываемых байт
        fs.Write(info, 2, 4)

        'Возвращаемся на начало файла
        fs.Position = 0

        'Читаем все байты и выводим на экран
        While fs.Position <= fs.Length - 1
            Console.Write(fs.ReadByte)
        End While

        'Закрываем поток (и файл), освобождая ресурсы
        fs.Close
    End Sub
End Class
```

Класс `MemoryStream` предоставляет возможность организовать поток в оперативной памяти. Свойство `Capacity` этого класса позволяет получить или установить количество байтов, выделенных под поток. Метод `ToArray()` записывает все содержимое потока в массив байт. Метод `WriteTo()` переносит содержимое потока в памяти в любой другой поток, производный от класса `Stream`.

Классы-потоки представляют поток как последовательность неформатированных байт. Однако в большинстве приложений удобнее читать и записывать в поток данные примитивных типов или строк. Библиотека классов .NET

Framework содержит набор парных классов вида XXXReader/XXXWriter, которые инкапсулируют поток и предоставляют к нему высокоуровневый доступ.

Классы BinaryReader и BinaryWriter позволяют при помощи своих методов читать и записывать в поток данные примитивных типов, строк и массивов байт или символов. Вся информация записывается в поток как последовательность байт. Рассмотрим работу с данными классами на следующем примере. Пусть имеется класс, который хранит информацию о студенте:

```
Class Student
    Public Name As String
    Public Age As Integer
    Public MeanScore As Double
End Class
```

Методы, которые осуществляют запись и чтение объекта этого класса в поток в виде последовательности байт, могут иметь следующий вид:

```
Sub SaveToStream(ByVal stm As Stream, ByVal s As Student)
  'Конструктор позволяет "обернуть" BinaryWriter вокруг потока
    Dim bw As BinaryWriter = New BinaryWriter(stm)
    'BinaryWriter содержит 18 версий метода Write()
    bw.Write(s.Name)
    bw.Write(s.Age)
    bw.Write(s.MeanScore)
    'Убеждаемся, что буфер BinaryWriter пуст
    bw.Flush
End Sub

Sub ReadFromStream(ByVal stm As Stream, ByVal s As Student)
    Dim br As BinaryReader = New BinaryReader(stm)
    'Для чтения каждого примитивного типа есть свой метод
    s.Name = br.ReadString
    s.Age = br.ReadInt32
    s.MeanScore = br.ReadDouble
End Sub
```

Абстрактные классы TextReader и TextWriter позволяю читать и записывать данные в поток как последовательность символов. От этих классов наследуются классы StreamReader и StreamWriter. Перепишем методы для сохранения данных класса Student с использованием StreamReader и StreamWriter:

```
Sub SaveToStream(ByVal stm As Stream, ByVal s As Student)
    Dim sw As StreamWriter = New StreamWriter(stm)
    'Запись напоминает вывод на консоль
    sw.WriteLine(s.Name)
    sw.WriteLine(s.Age)
    sw.WriteLine(s.MeanScore)
    sw.Flush
End Sub

Sub ReadFromStream(ByVal stm As Stream, ByVal s As Student)
    Dim sr As StreamReader = New StreamReader(stm)
```

```
        'Читаем данные как строки, их требуется преобразовать
        s.Name = sr.ReadLine
        s.Age = Int32.Parse(sr.ReadLine)
        s.MeanScore = Double.Parse(sr.ReadLine)
    End Sub
```

Классы `StringReader` и `StringWriter` – это наследники классов `TextReader` и `TextWriter`, которые представляют доступ к строке или к объекту класса `StringBuilder` как к потоку. Это может оказаться полезным, если текстовая информация добавляется в специальный буфер в оперативной информации. Работу с данными классами иллюстрирует следующий пример:

```
Dim sw As StringWriter = New StringWriter
'Пишем информацию в поток
sw.WriteLine("Hello!")
sw.WriteLine("This is an example...")
sw.Close

'Выводим всю информацию
Console.WriteLine(sw.ToString)

'Создадим объект StringReader на основе строки
Dim s As String = "Big" & Microsoft.VisualBasic.Chr(10) & _
          "Big string" & Microsoft.VisualBasic.Chr(10) & " 10"
Dim sr As StringReader = New StringReader(s)

'Последовательно читаем "кусочки" строки
Dim inpt As String
While Not ((inpt = sr.ReadLine) Is Nothing)
    Console.WriteLine(inpt)
End While
sr.Close
```

32. 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. Кроме этого, некоторые интерфейсы из описываемого пространства имен удобны для использования при сортировках, перечислении элементов массивов и структур. В данном параграфе будут описаны базовые приемы работы с элементами из пространства имен `System.Collections`.

Рассмотрим интерфейсы из пространства имен `System.Collections` и дадим их краткое описание.

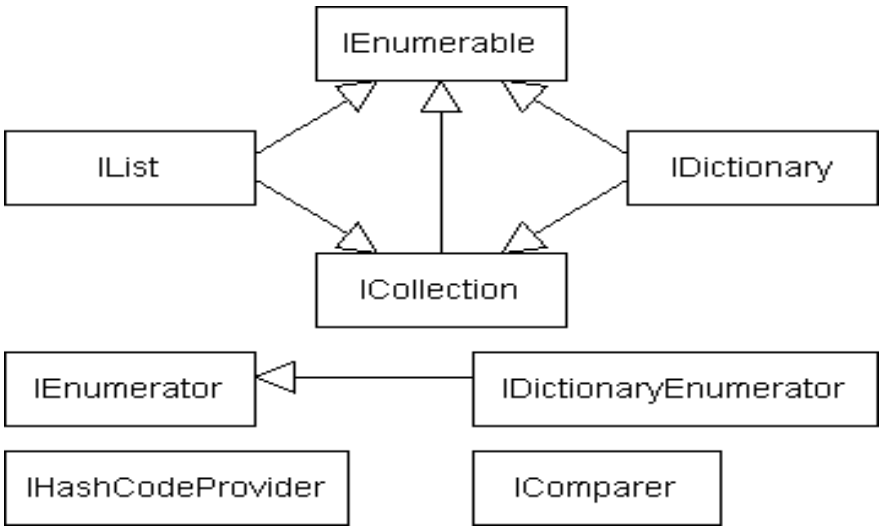

Рис. 7. Интерфейсы из `System.Collections`

Интерфейс `IEnumerator` представляет поддержку для набора объектов возможности перебора в цикле `For Each`.

Интерфейс `ICollection` служит для описания некоторого набора объектов. Этот интерфейс имеет свойство для чтения количества элементов в наборе (`Count`), а также метод для копирования набора в массив `Array` (`CopyTo`).

Интерфейс `IList` описывает набор данных, которые проецируются на массив. Приведем описание данного интерфейса, назначение его методов и свойств очевидно:

```
Interface IList
    Inherits ICollection, IEnumerable

    ReadOnly Property IsFixedSize() As Boolean
    ReadOnly Property IsReadOnly() As Boolean
    'Основное свойство
    Default Property Item(ByVal index As Integer) As Object

    Function Add(ByVal value As Object) As Integer
    Sub Clear()
    Function Contains(ByVal value As Object) As Boolean
    Function IndexOf(ByVal value As Object) As Integer
    Sub Insert(ByVal index As Integer, ByVal value As Object)
    Sub Remove(ByVal value As Object)
    Sub RemoveAt(ByVal index As Integer)
End Interface
```

Интерфейс `IDictionary` служит для описания коллекций, представляющих словари, то есть таких наборов данных, где доступ осуществляется с использованием произвольного ключа.

```
Interface IDictionary
    Inherits ICollection, IEnumerable

    ReadOnly Property IsFixedSize() As Boolean
    ReadOnly Property IsReadOnly() As Boolean
```

```
        ReadOnly Property Keys() As ICollection
        ReadOnly Property Values() As ICollection
        'Основное свойство
        Default Property Item(ByVal key As Object) As Object

        Sub Add(ByVal key As Object, ByVal value As Object)
        Sub Clear()
        Function Contains(ByVal key As Object) As Boolean
        Function GetEnumerator() As IDictionaryEnumerator
        Sub Remove(ByVal key As Object)
    End Interface
```

Интерфейс `IEnumerator` предназначен для организации перебора элементов коллекции, а интерфейс `IDictionaryEnumerator` кроме перебора элементов позволяет обратиться к ключу и значению текущего элемента. Назначение интерфейсов `IHashCodeProvider` и `IComparer` разъясняется далее по тексту.

На протяжении параграфа будем использовать следующий вспомогательный класс, хранящий информацию о студенте:

```
    Class CStudent
        Public name As String
        Public course As Integer
        Public ball As Double
        Public Sub New(name As String, course As Integer, _
                                       ball As Double)
            Me.name = name
            Me.course = course
            Me.ball = ball
        End Sub
        Public Overloads Overrides Function ToString() As String
         Return String.Format("Name = {0,10}; Course = {1,3};" & _
                              "Ball = {2,4}", name, course, ball)
        End Function
    End Class
```

Имея в своем распоряжении класс `CStudent`, мы можем 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 следующим образом:

```
    Interface IComparable
        Function CompareTo(ByVal o As Object) As Integer
    End Interface
```

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

Пусть мы хотим сортировать массив студентов по убыванию среднего балла. Внесем поддержку `IComparable` в класс `CStudent`:

```
    Class CStudent
```

```
    Implements IComparable
    . . .
    Public Function CompareTo(o As Object) As Integer _
                                Implements IComparable.CompareTo
        Dim tmp As CStudent = CType(o, CStudent)
        If (ball > tmp.ball) Then Return 1
        If (ball < tmp.ball) Then Return -1
        Return 0
    End Function
End Class
```

Объекты класса `CStudent` теперь можно использовать таким образом:

```
Module Main
    Sub Main()
     Dim Group() As CStudent ={New CStudent("Alex", 1, 7.0), _
                               New CStudent("Ivan", 1, 9.0), _
                               New CStudent("Anna", 1, 8.0), _
                               New CStudent("Peter", 1, 5.0)}

        Console.WriteLine("Unsorted group")
        Dim s As CStudent
        For Each s In Group
            Console.WriteLine(s)
        Next

        Array.Sort(Group)
        Array.Reverse(Group)
        Console.WriteLine("Sorted group")
        For Each s In Group
            Console.WriteLine(s)
        Next
    End Sub
End Module
```

Данная программа выводит следующее:

```
Unsorted group
Name =       Alex; Course =   1; Ball =    7
Name =       Ivan; Course =   1; Ball =    9
Name =       Anna; Course =   1; Ball =    8
Name =      Peter; Course =   1; Ball =    5
Sorted group
Name =       Ivan; Course =   1; Ball =    9
Name =       Anna; Course =   1; Ball =    8
Name =       Alex; Course =   1; Ball =    7
Name =      Peter; Course =   1; Ball =    5
```

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

```
Interface IComparer
   Function Compare(ByVal o1 As Object, ByVal o2 As Object) _
                                                    As Integer
End Interface
```

Использование IComparer позволяет осуществить сортировку по нескольким критериям. Для этого каждый критерий описывается вспомогательным классом, реализующим IComparer и определенный способ сравнения.

Возьмем за основу первоначальный вариант класса CStudent и опишем два вспомогательных класса следующим образом:

```
Class SortStudentsByBall
    Implements IComparer
    Public Function Compare(o1 As Object, o2 As Object) _
                        As Integer Implements IComparer.Compare
        Dim t1 As CStudent = CType(o1, CStudent)
        Dim t2 As CStudent = CType(o2, CStudent)
        If t1.ball > t2.ball Then Return 1
        If t1.ball < t2.ball Then Return -1
        Return 0
    End Function
End Class

Class SortStudentsByName
    Implements IComparer
    Public Function Compare(o1 As Object, o2 As Object) _
                        As Integer Implements IComparer.Compare
        Dim t1 As CStudent = CType(o1, CStudent)
        Dim t2 As CStudent = CType(o2, CStudent)
        Return String.Compare(t1.name, t2.name)
    End Function
End Class
```

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(Group, New SortStudentsByBall())

'Сортировка массива студентов по имени
Array.Sort(Group, New SortStudentsByName())
```

Язык VB.NET имеет цикл для перебора элементов коллекций – For Each. Пусть мы создали класс, хранящий некий набор элементов:

```
Class Room
    Private H() As CStudent

    'Создаем "комнату" в конструкторе. Это не лучшее решение!
    Sub New()
        ReDim H(2)
        H(0) = New CStudent("Alex", 1, 7.0)
        H(1) = New CStudent("Ivan", 1, 9.0)
        H(2) = New CStudent("Peter", 1, 5.0)
    End Sub
End Class
```

Было бы удобно для перебора элементов в объектах нашего класса Room использовать цикл For Each. Для этого необходимо организовать в классе поддержку интерфейса IEnumerable. Интерфейс IEnumerable устроен очень просто. Он содержит единственный метод, задача которого – вернуть интерфейс IEnumerator.

```
Interface IEnumerable
    Function GetEnumerator() As IEnumerator
End Interface
```

В свою очередь, интерфейс IEnumerator имеет следующее описание:

```
Interface IEnumerator
    'Передвинуть курсор данных на следующую позицию
    Function MoveNext() As Boolean
    'Свойство для чтения - текущий элемент
    ReadOnly Property Current As Object
    'Установить курсор перед началом набора данных
    Sub Reset()
End Interface
```

Очень часто интерфейсы IEnumerable и IEnumerator поддерживает один и тот же класс. Добавим поддержку данных интерфейсов в класс Room:

```
Class Room
    Implements IEnumerable, IEnumerator
    Private H() As CStudent
    'Внутреннее поле для курсора данных
    Private pos As Integer = -1

    Sub New()
        'Конструктор был описан выше
    End Sub

    Function GetEnumerator() As IEnumerator _
                            Implements IEnumerable.GetEnumerator
        Return CType(Me, IEnumerator)
    End Function

    Function MoveNext() As Boolean _
                                Implements IEnumerator.MoveNext
        If (pos < H.Length - 1) Then
            pos += 1
            Return True
        Else
            Return False
        End If
    End Function

    Sub Reset() Implements IEnumerator.Reset
        pos = -1
    End Sub

    ReadOnly Property Current As Object _
```

```
                                        Implements IEnumerator.Current
            Get
                Return H(pos)
            End Get
        End Property
    End Class
```

Теперь для перебора элементов Room можно использовать такой код:

```
Dim R_1003 As New Room()
Dim S As CStudent
For Each S In R_1003
    Console.WriteLine(S)
Next
```

В табл. 15 перечислен набор основных классов, размещенных в пространстве имен System.Collections.

Таблица 15

Классы из System.Collections

Имя класса	Класс представляет…	Класс реализует интерфейсы…
ArrayList	Массив изменяемого размера с целочисленным индексом	IList, ICollection, IEnumerable,ICloneable
BitArray	Компактный массив бит с поддержкой битовых операций	ICollection, IEnumerable,ICloneable
CaseInsensitiveComparer	Класс с реализацией интерфейса IComparer, которая игнорирует регистр в случае сравнения строк	IComparer
CaseInsensitive-HashCodeProvider	Класс с методом генерации хэш-кодов, игнорирующим регистр в случае генерации кодов для строк	IHashCodeProvider
CollectionBase	Абстрактный базовый класс для построения пользовательских типизированных коллекций	IList, ICollection, IEnumerable
Comparer	Класс с реализацией интерфейса IComparer, основанной на вызове методов CompareTo() сравниваемых объектов	IComparer
DictionaryBase	Абстрактный класс для построения пользовательских типизированных хэш-таблиц	IDictionary, ICollection, IEnumerable
Hashtable	Таблицу объектных пар «ключ-значение», оптимизированную для быстрого поиска по ключу	IDictionary, ICollection, IEnumerable, ICloneable, IDeserializationCallback, ISerializable
Queue	Очередь (FIFO)	ICollection, IEnumerable,ICloneable
ReadOnlyCollection-Base	Абстрактный класс для построения пользовательских типизированных коллекций с доступом только для чтения	ICollection, IEnumerable

SortedList	Таблицу 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 просто:

```
Dim list As New ArrayList()
list.Add("John")
list.Add("Paul")
list.Add("George")
list.Add("Ringo")
```

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 основное индексированное свойство Item. Индексы – целые числа, начинающиеся с нуля. С помощью свойства Item можно изменить значения существующего элемента[1]:

```
Dim st As String = CStr(list(0))
list(0) = 999
```

Свойство Count позволяет узнать количество элементов динамического массива. Для перебора элементов массива можно использовать цикл For Each:

```
For I As Integer = 0 To list.Count-1
    Console.WriteLine(list(i))
Next
'Можно перебирать элементы массива и так
'(правда, это примерно на 25% медленнее)
For Each s As String In List
    Console.WriteLine(s)
Next
```

[1] Попытка работать с несуществующим элементом генерирует исключение ArgumentOutOfRangeException.

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 заданной емкости
Dim Lst As New ArrayList(1000)
'Заполняем его
For I As Integer = 0 To 999
    Lst.Add(I)
Next

'Удаляем первые 500 элементов
Lst.RemoveRange(0, 500)

'Емкость массива по-прежнему 1000
Console.WriteLine(Lst.Capacity)

'"Обрежем" массив
Lst.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:

```
Dim table As New Hashtable()
table.Add("Sunday",    "Воскресенье")
table.Add("Monday",    "Понедельник")
table.Add("Tuesday",   "Вторник")
table.Add("Wednesday", "Среда")
table.Add("Thursday",  "Четверг")
table.Add("Friday",    "Пятница")
table.Add("Saturday",  "Суббота")
```

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

```
Dim s As String
s = CStr(table.Item("Friday"))
```

Элементы могут быть добавлены в хэш-таблицу при помощи свойства Item:

```
table("Sunday") = "Воскресенье"
table("Monday") = "Понедельник"
```

Если указываемый при добавлении ключ уже существует в таблице, метод Add генерирует исключительную ситуацию. При использовании свойства Item старое значение просто заменяется новым.

Отдельный элемент хэш-таблицы – это переменная структуры DictionaryEntry. Эта структура имеет свойства для доступа к ключу (Key) и значению (Value) элемента. Класс Hashtable поддерживает интерфейс IEnumerable, что делает возможным перебор элементов с использованием For Each:

```
For Each entry As DictionaryEntry In table
    Console.WriteLine("Key = {0}, Value = {1}", _
                                        entry.Key, entry.Value)
Next
```

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.

На скорость работы с элементам хэш-таблицы влияют два фактора: размер таблицы и уникальность хэш-кодов, продуцируемых ключами таблицы. 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 примерно 58% (72*0.8)
Dim tbl As 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;
- создать тип, реализующий интерфейс IHashCodeProvider и передавать экземпляр такого типа конструктору таблицы. Интерфейс IHashCodeProvider содержит метод GetHashCode, генерирующий хэши по входным объектам.

Для сравнения ключей таблица по умолчанию применяет метод Equals объектов. Если сравнение реализовано не эффективно, требуется переписать метод Equals или передать конструктору таблицы экземпляр класса, реализующего интерфейс IComparer.

В пространстве имен System.Collections определены два класса, которые можно использовать как основу для создания собственных типизированных коллекций-списков и коллекций-словарей. Это классы DictionaryBase и CollectionBase. Каждый из этих классов предоставляет ряд виртуальных методов, вызываемых при модификации коллекции. Например, OnClean вызывается, когда коллекция очищается; OnInsert – перед добавлением элемента; OnInsertComplete – после добавления; OnRemove – перед удалением элемента и т. п. Переопределяя эти методы можно создать дополнительные проверки и вызвать исключение в случае ошибок типизации. Например, следующий класс представляет собой коллекцию, которая допускает хранение только строк из четырех символов:

```
Imports System
Imports System.Collections

Class MyStringCollection
    Inherits CollectionBase

    'Этот метод будет использоваться для добавления элементов
    'Он просто вызывает метод Add() внутреннего списка List
    'Метод не производит проверки типов!
    Sub Add(value As Object)
        List.Add(value)
    End Sub

    'Перекрываем виртуальный метод OnInsert()
    'В нем выполняем проверку
    Protected Overrides Sub OnInsert(index As Integer, _
                                     value As Object)
        If (TypeOf value Is String) AndAlso _
           (CStr(value).Length = 4)
            MyBase.OnInsert(index, value)
        Else
          Throw New ArgumentException("Length 4 allowed only")
        End If
    End Sub

    'Метод для получения элементов
    'В нем выполняем приведение типов
    Function GetItem(index As Integer) As String
        Return CStr(List(index))
    End Function
End Class

Module MainModule
    Sub Main()
```

```
            Dim msc As New MyStringCollection
            msc.Add("Alex")
            msc.Add("QWER")
            Dim s As String
            s = msc.GetItem(1)
            Console.WriteLine(s)

            'Любая из следующих строк генерирует исключение
            msc.Add("asldalda")
            msc.Add(5)
        End Sub
    End Module
```

Для иллюстрации использования коллекций приведем пример приложения, подсчитывающего уникальные слова в текстовом файле. Текстовый файл передается консольному приложению в качестве параметра.

```
    Imports System
    Imports System.IO
    Imports System.Collections

    Module Module1
        Sub Main(ByVal args As String())
            'Проверка параметров командной строки
            If args.Length = 0 Then
                Console.WriteLine("Ошибка: нет имени файла")
                Return
            End If

            'Создаем объект для доступа к строкам файла
            'и объект для хранения уникальных слов и их количества
            Dim reader As StreamReader = Nothing
            Dim table As New Hashtable
            Try
                reader = New StreamReader(args(0))

                'Перебираем все строки файла
                Dim line As String = reader.ReadLine
                While Not (line Is Nothing)
                    'Этот массив хранит слова строки
                    Dim words As String() = GetWords(line)

                    'Перебираем все слова
                    For Each word As String In words
                        Dim iword As String = word.ToLower
                        'Если слово уже есть, счетчик + 1
                        If table.ContainsKey(iword) Then
                            table(iword) = CInt(table(iword)) + 1
                        Else 'Иначе помещаем слово в таблицу
                            table(iword) = 1
                        End If
                    Next
                    line = reader.ReadLine
```

```
            End While

            'Сортируем хэш-таблицу, используя класс SortedList
            Dim list As SortedList = New SortedList(table)

            'Выводим результат работы
         Console.WriteLine("Уникальных слов {0}", table.Count)
            For Each ent As DictionaryEntry In list
            Console.WriteLine("{0} ({1})", ent.Key, ent.Value)
            Next
        Catch e As Exception
            Console.WriteLine(e.Message)
        Finally
            If Not (reader Is Nothing) Then reader.Close()
        End Try
    End Sub

    'Функция для выделения слов в строке
    Function GetWords(line As String) As String()
        'Создадим ArrayList для промежуточных результатов
        Dim al As New ArrayList
        Dim i As Integer = 0

        'Разбираем строку на слова
        Dim chars As Char() = line.ToCharArray
        Dim word As String = GetNextWord(line, chars, i)
        While Not (word Is Nothing)
            al.Add(word)
            word = GetNextWord(line, chars, i)
        End While

    'Возвращаем статический массив, эквивалентный динамическому
        Dim words(al.Count - 1) As String
        al.CopyTo(words)
        Return words
    End Function

    'Метод для получения следующего слова в строке
    Function GetNextWord(line As String, chars As Char(), _
                         ByRef i As Integer) As String

        'Находим начало следующего слова
        While i < chars.Length AndAlso _
              Not Char.IsLetterOrDigit(chars(i))
            i += 1
        End While
        If i = chars.Length Then Return Nothing
        Dim start As Integer = i

        'Находим конец слова
        While i < chars.Length AndAlso _
              Char.IsLetterOrDigit(chars(i))
```

```
            i += 1
        End While

        'Возвращаем найденное слово
        Return line.Substring(start, i - start)
    End Function
End Module
```

Наряду с пространством имен `System.Collections`, .NET Framework предоставляет пространство имен `System.Collections.Specialized`. Оно содержит классы коллекций, которые подходят для решения специальных задач.

Таблица 16

Классы и структуры из `System.Collections.Specialized`

Имя класса (структуры)	**Для чего служит**	**Реализует интерфейсы**
`BitVector32`	Структура фиксированного размера (4 байта) для хранения массива булевых значений	
`CollectionsUtil`	Класс, разделяемые методы которого позволяют создавать коллекции, в которых игнорируется регистр строк	
`HybridDictionary`	Реализует интерфейс `IDictionary` с использованием `ListDictionary` для малых коллекций; автоматический переключается на использование `Hashtable`, когда размер коллекции увеличивается	`IDictionary, ICollection, IEnumerable`
`ListDictionary`	Реализует интерфейс `IDictionary` с использованием односвязного списка. Рекомендуется для коллекций с количеством элементов меньше 10	`IDictionary, ICollection, IEnumerable`
`NameObjectCollectionBase`	Абстрактный базовый класс для сортированных таблиц «ключ-значение», ключом которых является строка (доступ к элементу – по ключу или по индексу)	
`NameValueCollection`	Класс для сортированных таблиц «ключ-значение», ключом и значением в которых является строка (доступ к элементу – по ключу или по индексу)	
`StringCollection`	Представляет коллекцию строк	
`StringDictionary`	Класс для хэш-таблиц, ключом которых является строка	`IEnumerable`
`StringEnumerator`	Итератор для `StringCollection`	

Рассмотрим некоторые из классов, упомянутых в таблице. Класс `StringCollection` представляет реализацию класса `ArrayList`, которая может хранить только строки. Это обеспечивает дополнительный контроль при помещении объекта в коллекцию и избавляет от необходимости выполнять приведения типов. Класс `StringDictionary` представляет реализацию хэш-таблицы, в ко-

торой ключ и значение всегда имеют тип String. Кроме этого, ключ таблицы не зависит от регистра символов.

```
Dim sd As New StringDictionary()
sd("Leonid") = "Minchenko"
sd("Alexey") = "Volosevich"
Console.WriteLine(sd("LEONID"))    'выводит Minchenko
```

Классы коллекций, предоставляемые .NET Framework, подходят для решения большинства типичных задач, встречающихся при написании приложений. Если же стандартных классов не достаточно, программист может воспользоваться сторонними библиотеками или разработать собственный класс для некой структуры данных.

33. Асинхронный вызов методов.

Платформа .NET содержит средства для поддержки асинхронного вызова методов. При *асинхронном вызове* поток выполнения разделяется на две части: в одной выполняется метод, а в другой – нормальный процесс программы. Асинхронный вызов может служить (в некоторых случаях) альтернативой использованию многопоточности явным образом.

Асинхронный вызов метода всегда выполняется посредством объекта некоторого делегата. Любой такой объект содержит два специальных метода для асинхронных вызовов – `BeginInvoke()` и `EndInvoke()`. Данные методы генерируются во время выполнения программы (как и метод `Invoke()`), так как их сигнатура зависит от делегата.

Метод `BeginInvoke()` обеспечивает асинхронный запуск. Данный метод имеет два дополнительных параметра по сравнению с описанными в делегате. Назначение первого дополнительного параметра – передать делегат, указывающий на функцию обратного вызова, выполняемую после работы асинхронного метода (*функция завершения*). Второй дополнительный параметр – это объект, при помощи которого функции завершения может быть передана некоторая информация. Метод `BeginInvoke()` возвращает объект, реализующий интерфейс `IAsyncResult`, при помощи этого объекта становится возможным различать асинхронные вызовы одного и того же метода.

Приведем описание интерфейса `IAsyncResult`:

```
Interface IAsyncResult
    ReadOnly Property AsyncState() As Object
    ReadOnly Property AsyncWaitHandle() As WaitHandle
    ReadOnly Property CompletedSynchronously() As Boolean
    ReadOnly Property IsCompleted() As Boolean
End Interface
```

Поле `IsCompleted` позволяет узнать, завершилась ли работа асинхронного метода. В поле `AsyncWaitHandle` храниться объект типа `WaitHandle`. Программист может вызывать методы класса `WaitHandle`, такие как `WaitOne()`, `WaitAny()`, `WaitAll()`, для контроля над потоком выполнения асинхронного делегата. Объект `AsyncState` хранит последний параметр, указанный при вызове `BeginInvoke()`.

Делегат для функции завершения описан следующим образом:

```
Delegate Sub AsyncCallback(ByVal ar As IAsyncResult)
```

Как видим, функции завершения передается единственный параметр: объект, реализующий интерфейс `IAsyncResult`.

Рассмотрим пример, иллюстрирующий описанные возможности.

```
Imports System
Imports System.Threading 'Нужно для "усыпления" потоков

'Делегат для асинхронного метода
Public Delegate Sub Func(ByVal x As Integer)

Class MainClass
    'Этот метод делает необходимую работу
    Public Shared Sub Fib(ByVal n As Integer)
        Dim a As Integer = 1
        Dim b As Integer = 1
        Dim res As Integer = 1
        For i As Integer = 3 To n
            res = a + b
            a = b
            b = res
            Thread.Sleep(10) 'Намеренно замедлим!
        Next
        Console.WriteLine("Fib calculated: " + res)
    End Sub

    Public Shared Sub Main()
        Dim A As Func = AddressOf Fib

        'Асинхронный вызов "выстрелил и забыл"
        'У BeginInvoke() три параметра, два мы не используем
        A.BeginInvoke(6, Nothing, Nothing)

        'Изображаем работу
        For i As Integer = 1 To 9
            Thread.Sleep(20)
            Console.Write(i)
        Next
    End Sub
End Class
```

Вывод программы:

```
12Fib calculated: 8
3456789
```

В данном приложении имеется функция для подсчета n-ного числа Фибоначчи. Чтобы эмулировать продолжительные действия, функция намеренно замедлена. После подсчета число выводится на экран. Ни функции завершения, ни возвращаемое `BeginInvoke()` значение не используется. Подобный метод

работы с асинхронными методами называется *«выстрелил и забыл»* (fire and forget).

Модифицируем предыдущее приложение. Будем использовать при вызове `BeginInvoke()` функцию завершения, выводящую строку текста:

```
Imports System
Imports System.Threading

Public Delegate Sub Func(ByVal x As Integer)

Class MainClass
    Public Shared Sub Fib(ByVal n As Integer)
    . . .
    End Sub

    'Это будет функция завершения
    Public Shared Sub Callback(ByVal ar As IAsyncResult)
        'Достаем параметр
        Dim s As String = CType(ar.AsyncState, String)
     Console.WriteLine("AsyncCall is finished with string "+s)
    End Sub

    Public Shared Sub Main()
        Dim A As Func = AddressOf Fib
        'Два асинхронных вызова
        A.BeginInvoke(6, AddressOf Callback, "The end")
        A.BeginInvoke(8, AddressOf Callback, "Second call")
        'Изображаем работу
        For i As Integer = 1 To 9
            Thread.Sleep(20)
            Console.Write(i)
        Next
    End Sub
End Class
```

Вывод программы:

```
12Fib calculated: 8
Async Call is finished with string The end
345Fib calculated: 21
Async Call is finished with string Second call
6789
```

В рассмотренных примерах использовались такие асинхронные методы, которые не возвращают значения. В приложениях может возникнуть необходимость работать с асинхронными методами-функциями. Для этой цели предназначен метод делегата `EndInvoke()`. Сигнатура метода `EndInvoke()` определяется на основе сигнатуры метода, инкапсулированного делегатом. Во-первых, метод является функцией, тип возвращаемого значения – такой же, как у делегата. Во-вторых, метод `EndInvoke()` содержит все `ByRef`-параметры делегата, а его последний параметр имеет тип `IAsyncResult`. При вызове метода `EndIn-`

voke() основной поток выполнения приостанавливается до завершения работы соответствующего асинхронного метода.

Изменим метод Fib() из примера. Пусть он имеет следующую реализацию:

```
Public Shared Function Fib(n As Integer, _
                           ByRef overflow As Boolean) _
                                                  As Integer
    Dim a As Integer = 1
    Dim b As Integer = 1
    Dim res As Integer = 1
    overflow = False
    For i As Integer = 3 To n
        res = a + b
        'Устанавливаем флаг переполнения
        If res < 0 Then overflow = True
        a = b
        b = res
    Next
    Return res
End Function
```

В следующем примере запускаются два асинхронных метода, затем приложение дожидается их выполнения и выводит результаты на экран.

```
Imports System
Imports System.Threading

Delegate Function Func(n As Integer, _
                       ByRef overflow As Boolean) _
                                                  As Integer
Class MainClass
    'Функция считает числа Фибоначчи, следит за переполнением
    Public Shared Function Fib(n As Integer, _
                         ByRef overflow As Boolean) As Integer
        Dim a As Integer = 1
        Dim b As Integer = 1
        Dim res As Integer = 1
        overflow = False
        For i As Integer = 3 To n
            res = a + b
            'Устанавливаем флаг переполнения
            If res < 0 Then overflow = True
            a = b
            b = res
        Next
        Return res
    End Function

    Public Shared Sub Main()
        Dim over As Boolean = False
        Dim A As Func = AddressOf Fib
```

```
        'Так как требуется отслеживать окончание работы методов,
        'игнорировать возвращаемое BeginInvoke() значение нельзя
          Dim ar1 As IAsyncResult = _
                       A.BeginInvoke(10, over, Nothing, Nothing)
          Dim ar2 As IAsyncResult = _
                       A.BeginInvoke(50, over, Nothing, Nothing)

          'Изображаем работу
          For i As Integer = 1 To 9
              Thread.Sleep(20)
              Console.Write(i)
          Next

          'Вспомнили о методах. Остановились и ждем результатов
          Dim res As Integer = A.EndInvoke(over, ar2)
        Console.WriteLine("Result = {0}, over = {1}", res, over)

          ' Теперь второй метод
          res = A.EndInvoke(over, ar1)
        Console.WriteLine("Result = {0}, over = {1}", res, over)
    End Sub
End Class
```

Вывод программы:

```
123456789Result = -298632863, over = True
Result = 55, over = False
```

Подведем небольшой итог. Асинхронный вызов является альтернативой использования многопоточности, так как реализует ее неявно, при помощи среды исполнения. Широкий спектр настроек позволяет решать при помощи асинхронных вызовов большой круг практических задач программирования.

34. Создание приложений Windows Forms.

В этом параграфе будут описаны базовые возможности Visual Basic .NET, связанные с созданием приложений на основе Windows-форм.

Начнем с простейшего приложения, которое выводит на экран пустую форму. Все классы, связанные с формами и визуальными элементами управления, сгруппированы в пространство имен `System.Windows.Forms`. Любая пользовательская форма является объектом класса, порожденного от класса `System.Windows.Forms.Form`. Следовательно, в нашем приложении необходимо импортировать упомянутое пространство имен и создать класс, соответствующий пользовательской форме.

```
Imports System.Windows.Forms
Class MyForm
    Inherits Form
End Class
```

За отображение объекта, соответствующего форме, и организацию цикла обработки сообщений формы отвечает разделяемый метод `Run` класса `Application` пространства имен `System.Windows.Forms`. Обычно код метода

Main оконного приложения содержит строку вызова метода Application.Run[1]. Создадим метод Main, поместив его в классе формы:

```
Imports System.Windows.Forms
Class MyForm
    Inherits Form
    Shared Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class
```

Скомпилируем данный код (пусть он находится в файле f.vb):

```
vbc /r:System.dll,System.Windows.Forms.dll /t:winexe f.vb
```

Обратите внимание: ключ /r:System.Windows.Forms.dll используется для подключения необходимого пространства имен, ключ /t:winexe указывает, что создается оконное приложение (если не указать его, перед выводом окна формы на заднем фоне будет выведено окно консоли).

Попробуем изменить некоторые свойства созданной формы. Простейший способ сделать это – написать собственный конструктор класса формы:

```
Imports System.Windows.Forms
Class MyForm
    Inherits Form
    Sub New()
        MyBase.New() 'Вызов базового конструктора
        Text = "Hello, World!" 'Изменили заголовок формы
    End Sub
    Shared Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class
```

Как правило, интегрированные среды разработки создают конструкторы форм, которые содержат только вызов унаследованного конструктора и вызов специального метода InitializeComponent(), содержащего код инициализации формы и ее отдельных компонентов. Мы модифицируем код в этом же стиле:

```
Imports System.Windows.Forms
Class MyForm
    Inherits Form
    Sub New()
        MyBase.New()
        InitializeComponent()
    End Sub
```

[1] Для отображения формы можно использовать метод формы ShowDialog(). Таким образом, следующий фрагмент кода также создаст и покажет форму на экране:

```
Shared Sub Main()
    Dim F As New MyForm
    F.ShowDialog()
End Sub
```

```
    Private Sub InitializeComponent()
        Text = "Hello, World!" 'Изменили заголовок формы
    End Sub
    Shared Sub Main()
        Application.Run(New MyForm())
    End Sub
End Class
```

Поместим на нашу форму некоторые компоненты пользовательского интерфейса. Как и в Delphi, размещенному на форме компоненту соответствует некое поле формы. Само создание компонента и начальная настройка его свойств обычно происходят в методе `InitializeComponent()`:

```
Imports System.Windows.Forms
Class MyForm
    Inherits Form
    Private TB As TextBox    'Это будет поле ввода
    Private B As Button      'А это кнопка
    Sub New()
        MyBase.New()
        InitializeComponent()
    End Sub
    Private Sub InitializeComponent()
        Text = "Hello, World!" 'Изменили заголовок формы
        TB = New TextBox()     'Создали поле ввода
        B = New Button()       'Создали кнопку
        TB.Location = New System.Drawing.Point(10,10)
        TB.Size = New System.Drawing.Size(100,50)
        'Вот так задаются размеры. Кстати, при компиляции
        'надо подключить System.Drawing.dll
        TB.Text = "Edit1"
        'Аналогично для кнопки
        B.Location = New System.Drawing.Point(150,150)
        B.Text = "Button"
        . . .
    End Sub
    . . .
```

Однако, для отображения компонента на форме этих действий не достаточно. В Delphi для отображения необходимо было установить у компонент свойство `Parent`. Аналогичная ситуация и в Visual Basic .NET:

```
Private Sub InitializeComponent()
    . . . 'Здесь код, записанный в предыдущем листинге
    TB.Parent = Me   'Компоненты отображаются на форме
    B.Parent = Me
    Controls.AddRange(New Control() {TB,B})
End Sub
```

У любой формы имеется свойство `Controls`, которое хранит компоненты, которыми владеет форма. Интегрированные среды устанавливают отношение владения вызовом специального метода `Controls.AddRange` (если требуется добавить в список владения сразу несколько компонентов; когда требуется до-

бавить один компонент, можно использовать метод Controls.Add). Предыдущий код можно переписать следующим образом:

```
Private Sub InitializeComponent()
        . . .
    Controls.AddRange(New Control() {TB,B})
    'Альтернативный способ установить отношения владения
End Sub
```

Код, который создается для генерации формы интегрированной средой, обычно содержит реализацию метода Dispose, вызывающегося по окончанию работы с формой и уничтожающего все компоненты на форме.

```
'Данный фрагмент взят из кода, созданного Visual Studio
Protected Overloads Overrides Sub Dispose(ByVal disposing _
                                          As Boolean)
    If disposing Then
        If Not (components Is Nothing) Then
            components.Dispose()
        End If
    End If
    MyBase.Dispose(disposing)
End Sub
```

Рассмотрим вопросы, связанные с назначением обработчиков событий для компонент, размещаемых на форме. В качестве базового типа для событий всех компонент используется делегат System.EventHandler, определенный следующим образом:

```
Delegate Sub EventHandler(ByVal sender As Object, _
                          ByVal e As EventArgs)
```

Параметр *sender* определяет объект-источник события. Параметр *e* – это объект класса EventArgs. Данный класс не содержит полей. Если обработчик события должен получить дополнительные параметры, то описывается класс, порожденные от класса EventArgs и содержащий поля для параметров.

Назначение обработчиков событий для компонент и формы происходит при помощи стандартной конструкции AddHandler. Естественно, методы, которые должны быть обработчиками событий, должны удовлетворять сигнатуре делегатов, описывающих данные события в классах компонент.

Рассмотрим следующий пример, в котором используется назначение обработчиков событий. Пусть на форме имеется кнопка и поле ввода. При нажатии на кнопку происходит назначение обработчика события полю ввода. Этот обработчик отслеживает нажатые клавиши в поле ввода и выводит их в заголовок окна формы. Если нажимается клавиша ввода, то заголовок окна формы очищается.

```
Imports System
Imports System.Drawing
Imports System.Windows.Forms
```

```
Class MainForm
    Inherits Form

    Private But As Button
    Private TB As TextBox

    Shared Sub Main()
        Dim fMainForm As New MainForm
        fMainForm.ShowDialog()
    End Sub

    Sub New()
        MyBase.New()
        InitializeComponent()
    End Sub

    Private Sub InitializeComponent()
        But = New Button()
        TB = New TextBox()
        But.Location = New Point(24, 16)
        But.TabIndex = 0
        But.Text = "The button"
        AddHandler But.Click, AddressOf ButClick
        TB.Location = New Point(136, 16)
        TB.TabIndex = 1
        TB.Text = "TextBox"
        Me.Controls.Add(Me.TB)
        Me.Controls.Add(Me.But)
        Me.Text = "MainForm"
        'Me.ResumeLayout(false)
    End Sub

    Private Sub ButClick(sender As Object, e As EventArgs)
        AddHandler TB.KeyPress, AddressOf TextBoxEH
    End Sub

    Private Sub TextBoxEH(sender As Object, _
                                          e As KeyPressEventArgs)
        If e.KeyChar = Microsoft.VisualBasic.ChrW(13)
            Me.Text = ""
        Else
            Me.Text = Me.Text & e.KeyChar
        End If
    End Sub
End Class
```

ЛИТЕРАТУРА

1. Hoang Lam, Thuan L. Thai .NET Framework Essentials – O'Reilly, 2003.
2. Jeff Prosise. Programming Microsoft .NET (core reference) – Microsoft Press, 2002.
3. Архангельский А. Программирование в Delphi 6. СПб.: Бином, 2001.
4. Бобровский С. Delphi 6 и Kylix. Библиотека программиста. СПб.: Питер, 2002.
5. Дубовцев А. В. Microsoft .NET в подлиннике. – СПб.: БХВ-Петербург, 2004.
6. Кэнту М. Delphi 6 для профессионалов. СПб.: Питер, 2002.
7. Лишнер Р. Delphi: Справочник. СПб.: Питер, 2001.
8. Пачеко К., Тейксейра С. Borland Delphi 6: Руководство разработчика. М.: Вильямс, 2002.
9. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. – М.: Русская редакция, 2002.