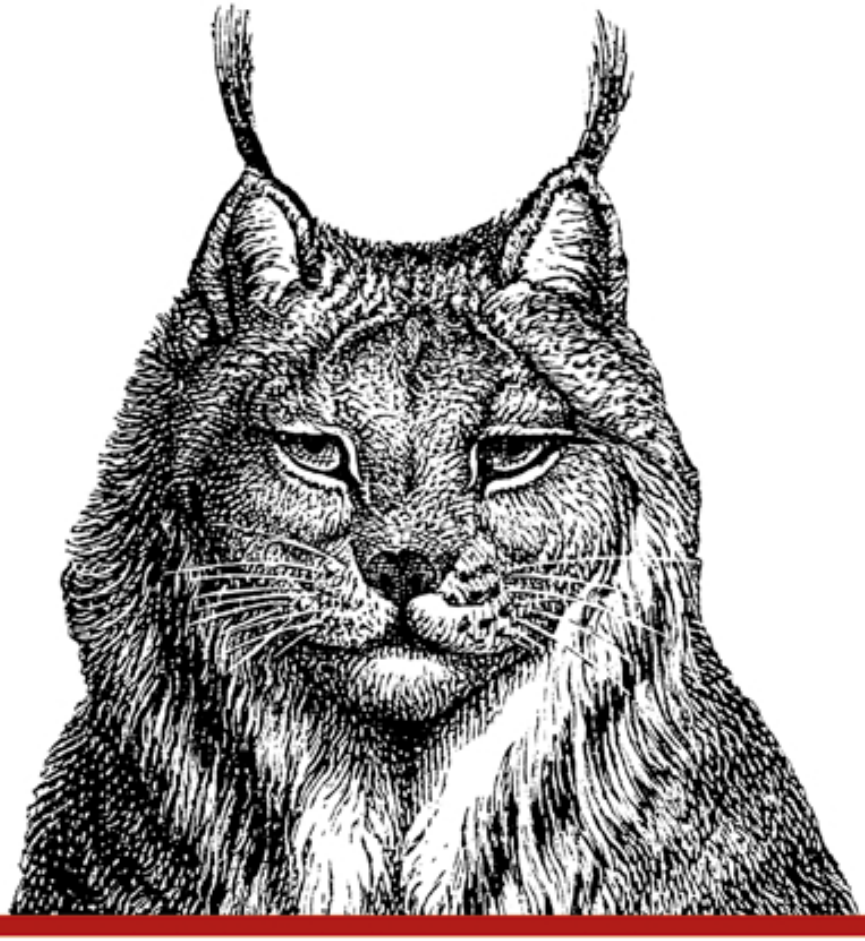
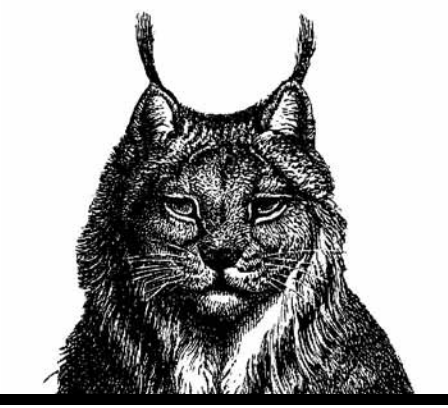


DELPHI

СПРАВОЧНИК

Символ
O'REILLY®

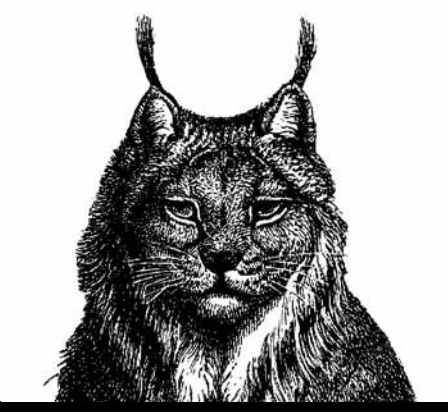
Рэй Лишнер



DELPHI
IN A NUTSHELL

Ray Lischner

O'REILLY®



DELPHI
СПРАВОЧНИК

Рэй Лишнер

Санкт-Петербург
2001

Рэй Лишнер

Delphi. Справочник

Перевод Л. Фрейдина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Г. Шахов*
Редактор	*Н. Японцева*
Корректура	*С. Беляева, С. Журавина*
Верстка	*М. Кривулев, Н. Макарова*

Лишнер Р.
Delphi. Справочник. – Пер. с англ. – СПб: Символ-Плюс, 2001. – 640 с., ил.
ISBN 5-93286-019-7

Популярный справочник по Delphi Pascal издательства O'Reilly содержит подробное описание языка программирования одного из лучших средств быстрой разработки приложений для Windows. Книга открывается введением в язык Delphi Pascal и подробным описанием объектной модели Delphi. Затем следует описание информации о типе времени выполнения (RTTI), являющейся ключом к интегрированной среде разработки (IDE) Delphi. Этот материал плохо документирован в других источниках, в том числе в официальных файлах помощи Delphi. Отдельная глава посвящена параллельному программированию в Delphi и созданию многопоточных приложений.

Основу книги составляет полный упорядоченный справочник по набору средств языка программирования Delphi. Каждая статья справочника включает: синтаксис в соответствии со стандартными соглашениями, описание, список аргументов функции или процедуры, советы и приемы использования данного средства языка в реальных программах, краткий пример и перекрестные ссылки на связанные ключевые слова.

Каким бы опытом работы с Delphi вы ни обладали, эта книга станет вашим постоянным помощником. В ней вы найдете решение многих проблем, а также получите возможность изучить более тонкие вопросы языка.

ISBN 5-93286-019-7
ISBN 1-56592-659-5 (англ)

© Издательство Символ-Плюс, 2001

Authorized translation of the English edition © 2000 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, editorial@books.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 9.04.2001. Формат 70x100$^{1}/_{16}$. Бумага офсетная.
Печать офсетная. Объем 40 печ. л. Тираж 4000 экз. Заказ N

Отпечатано с диапозитивов в ФГУП «Печатный Двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Предисловие

Borland Delphi сочетает в себе современный язык программирования, интегрированную среду разработки (IDE) и библиотеку визуальных компонентов (VCL). Среда разработки Delphi понятна любому, кто использовал подобные средства. Например, редактор форм WYSIWYG[1] позволяет визуально создавать окно, перенося на него компоненты мышью. И что более важно, среда Delphi является объектно-ориентированной, легко расширяемой и настраиваемой, что обусловлено мощью и гибкостью языка программирования Delphi.

Ядро Delphi – это язык программирования Delphi Pascal, содержащий ключевые средства для поддержки IDE и VCL. В нем есть все черты современного объектно-ориентированного языка наряду с элегантностью и простотой Паскаля.

«Настольный справочник по Delphi» – это современное справочное руководство по Delphi Pascal, в котором полностью описан язык и рассмотрены способы его эффективного применения. Опытные Delphi-программисты могут использовать эту книгу как алфавитный справочник. Новичкам Delphi стоит потратить время на прочтение первых нескольких глав. Я надеюсь, что каждый найдет для себя на этих страницах что-то ценное.

Это не старый добрый Паскаль

Delphi Pascal – это один из многих объектно-ориентированных вариантов Паскаля. Язык, используемый в Delphi, развивался в течение нескольких лет и больше не похож на Паскаль, который вы когда-то давно изучали в школе. Помимо модульного программирования и мощной модели классов язык Delphi Pascal содержит немало средств, присущих современному языку программирования, в том числе:

- Интерфейсы (сходные с интерфейсами Java™ и COM)

[1] WYSIWYG – What You see is what You get – что видишь, то и получаешь. – *Примеч. науч. ред.*

- Строки Unicode
- Свойства
- Обработка исключительных ситуаций

Среда Delphi с самого начала разрабатывалась как язык и среда программирования для Windows, и многие Delphi-программисты (в том числе и я) считают Delphi лучшей средой разработки для Windows из имеющихся на сегодняшний день. Delphi включает полную поддержку технологий COM и ActiveX, объектно-ориентированную библиотеку компонентов (VCL) и среду быстрой разработки приложений, которая легко расширяется и настраивается.

Delphi для Linux

Как следует из заголовка, Borland активно работает над переносом Delphi на Linux. Возможно сейчас, когда вы читаете эти строки, версия Delphi для Linux уже вышла[1], привнеся в X-Windows интегрированную среду разработки, включая редактор форм WYSIWYG, поддержку многоуровневых баз данных и технологию CORBA.

Пока Borland не закончил работу, и версия Delphi для Linux не вышла, я могу только предполагать, как будет выглядеть конечный продукт. (Нет, никакой секретной информации я не имею). Можно с уверенностью утверждать, что языковое ядро в Delphi для Linux и Delphi для Windows будет одинаковым, включая классы, объекты, интерфейсы, строки, динамические массивы, обработку исключительных ситуаций, а также базовые типы данных. Большинство встроенных процедур будут работать под Linux так же, как и под Windows.

Некоторые средства языка, описанные в этой книге, являются специфичными для Windows, например, переменные `CmdShow` и `DllProc` или функция `FindHInstance`. Если вы хотите писать код, переносимый с Windows на Linux, нужно избегать использования подобных специфических средств.

Среда Delphi для Windows – наилучшее средство разработки приложений и библиотек для Windows. Для достижения такого результата Borland включил в Delphi несколько специфических для Windows средств. Теперь Borland стремится сделать Delphi для Linux лучшим инструментом разработки для Linux. Можно ожидать, что с этой целью в Delphi будут включены некоторые специфические для Linux возможности.

[1] Данный продукт получил кодовое название Kylix. Работа над этим проектом была начата в 1999 году. О выходе официальной версии Kylix фирма Borland объявила 7 марта 2001 года. – *Примеч. науч. ред.*

Я могу только догадываться, но думаю, что вполне реальным будет написание кода, работающего и под Windows, и под Linux. Однако придется пожертвовать средствами, уникальными для каждой из версий. Написание легко переносимых компонентов, особенно интерфейсных элементов, наверное, будет нелегкой задачей. Создание переносимых приложений, скорее всего, будет проще[1].

Об этой книге

Первые четыре главы содержат материал, посвященный эффективному использованию Delphi, последующие главы представляют собой справочник по языку.

В главе 1 «Delphi Pascal» описаны различия между языком Delphi Pascal и стандартным Паскалем. Если вам знаком Turbo Pascal или другие версии Object Pascal, то будет достаточно быстрого просмотра главы 1, чтобы узнать о новых свойствах Delphi Pascal. Если же вы не работали с Паскалем со времени учебы в колледже много лет назад, стоит прочитать главу 1 целиком, чтобы узнать о новых элегантных возможностях Delphi Pascal. Думаю, вы удивитесь, насколько далеко вперед ушел язык программирования Паскаль за эти годы.

Глава 2 «Объектная модель Delphi» содержит более глубокое описание классов и объектов. Если вы использовали другой вариант Object Pascal, стоит прочитать эту главу, т. к. объектная модель Delphi совершенно иная. Если у вас есть опыт применения других объектно-ориентированных языков программирования, прочитайте главу 2, чтобы понять разницу между Delphi и другими языками, например Java и C++.

В главе 3 «Информация о типе времени выполнения» освещается ключевой момент интегрированной среды разработки Delphi. RTTI (Runtime type information) не описана в официальных файлах помощи Borland, но любой разработчик, имеющий дело с компонентами (т. е. каждый Delphi-программист), должен понимать суть RTTI, в том числе имеющиеся ограничения и возможности использования. Глава 3 содержит все, что следует знать о RTTI, и даже немного больше.

Глава 4 «Создание многопоточных приложений» посвящена месту Delphi в современном многопоточном, многопроцессорном мире. Среда Delphi содержит несколько языковых средств для создания многопоточных приложений, но эти средства трудно использовать, не зная тонкостей и проблем многопоточного программирования. Эта глава

[1] По заявлениям разработчиков Kylix, в состав этого продукта будет включена целая библиотека переносимых компонентов, получившая название CLX. Эта библиотека будет включена также в следующие версии Delphi и C++ Builder для Windows. – *Примеч. науч. ред.*

поможет вам начать эффективную разработку современных программ при помощи Delphi.

Глава 5 «Справочник по языку» составляет основной объем книги. Алфавитный справочник содержит все ключевые слова, директивы, процедуры, типы и переменные языка Delphi Pascal и системных модулей. Подробные примеры иллюстрируют способы эффективного и правильного использования средств языка.

Глава 6 «Системные константы» содержит таблицу соответствующих констант. Эти данные вынесены в отдельную главу, чтобы облегчить работу со справочником, поскольку глава 5 и так слишком велика.

Глава 7 «Операторы» описывает все арифметические и прочие операторы языка Delphi Pascal. Символы не так легко расположить в алфавитном порядке, поэтому вынесение их описания в отдельную главу облегчает поиск информации по конкретному оператору.

Глава 8 «Директивы компилятора» содержит список всех специальных комментариев, которые вы можете включить в исходный код для управления процессом компиляции и сборки программы.

Приложение A «Утилиты командной строки» описывает использование и параметры различных утилит командной строки, имеющихся в Delphi. Эти программы не относятся непосредственно к языку Delphi Pascal, но их часто не замечают, хотя они могут быть очень полезны для профессионала в Delphi.

В приложении B «Модуль SysUtils» вы найдете список всех процедур, типов и переменных модуля `SysUtils`. Этот модуль не является ни частью компилятора (как модуль `System`), ни частью языка Delphi Pascal. `SysUtils` – это часть библиотеки времени выполнения Delphi. Несмотря на это, многие Delphi-программисты рассматривают `SysUtils` как часть языка. И действительно, многие процедуры модуля `SysUtils` превосходят аналогичные утилиты модуля `System` (например `AnsiPos` по сравнению с `Pos`).

Типографские соглашения

В книге приняты следующие соглашения по форматированию текста:

`Моноширинный шрифт`

Применяется для идентификаторов и специальных символов Delphi, а также для всех ключевых слов и директив. В справочнике по языку моноширинный шрифт указывает на синтаксические элементы, которые следует использовать точно так, как напечатано. Например, объявление массива требует наличия квадратных скобок и других символов, а примеры использования ключевых слов `type`, `array` и `of` приведены ниже:

```
type Имя = array [Тип индекса, ...] of Базовый тип;
```

Моноширинный курсив

В справочнике по языку обозначает синтаксические элементы, которые должны быть заменены в реальном коде. В предыдущем примере следует подставить Имя типа, Тип индекса и Базовый тип.

Жирный моноширинный шрифт

В больших примерах кода служит для выделения обсуждаемого в данный момент элемента языка.

Курсив

Предназначен для переменных, имен файлов, каталогов, URL и словарных терминов.

Символы примечаний

Изображение совы означает подсказку, совет или общее замечание к обозначенному тексту.

Изображение индюка означает предупреждение, относящееся к данному тексту.

Другие источники информации

Если у вас есть вопрос по Delphi, в первую очередь следует обратиться к файлам помощи. В поставку Delphi также входит много примеров (каталог *Demos*), которые часто оказываются полезнее, чем файлы помощи.

Если вы так и не нашли ответ, попробуйте отправить свой вопрос в одну из многочисленных телеконференций. Существует несколько стандартных телеконференций, а также форумы, созданные фирмой Borland на своем сервере: *forums.borland.com*. В частности, для вопросов по языку Delphi Pascal подходит телеконференция *borland.public.delphi.objectpascal*.

Если вы хотите узнать об интегрированной среде Delphi, библиотеке визуальных компонентов и других темах, связанных с программированием Delphi, прочитайте две наиболее популярные книги: Marco Cantu «Mastering Delphi 5», издательство Sybex, 1999 год и Steve Teixeira и Xavier Pacheco «Delphi 5 Developer's Guide»[1], издательство Sams, 1999 год.

[1] Стив Тейксейра и Ксавье Пачеко «Delphi 5. Руководство разработчика», в 2–х томах, издательство «Вильямс», 2000 г.

Если вы найдете ошибки или пропуски в этой книге, пожалуйста, сообщите о них по адресу *nutshell@tempest-sw.com*. Я получаю очень много писем и не могу ответить на каждое отдельно, но не сомневайтесь, что я прочитаю все (по крайней мере то, что проходит через мои фильтры против почтового мусора).

Как с нами связаться

В этой книге мы постарались дать максимально точную и проверенную информацию, но ошибки и оплошности все равно могут вам встретиться. Сообщения о них, а также предложения для будущих изданий присылайте по адресу:

O'Reilly & Associates, Inc.

101 Morris Street

Sebastopol, CA 95472

(800) 998-9938 (in the U.S. or Canada)

(707) 829-0515 (international or local)

(707) 829-0104 (fax)

Вы также можете связаться с нами по электронной почте. Чтобы подключиться к нашему списку рассылки или заказать каталог, напишите по адресу:

info@oreilly.com

По техническим вопросам или с комментариями к этой книге обращайтесь по адресу:

bookquestions@oreilly.com

Об ошибках и планах по дальнейшим изданиям можно узнать на вебсайте этой книги:

www.oreilly.com/catalog/delphi

Другую информацию об этой и других книгах издательства O'Reilly можно найти на сайте издательства:

www.oreilly.com

Благодарности

Я благодарю Тима О'Рейли за то, что он воспользовался шансом издать свою первую книгу по Delphi. Надеюсь и дальше читать и писать книги по Delphi, издаваемые издательством O'Reilly.

Технические редакторы – Аллен Бауэр и Холвард Вассботн – отлично выполнили работу по поиску моих ошибок. Многочисленные и подробные комментарии Холварда были бесценны. Оставшиеся в книге

ошибки – это те, которые я добавил после того, как редакторы закончили свою тщательную работу.

Я благодарю моего редактора, Саймона Хэйеса, и всю бригаду издательства O'Reilly – Боба Хербстмана, Троя Мотта, сотрудников, занимающихся дизайном и производством, за то, что они превратили мою скромную рукопись в ту элегантную книгу, которую вы теперь видите.

Ни одна из моих книг не была бы возможна без поддержки семьи. Я благодарю мою жену, Черил, и программиста будущего, Артура, ради которых все это делалось.

Глава 2

Объектная модель Delphi

Поддержка объектно-ориентированного программирования в Delphi богата и разнообразна. В дополнение к традиционным классам и объектам, в Delphi также имеются интерфейсы (аналогично имеющимся в Java[1] и в технологии COM), обработка исключительных ситуаций и поддержка многопоточного программирования. В этой главе подробно раскрывается объектная модель Delphi. Вы должны быть уже знакомы со стандартным Паскалем и основными принципами объектно-ориентированного программирования.

Классы и объекты

Классы – это записи на стероидах. Как и запись, класс (`class`) описывает тип, в котором может быть какое угодно количество переменных, называемых *полями*. В отличие от записи, класс может также содержать функции и процедуры, называемые *методами*, и *свойства*. Класс также может быть потомком другого класса, наследуя при этом все поля, методы и свойства объекта-предка.

Объект – это динамический экземпляр класса. Объект всегда создается динамически, в «куче», поэтому ссылка на объект фактически является указателем (но при этом не требует обычного оператора разыменования). Когда вы присваиваете переменной ссылку на объект, Delphi копирует только указатель, а не весь объект. Используемый объект должен быть освобожден явно. Delphi не имеет автоматической

[1] Интерфейсы в Delphi не вполне аналогичны интерфейсам в Java. В частности, любой интерфейс в Delphi является расширением интерфейса IUnknown, в то время как интерфейсы в Java свободны от этого ограничения. – *Примеч. науч. ред.*

системы «сборки мусора» (смотрите раздел «Интерфейсы» далее в этой главе).

Для краткости термин *ссылка на объект* часто сокращается до *объект*, но, если быть точным, объект – это участок памяти, где Delphi хранит значения всех полей объекта. Ссылка на объект – это указатель на объект. В Delphi с объектом можно работать только через ссылку на него. Ссылка на объект обычно представлена в форме переменной, но она может быть также функцией или свойством другого объекта.

Класс – это также вполне конкретное понятие (аналогичное классу в Java, но отличное от C++). В Delphi класс представлен таблицей указателей на виртуальные методы и множеством другой, доступной только для чтения, информации об этом классе. *Ссылка на класс* – это указатель на эту таблицу. (В главе 3 «Информация о типе времени выполнения» подробно описаны структуры этих таблиц.) Чаще всего ссылка на класс используется для создания объектов или для проверки их типов, но ее можно использовать и во многих других ситуациях, в том числе передавать ее в качестве параметра или возвращать как результат функции. Тип ссылки на класс называется *метаклассом*.

В примере 2-1 приведено несколько определений классов. Определение класса – это определение типа, которое начинается с ключевого слова `class`. Определение класса содержит описания полей, методов и свойств, завершающееся ключевым словом `end`. Каждое описание метода аналогично объявлению прототипа процедуры или функции: далее в модуле вы должны привести его код (кроме абстрактных методов, которые обсуждаются в этой главе ниже).

Пример 2-1. Пример объявления классов и объектов

```
type
  TAccount = class
  private
    fCustomer: string;  // имя покупателя
    fNumber: Cardinal;  // номер счета
    fBalance: Currency; // текущий баланс на счете
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer; // фактический процент, умноженный на 1000
  end;
  TCheckingAccount = class(TAccount)
  private
    fReturnChecks: Boolean;
  end;
  TCertificateOfDeposit = class(TSavingsAccount)
  private
    fTerm: Cardinal;  // Срок депозита, в днях
  end;
```

```
var
  CD1, CD2: TAccount;
begin
  CD1 := TCertificateOfDeposit.Create;
  CD2 := TCertificateOfDeposit.Create;
  ...
```

На рис. 2-1 приведено распределение памяти для объектов и классов из примера 2-1. Переменные и относящиеся к ним объекты располагаются в доступной для записи памяти. Классы располагаются в защищенном участке памяти вместе с кодом программы.

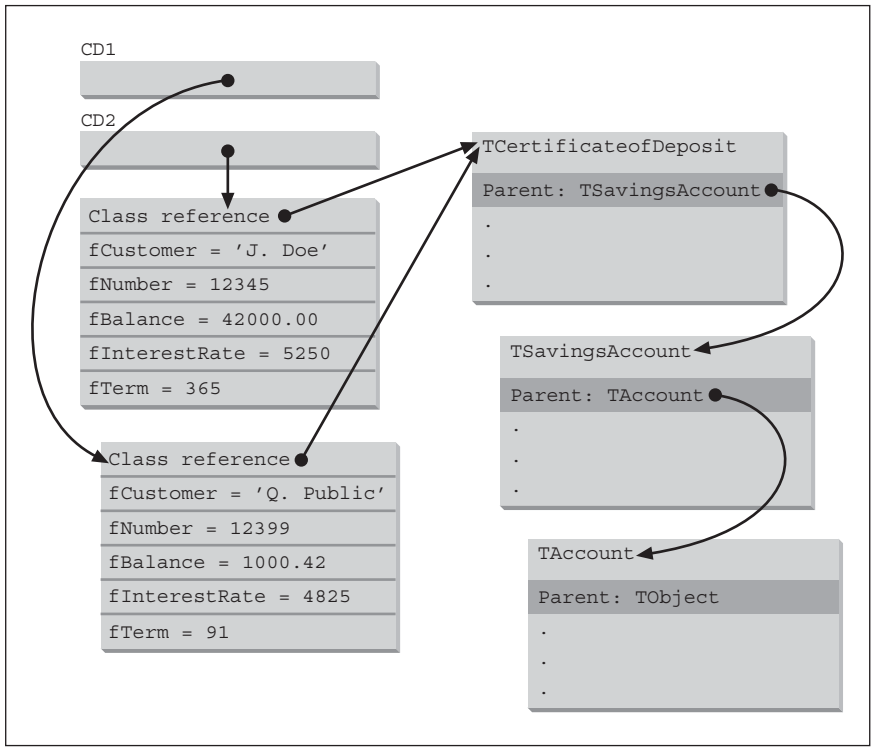


Рис. 2-1. Распределение памяти для объектов и классов

Объектная модель в Delphi аналогична другим языкам, например C++ и Java. В табл. 2-1 приведено краткое сравнение между Delphi и несколькими другими популярными языками программирования. Далее все указанные в таблице языковые возможности описаны подробно[1].

[1] В приведенной таблице автор рассматривает языковые возможности стандартного C++. В ряде широко известных реализаций этого языка (например, в такой популярной, как Microsoft Visual C++) имеется встроенная поддержка многопоточных приложений, OLE-автоматизации, интерфейсов, RTTI и целый ряд других возможностей. – *Примеч. науч. ред.*

Таблица 2-1. Delphi в мире языков программирования

Возможности языка	Delphi	Java	C++	Visual Basic
Наследование	✓	✓	✓	
Множественное наследование			✓	
Интерфейсы	✓	✓	a	✓
Единый корневой класс	✓	✓		
Метаклассы	✓	✓		
Статические поля (поля классов)		✓	✓	
Виртуальные методы	✓	✓	✓	
Абстрактные виртуальные методы	✓	✓	✓	
Статические методы (методы классов)	✓	✓	✓	
Динамические методы	✓			
«Сборка мусора»	b	✓		b
Вариантные типы	✓			✓
OLE-автоматизация	✓			✓
Статическая проверка типов	✓	✓	✓	
Обработка исключительных ситуаций	✓	✓	✓	✓
Перегрузка функций	✓	✓	✓	
Перегрузка операторов			✓	
Неклассовые функции	✓		✓	✓
Необъектные переменные	✓		✓	✓
Свойства	✓			✓
Информация о типах времени выполнения	✓	✓	c	
Шаблоны типов			✓	
Встроенная поддержка многопоточных приложений	✓	✓		
Передача сообщений	✓			
Встроенный ассемблер	✓		d	
Встроенные функции			✓	

[a] C++ может эмулировать интерфейсы с помощью абстрактных классов.

[b] Интерфейсы используют счетчики ссылок для определения своего жизненного цикла.

[c] Использование RTTI в C++ ограничено сравнением и преобразованием типов.

[d] Встроенный ассемблер не является частью стандарта языка C++, но большинство компиляторов, в том числе компиляторы фирмы Borland, поддерживают встроенный ассемблер как расширение языка.

Классы

Определение класса – это особый вид определения типа. В определении класса перечислены поля, методы и свойства класса. Вы можете объявить класс как в разделе интерфейса, так и в разделе реализации модуля, а методы – как и любую процедуру или функцию – только в разделе реализации. Реализация методов класса должна обязательно располагаться в том же модуле, что и определение класса.

Определение класса может иметь одну или более секций для различных уровней доступа (`private`, `public`, `published` или `automated`). Уровни доступа обсуждаются в этой же главе позднее. Вы можете располагать разделы в любом порядке и повторять разделы с одинаковым уровнем доступа.

Однин раздел может содержать любое количество полей, за которыми следуют описания методов и свойств. Описания методов и свойств можно смешивать между собой, но они всегда должны идти после описаний всех полей раздела. В отличие от Java и C++, в Delphi невозможно объявлять какие-либо типы внутри описания класса.

Класс может иметь один базовый класс, от которого наследуются все поля, свойства и методы. Если вы явно не указываете базовый класс, Delphi использует в качество базового класс `TObject`. Кроме того, в классе также может быть реализовано любое количество интерфейсов. Таким образом, объектная модель Delphi больше всего похожа на Java, где класс также может быть расширением только одного класса и реализовывать несколько интерфейсов.

В Delphi принято соглашение, при котором название типа обычно начинается с буквы T, как в `TObject`. Это только соглашение, а не правило языка. IDE, с другой стороны, всегда формирует названия классов форм, начиная с буквы T.

Ссылка на класс – это выражение, ссылающееся на определенный класс. Ссылка на класс не является первым объектом класса, как это принято в Java или Smalltalk, а служит для создания новых объектов, вызовов методов класса, проверки или преобразования типа объекта. Ссылка на класс реализуется как указатель на таблицу, содержащую информацию о классе, в частности, на таблицу виртуальных методов класса (VMT). (Подробности о структуре VMT читайте в главе 3).

Чаще всего ссылка на класс используется для создания экземпляров этого класса с помощью вызова конструктора. Ссылки на класс также

применяются для проверки типа объекта (с помощью оператора `is`) или преобразования объектов к определенному типу (оператор `as`). Обычно ссылка на класс – это имя класса, но это может быть и переменная типа «метакласс», функция или свойство, которое возвращает ссылку на класс. В примере 2-2 представлен образец описания класса.

Пример 2-2. Объявление класса и метакласса

```
type
  TComplexClass = class of TComplex; // тип «метакласс»
  TComplex = class(TPersistent)
  private
    fReal, fImaginary: Double;
  public
    constructor Create(Re: Double = 0.0); overload;
    constructor Create(Re, Im: Double); overload;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    function AsString: string;
  published
    property Real: Double read fReal write fReal;
    property Imaginary: Double read fImaginary write fImaginary;
  end;
```

Объекты

Объект – это динамический экземпляр класса. Объект содержит значения всех полей, объявленных классом и его предками. В объекте также имеется скрытое поле, в котором содержится ссылка на класс этого объекта.

Объекты всегда создаются динамически, в «куче», и потому ссылка на объект – это фактически указатель. Программист сам отвечает за создание объектов и за их своевременное освобождение. Для создания объекта используется ссылка на класс с указанием конструктора, например:

```
Obj := TSomeClass.Create;
```

Большинство конструкторов называются `Create`, но это лишь соглашение, а не требование Delphi. Вы можете иногда встретить конструкторы и с другими именами, обычно у старых классов, написанных еще до появления в Delphi поддержки перегрузки подпрограмм. Для максимальной совместимости с C++ Builder, в котором запрещается самостоятельное именование конструкторов, рекомендуется ограничиваться названием `Create` для всех конструкторов ваших объектов. Чтобы избавиться от объекта, который больше не требуется вашей программе, вызовите метод `Free`. Для гарантии правильного освобождения объекта, даже в случае возникновения исключительной ситуации, используйте обработчик `try-finally`. (Подробную информацию о конструкции `try-finally` смотрите в главе 1 «Delphi Pascal».) Например:

```
Obj := TSomeOtherClass.Create;
try
  Obj.DoSomethingThatMightRaiseAnException;
  Obj.DoSomethingElse;
finally
  Obj.Free;
end;
```

При освобождении глобальной переменной или поля всегда устанавливайте переменную в `nil` до освобождения объекта, чтобы в переменной не оставалось неверного указателя. Если деструктор или метод, вызываемый из деструктора, ссылается на эту переменную, ее обычно обнуляют, чтобы исключить потенциальные проблемы. Самый простой способ это сделать – вызвать процедуру `FreeAndNil` (из модуля `SysUtils`):

```
GlobalVar := TFruitWigglies.Create;
try
  GlobalVar.EatEmUp;
finally
  FreeAndNil(GlobalVar);
end;
```

Каждый объект содержит отдельную копию всех своих полей. Поля не могут разделяться между несколькими объектами. Если вы хотите, чтобы несколько объектов использовали одну переменную, объявите ее на уровне модуля или обращайтесь к ней неявно: многие объекты могут иметь разные указатели или ссылки на объекты, ссылающиеся на общие данных.

Наследование

Класс может быть наследником другого класса. Производный класс наследует все поля, методы и свойства базового класса. Delphi поддерживает только однократное наследование, т.е. класс имеет только один базовый. Базовый класс может иметь свой базовый класс и так далее, и класс наследует поля, свойства и методы всех родительских классов. Класс также может реализовывать любое количество интерфейсов (смотрите далее в этой главе). Так же, как в Java, но в отличие от C++, каждый класс является наследником одного корневого класса – `TObject`, который в Delphi является базовым по умолчанию.

Базовый класс – это непосредственный предок класса, который описан в его объявлении. Родительский класс – это либо базовый класс, либо любой другой класс в цепочке наследования вплоть до `TObject`. Так, в примере 2-1 для класса `TCertificateOfDeposit` базовым является `TSavingsAccount`, а родителями – классы `TObject`, `TAccount` и `TSavingsAccount`.

В классе TObject объявлено несколько методов и одно специальное, скрытое поле для хранения ссылки на класс объекта. Это скрытое поле указывает на таблицу виртуальных методов (VMT). Каждый класс имеет уникальную VMT, а все объекты используют одну и ту же VMT. В главе 5 «Справочник по языку» подробно описан класс TObject и его методы.

Вы можете присвоить ссылку на объект переменной, тип которой либо совпадает с типом класса, либо является одним из его родительских классов. Другими словами, объявленный тип ссылки не всегда совпадает с фактическим типом объекта. Присваивания в обратном направлении – присваивание ссылки на базовый класс переменной производного класса – не допускаются.

В Delphi сохраняется строгая типизация Паскаля, и компилятор выполняет проверку на основе объявленного типа объекта. Таким образом, все методы должны существовать в объявленном классе, и компилятором выполняется обычная проверка аргументов процедур и функций. Компилятор не обязательно привязывает вызов метода к конкретной его реализации. Если метод виртуальный, Delphi откладывает эту работу до времени выполнения и использует в этом случае истинный тип объекта для определения метода, который следует вызвать. Смотрите далее в этой главе раздел «Методы».

Для определения истинного типа объекта используется оператор is. Он возвращает истину, если указанный класс совпадает с классом объекта или с одним из его родительских классов. Если это не так или ссылка на объект нулевая, возвращается «ложь». Например:

```
if Account is TCheckingAccount then ... // Проверяет класс для Account
if Account is TObject then ...          // Истина, если Account не равно nil
```

Кроме того, получить ссылку на объект другого класса можно с помощью преобразования типа. Преобразование типа не изменяет объект; оно лишь дает вам иную ссылку на объект. Обычно для преобразования типов объектов используется оператор as. Этот оператор автоматически проверяет тип объекта и вызывает ошибку в том случае, если класс объекта не является потомком целевого класса. (Модуль SysUtils преобразует ошибку времени выполнения в исключительную ситуацию EinvalidCast.)

Другой способ преобразования ссылки на объект – указание имени целевого класса в стандартном формате преобразования типа, как при вызове функции. Этот вариант преобразования не выполняет проверку на соответствие типов, поэтому используйте его, только если вы уверены в безопасности преобразования, как в примере 2-3.

Пример 2-3. Использование статических преобразований типов

```
var
  Account: TAccount;
```

```
  Checking: TCheckingAccount;
begin
  Account  := Checking;                    // Допустимо
  Checking := Account;                     // Ошибка компиляции
  Checking := Account as TCheckingAccount; // Допустимо
  Account as TForm;                        // Ошибка при выполнении
  Checking := TCheckingAccount(Account);   // Допустимо, но не рекомендуется
  if Account is TCheckingAccount then      // Лучше
    Checking := TCheckingAccount(Account)
  else
    Checking := nil;
```

Поля

Поле – это переменная, являющаяся частью объекта. Класс может объявлять любое количество полей, и в каждом объекте содержится собственная копия всех полей, объявленных в его классе и во всех родительских классах. В других языках поля называются членами (data member), переменными экземпляра (instance variable) или атрибутами (attribute). В Delphi отсутствуют переменные класса, переменные экземпляра, статические члены или их эквиваленты (т. е. переменные, разделяемые между всеми объектами одного класса). Вместо них обычно используются переменные, объявленные на уровне модуля, обеспечивающие аналогичную функциональность.

Поле может быть любого типа, кроме полей из секции `published`. Эти поля обязательно должны относиться к какому-либо классу, который имеет информацию времени выполнения (т. е. компилированному с директивой `$M+`). Подробности смотрите в главе 3.

Когда Delphi впервые создает объект, все его поля пусты, т. е. указатели равны `nil`, строки и динамические массивы пусты, числа равны нулю, логические поля имеют значение «ложь», поля типа `Variant` – `Unassigned`. (Подробности смотрите в описании процедур `NewInstance` и `InitInstance` в главе 5.)

Производный класс может объявить поле с тем же именем, что и поле в родительском классе, в этом случае поле с тем же именем в родительском классе скрывается. Методы в производном классе будут ссылаться на новое поле, методы в классе предке – на старое.

Методы

Методы – это функции и процедуры, применяемые только к объектам определенного класса и его потомкам. В языке C++ методы называются «функции-члены» (member functions). Методы отличаются от обычных процедур и функций тем, что каждый метод содержит неявный параметр `Self`, ссылающийся на объект, вызвавший этот метод. `Self` аналогичен `this` в C++ и Java. Вызов метода выполняется аналогично

вызову функции или процедуры, но ему предшествует ссылка на объект, например:

```
Object.Method(Argument);
```

Метод класса относится к классу в целом и его потомкам. В методе класса `Self` ссылается не на отдельный объект, а на класс. В C++ для методов класса используется термин «статическая функция-член» (static member function).

Вы можете вызвать метод, объявленный в собственном классе объекта или в любом из его классов-предков. Если один и тот же метод объявлен в исходном классе и в производном, Delphi вызывает метод производного класса, как в примере 2-4.

Пример 2-4. Связывание статических методов

```
type
  TAccount = class
  public
    procedure Withdraw(Amount: Currency);
  end;
  TSavingsAccount = class(TAccount)
  public
    procedure Withdraw(Amount: Currency);
  end;
var
  Savings: TSavingsAccount;
  Account: TAccount;
begin
  ...
  Savings.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
  Account.Withdraw(1000.00);     // Вызывается TAccount.Withdraw
```

Обычный метод еще называют *статическим*[1], так как Delphi напрямую связывает вызов метода с его реализацией на этапе компиляции. Другими словами, выполняется статическое связывание. В C++ это называется «обычной функцией-членом» (ordinary member function), в Java – «окончательный метод» (final method). Большинство Delphi-программистов избегают термина «статический метод», предпочитая более простой – *метод* или даже *невиртуальный метод*.

Виртуальный метод – это метод, связывание которого выполняется во время выполнения, а не во время компиляции. При компиляции Delphi определяет по объявленному типу объекта, какие методы допустимы для использования с этим объектом. Вместо формирования прямой ссылки на определенный метод компилятор вставляет неяв-

[1] Не следует путать статические методы в Delphi со статическими функциями (методами) в C++ и Java. Близким аналогом статических функций (методов) C++ и Java в Delphi являются методы класса. – *Примеч. науч. ред.*

ную ссылку на метод, которая зависит от фактического класса объекта. Во время выполнения программы Delphi находит метод в таблицах класса (конкретно, в VMT) и вызывает метод фактического класса объекта. Истинный класс объекта может как совпадать с объявленным, так и быть производным от него, – это значения не имеет, т. к. ссылка на нужный метод будет взята из VMT.

Для объявления виртуального метода предназначена директива `virtual` в базовом классе, и директива `override` в описании нового метода в производном классе. В отличие от Java, все методы по умолчанию являются статическими, и для объявления виртуального метода требуется указание директивы `virtual`. В отличие от C++, для переопределения виртуального метода в производном классе требуется директива `override`. В примере 2-5 применяются виртуальные методы.

Пример 2-5. Связывание виртуальных методов

```
type
  TAccount = class
  public
    procedure Withdraw(Amount: Currency); virtual;
  end;
  TSavingsAccount = class(TAccount)
  public
    procedure Withdraw(Amount: Currency); override;
  end;
var
  Savings: TSavingsAccount;
  Account: TAccount;
begin
  ...
  Savings.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
  Account := Savings;
  Account.Withdraw(1000.00);     // Вызывается TSavingsAccount.Withdraw
```

Директива `virtual` может быть заменена директивой `dynamic`. Семантики этих директив одинаковы, но реализации различны. Поиск виртуального метода в VMT выполняется быстрее, так как компилятор генерирует ссылку непосредственно на VMT. Поиск динамического метода происходит дольше, так как его вызов требует линейного поиска в таблице динамических методов класса (DMT). Если класс не переопределяет такой метод, поиск продолжается в DMT базового класса. Поиск продолжается во всех родительских классах либо до `TObject`, либо до того, как метод не будет найден. Динамические методы могут занимать меньше памяти, чем виртуальные, однако, если только вы не пишете заменитель VCL, используйте виртуальные методы, а не динамические. В главе 3 приведено полное описание реализации динамических и виртуальных методов.

Виртуальный или динамический метод может быть объявлен с директивой `abstract`, в этом случае класс не предоставляет реализации этого

метода. Вместо него метод *должен* быть переопределен в производных классах. В C++ для абстрактного метода принят термин «чисто виртуальный метод» (pure virtual method). При вызове конструктора для класса, который содержит абстрактный метод, компилятор выдает предупреждение, сообщающее о возможной ошибке. Возможно, вы хотите создать экземпляр одного из производных классов, в котором абстрактный метод переопределен. Класс, содержащий один и более абстрактных методов, часто называют *абстрактным*, хотя этот термин, строго говоря, обозначает классы, в котором все методы – абстрактные.

Если вы создаете абстрактный класс, который является наследником другого абстрактного класса, вам следует переопределить все абстрактные методы с указанием директив `override` и `abstract`. Delphi этого не требует, но этого требует здравый смысл. Новые определения ясно сообщают, что эти методы являются абстрактными. Иначе читатель вашего кода может не понять, следует ли реализовывать эти методы или оставить их абстрактными. Например:

```
type
  TBaseAbstract = class
    procedure Method; virtual; abstract;
  end;
  TDerivedAbstract = class(TBaseAbsract)
    procedure Method; override; abstract;
  end;
  TConcrete = class(TDerivedAbstract)
    procedure Method; override;
  end;
```

Метод класса или конструктор также могут быть виртуальными. В Delphi ссылка на класс – вполне реальная сущность, которая может быть присвоена переменной, передана как параметр или использована для вызова методов класса. Если конструктор является виртуальным, вы можете присвоить производный класс ссылке на базовый. Delphi найдет виртуальный конструктор в VMT и вызовет конструктор производного класса.

Методы (как и другие функции и процедуры) могут быть перегружены, т. е. несколько подпрограмм могут иметь одно и то же имя, если они имеют разные аргументы. Перегружаемые методы объявляются с директивой `overload`. Класс может перегрузить метод, унаследованный из базового класса. В этом случае директива `overload` требуется только в производном классе. Действительно, создатель базового класса не может предвидеть будущего и знать, что другие программисты захотят перегрузить унаследованный метод. Без директивы `overload` в производном классе новый метод «спрячет» метод базового класса, как в примере 2-6.

Пример 2-6. Перегрузка методов

```
type
  TAuditKind = (auInternal, auExternal, auIRS, auNasty);
  TAccount = class
  public
    procedure Audit;
  end;
  TCheckingAccount = class(TAccount)
  public
    procedure Audit(Kind: TAuditKind); // Скрывает TAccount.Audit
  end;
  TSavingsAccount = class(TAccount)
  public
    // Позволяет вызвать как TSavingsAccount.Audit, так и TAccount.Audit
    procedure Audit(Kind: TAuditKind); overload;
  end;
var
  Checking: TCheckingAccount;
  Savings: TSavingsAccount;
begin
  Checking := TCheckingAccount.Create;
  Savings := TSavingsAccount.Create;
  Checking.Audit;              // Ошибка, так как TAccount.Audit спрятан
  Savings.Audit;               // Правильно, так как Audit перегружен
  Savings.Audit(auNasty);      // Правильно
  Checking.Audit(auInternal); // Правильно
```

Конструкторы

Каждый класс может иметь один и более *конструкторов* (constructor), которые могут быть унаследованы от базового класса. Согласно общепринятой практике, конструкторы обычно называются `Create`, хотя вы можете использовать любое имя. Имена некоторых конструкторов начинаются с префикса `Create`, за которым следует дополнительная информация, например `CreateFromFile` или `CreateFromStream`. Однако обычно достаточно имени `Create`, а несколько конструкторов можно определить при помощи перегрузки методов. Другая причина использования перегрузки конструктора с именем `Create` заключается в необходимости обеспечить совместимость с C++ Builder, поскольку в C++ не разрешаются конструкторы с разными именами.

Вызов конструктора

Конструктор – это гибрид метода объекта и метода класса. Вы можете вызывать его как по ссылке на объект, так и по ссылке на класс. Delphi передает конструктору дополнительный, скрытый параметр для указания способа, при помощи которого он был вызван. Если конструктор был вызван с помощью ссылки на класс, Delphi вызывает метод класса `NewInstance` для создания нового экземпляра класса. После вызова `NewInstance` конструктор продолжает выполнение и инициализирует

объект. Delphi автоматически формирует блок `try-except`, и если при работе конструктора возникает исключительная ситуация, Delphi вызывает деструктор.

При вызове конструктора по ссылке на объект Delphi не формирует блок `try-except` и не вызывает метод `NewInstance`. Вместо этого конструктор вызывается как обычный метод. Это позволяет вызвать унаследованный конструктор без дополнительных накладных расходов.

Обычной ошибкой является попытка создать объект, вызвав конструктор со ссылкой на объект вместо использования ссылки на класс и присваивания результата переменной объектного типа:

```
var
  Account: TSavingsAccount;
begin
  Account.Create;                    // неправильно
  Account := TSavingsAccount.Create; // правильно
```

Одной из возможностей Delphi является предоставление программисту полного контроля над тем, когда, где и как вызывать унаследованный конструктор. Это позволяет создавать мощные и интересные классы, но также является источником частых ошибок.

Delphi всегда вначале исполняет конструктор производного класса, и только в том случае, если в этом классе вызывается конструктор базового класса, исполняет и его. C++ создает классы в обратном порядке, начиная с родительского класса и заканчивая производным классом. Таким образом, если класс C является наследником B, который является наследником A, Delphi сначала вызывает конструктор класса C, потом B и в конце A. В отличие от этого, C++ сначала запускает конструктор класса A, потом B и завершает C.

Виртуальные методы и конструкторы

Другой существенной разницей между C++ и Delphi является то, что в C++ конструктор всегда исполняется с виртуальной таблицей того класса, конструктор которого работает в данный момент, а в Delphi работают виртуальные методы производного класса, даже если исполняется конструктор базового класса. Поэтому следует быть осторожным при написании виртуальных методов, которые могут быть вызваны из конструктора. Возможно, что при вызове этого метода объект еще не полностью сформирован. Чтобы избежать подобных проблем, следует переопределить метод `AfterConstruction` и использовать его для кода, который требует полностью сформированного объекта. Если вы переопределяете `AfterConstruction`, не забудьте также вызвать унаследованный метод.

Один конструктор может вызывать другой конструктор. Delphi может определить, что вызов был сделан с помощью ссылки на объект (а имен-

но Self), и исполнить его как обычный метод. Наиболее частая причина вызова одного конструктора из другого – это желание поместить весь код инициализации в один конструктор. В примере 2-7 показаны несколько различных способов описания и вызова конструкторов.

Пример 2-7. Объявление и вызов конструкторов

```
type
  TCustomer = class ... end;
  TAccount = class
  private
    fBalance: Currency;
    fNumber: Cardinal;
    fCustomer: TCustomer;
  public
    constructor Create(Customer: TCustomer); virtual;
    destructor Destroy; override;
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer; // Умноженное на 1000
  public
    constructor Create(Customer: TCustomer); override; overload;
    constructor Create(Customer: TCustomer; InterestRate: Integer);
        overload;
    // Заметьте, что TSavingsAccount не требуется деструктор. Он только
    // наследует деструктор класса TAccount.
  end;

var
  AccountNumber: Cardinal = 1;

constructor TAccount.Create(Customer: TCustomer);
begin
  inherited Create;             // Вызов TObject.Create.
  fNumber := AccountNumber;     // Присваивание уникального
  // номера счета.
  Inc(AccountNumber);
  fCustomer := Customer;
  Customer.AttachAccount(Self); // Сообщает клиенту о создании счета.
end;

destructor TAccount.Destroy;
begin
  // Если конструктор прерывается до инициализации fCustomer, это
  // поле остается нулевым. Освобождение счета требуется только
  // если fCustomer не равно nil.
  if Customer <> nil then
    Customer.ReleaseAccount(Self);
  // Вызов TObject.Destroy.
  inherited Destroy;
end;
```

```
const
  DefaultInterestRate = 5000;  // 5%, умноженное на 1000

constructor TSavingsAccount.Create(Customer: TCustomer);
begin
  // Вызов «братского» конструктора.
  Create(Customer, DefaultInterestRate);
end;

constructor TSavingsAccount(Customer: TCustomer; InterestRate:Integer);
begin
  // Вызов TAccount.Create.
  inherited Create(Customer);
  fInterestRate := InterestRate;
end;
```

Деструкторы

Деструкторы, так же как и конструкторы, получают дополнительный скрытый параметр. Первый вызов деструктора передает в дополнительном параметре значение True, по которому Delphi «понимает», что необходимо вызвать `FreeInstance` для освобождения объекта. Если деструктор вызывает унаследованный деструктор, Delphi передает в скрытом параметре значение False, чтобы предотвратить попытку повторного освобождения того же объекта.

В классе обычно имеется один деструктор с названием `Destroy`. Delphi позволяет объявить и дополнительные деструкторы, но это не принесет вам никакой пользы, лишь создаст путаницу.

Перед запуском тела деструктора вызывается виртуальный метод `BeforeDestruction`. Вы можете переопределить `BeforeDestruction`, чтобы изменить состояние программы или выполнить другие действия, которые должны завершиться до начала работы конструктора. Это позволяет написать код класса более корректным образом, не беспокоясь о том, как деструктор базового класса будет вызываться производными классами.

Иногда при написании класса вам может потребоваться перекрыть деструктор `Destroy`, но не переопределяйте метод `Free`. При освобождении объекта следует вызывать именно метод `Free`, а не деструктор. Разница имеет значение, т. к. `Free` проверяет, не является ли ссылка на объект нулевой, и вызывает `Destroy` только для ненулевой ссылки. В особых случаях класс может переопределять метод `Free` (как, например, в классе `TInterface` в редко используемом модуле `VirtIntf`), что делает вызов `Free` вместо `Destroy` еще более актуальным.

Если конструктор или метод AfterConstruction создает исключительную ситуацию, Delphi автоматически вызывает деструктор объекта. Создавая деструктор, следует помнить, что уничтожаемый объект может быть не полностью проинициализирован. Delphi может гарантировать, что все поля до начала работы конструктора равны нулю, но если исключительная ситуация возникает в середине его работы, некоторые поля могут быть проинициализированы, а некоторые – оставаться нулевыми. Если деструктор лишь освобождает объекты и указатели, можете не беспокоиться, т. к. и метод Free, и процедура FreeMem проверяют указатели на nil. Если же ваш деструктор вызывает другие методы, всегда выполняйте проверку на нулевые указатели.

Жизненный цикл объекта

Для большинства объектов сначала вызывается деструктор, затем объект используется и в конце освобождается методом Free. Delphi выполняет за вас все мелочи. Однако иногда полезно знать больше о внутренних механизмах объектной модели Delphi. В примере 2-8 приведены методы, которые Delphi вызывает или эмулирует при создании и освобождении объекта.

Пример 2-8. Жизненный цикл объекта

```
type
  TSomething = class
    procedure DoSomething;
  end;
var
  Ref: TSomething;
begin
  Ref := TSomething.Create;
  Ref.DoSomething;
  Ref.Free;
end;

// Скрытый код конструктора выглядит примерно так:
function TSomething.Create(IsClassRef: Boolean): TSomething;
begin
  if IsClassRef then
  try
    // Allocate the new object.
    Self := TSomething.NewInstance;

    // Метод NewInstance инициализирует объект так же, как это
    // делает InitInstance. Если вы переопределите NewInstance
    // и не вызовете унаследованный NewInstance, обязательно
    // вызовите InitInstance. Этот вызов показан ниже, чтобы
    // вы знали, что происходит, но фактически Delphi не вызывает
     // InitInstance.
    InitInstance(Self);
```

```
    // Выполнение реальной работы конструктора, но без
    // накладных расходов, связанных со ссылкой на класс. Delphi
    // фактически не выполняет рекурсивный вызов конструктора.
    Self.Create(False);

    Self.AfterConstruction;
  except
    // Если происходит исключительная ситуация, Delphi автоматически
    // вызывает деструктор объекта.
    Self.Destroy;
  end
  else
    Self.Create(False);
  Result := Self;
end;

// Скрытый код деструктора выглядит примерно так:
procedure TSomething.Destroy(Deallocate: Boolean);
begin
  if Deallocate then
    Self.BeforeDestruction;

  // Delphi фактически не вызывает деструктор рекурсивно, но именно
  // здесь выполняется реальная работа деструктора.
  Self.Destroy(False);

  if Deallocate then
  begin
    // Delphi фактически не вызывает CleanupInstance. Вместо этого
    // очистку выполняет метод FreeInstance. Если вы переопределяете
    // FreeInstance и не вызываете унаследованный FreeInstance, то
    // для очистки строк, динамических массивов и полей типа Variant
    // нужно вызвать CleanupInstance.
    Self.CleanupInstance;
    // Вызов FreeInstance для освобождения памяти объекта.
    Self.FreeInstance;
  end;
end;
```

Уровни доступа

Как в C++ и Java, в Delphi имеются различные уровни доступа, которые определяют, какие объекты могут иметь доступ к полям, методам и свойствам другого объекта. Существуют следующие уровни доступа:

private

Объявления, обозначенные директивой `private` («закрытые»), доступны только собственным методам класса, а также всем методам, процедурам и функциям, определенным в секции реализации того же модуля. В Delphi отсутствуют аналогичные C++ «дружественные» объявления или аналогичный Java доступ на уровне класса. В Delphi для этого можно объявить такие классы в том же модуле,

тем самым дав им доступ к секциям private и protected определенных в модуле классов.

protected

Объявленные в секции protected («защищенные») поля, методы и свойства доступны из любого метода данного класса и его потомков. Производные классы могут находиться в разных модулях.

public

Public («открытые» или «общедоступные») методы предоставляют неограниченный уровень доступа. Все методы, функции и процедуры имеют доступ к полям и методам, объявленным в секции public. Если не используется директива компилятора $M+ (смотрите подробнее в главе 8 «Директивы компиляторов»), по умолчанию принимается уровень доступа public.

published

Секции published («публикуемые») аналогичны секциям public, за исключением того, что для объявлений секции published создается информация времени выполнения. Не все объявления могут быть помещены в секцию published (смотрите подробности в главе 3). Если данный класс или его базовый класс компилировались с директивой $M+, уровень доступа по умолчанию – published.

IDE Delphi помещает объявления полей и методов в первую неименованную секцию описания формы. Поскольку класс TForm является наследником TPersistent, который использует директиву $M+, для первой секции установлен уровень доступа published. Когда Delphi загружает описание формы из *dfm*-файла, информация для построения объекта формы берется из секции published. IDE использует начальную, неименованную секцию класса. Если вы внесете изменения в эту секцию, есть риск нарушить работу редактора форм с этим классом.

automated

Объявления automated («автоматические») аналогичны public, за исключением того, что для них Delphi формирует дополнительную информацию времени выполнения для поддержки работы серверов OLE-автоматизации. Объявления automated – устаревшая технология; вместо этого следует использовать редактор библиотеки типов, но пока директива automated сохранена в языке для обратной совместимости. Из будущих версий Delphi она может быть исключена. Более подробно секции automated описаны в главе 3.

Производный класс позволяет увеличить уровень доступа свойства путем его повторного объявления с другим уровнем доступа (например, можно изменить protected на public). Уменьшить уровень доступа свойства нельзя, также нельзя изменить уровень доступа для поля или метода. Можно переопределить виртуальный метод и объявить новый

метод как с тем же, так и с более высоким уровнем доступа, но уменьшить уровень доступа нельзя.

Свойства

Свойство выглядит как поле, но работает как метод. Свойства заменяют методы для доступа и изменения полей класса (иногда называемые считыватели (getters) и установщики (setters)), но обладают значительно большей гибкостью и мощностью. Свойства являются жизненно важной частью IDE Delphi и используются во многих других ситуациях.

Для каждого свойства определены считыватели и установщики, позволяющие получать и устанавливать их значения. Считыватель может быть именем поля, некоторой частью сложного поля или методом, возвращающим значение свойства. Вы можете создать свойство только для чтения, не указывая установщика, и свойство только для записи, не указывая считывателя, хотя применение такой конструкции несколько ограничено. Отсутствие и считывателя, и установщика не имеет смысла, и Delphi не позволяет это сделать.

Чаще всего в качестве считывателя и установщика указываются поля и имена методов, но вы также можете ссылаться на часть сложного поля (записи или массива). Если считыватель или установщик ссылаются на элемент массива, индекс массива должен быть константой и поле не может быть динамическим массивом. Записи и массивы могут быть вложенными, и вы даже можете использовать вариантные записи. В примере 2-9 приведен расширенный тип прямоугольника, аналогичный типу `TRect` из Windows, но являющийся классом и имеющий свойства и методы.

Пример 2-9. Считыватели и установщики свойств

```
TRectEx = class(TPersistent)
  private
    R: TRect;
    function GetHeight: Integer;
    function GetWidth: Integer;
    procedure SetHeight(const Value: Integer);
    procedure SetWidth(const Value: Integer);
  public
    constructor Create(const R: TRect); overload;
    constructor Create(Left, Top, Right, Bottom: Integer); overload;
    constructor Create(const TopLeft, BottomRight: TPoint); overload;

    procedure Assign(Source: TPersistent); override;

    procedure Inflate(X, Y: Integer);
    procedure Intersect(const R: TRectEx);
    function IsEmpty: Boolean;
```

Скрытый конструктор

Иногда класс может быть не предназначен для общего пользования, а являться вспомогательным, полностью подчиненным другому классу. В этом случае вы, возможно, захотите сделать конструктор вспомогательного класса закрытым или защищенным, что сделать не так просто. В `TObject` объявляется открытый конструктор `Create`. Даже если конструкторы вспомогательного класса являются закрытыми или защищенными, остается возможность вызвать открытый конструктор, унаследованный от `TObject`.

И хотя вы не можете изменить уровень доступа унаследованного конструктора `Create`, можно скрыть его другим открытым конструктором. Поскольку производный конструктор не предназначен для запуска, он может вызывать исключительную ситуацию. Например:

```
type
  TPublic = class;
  TPrivateHelper = class
  private
    // TPublic - это единственный класс, которому
    // разрешен вызов реального конструктора:
    constructor Create(Owner: TPublic);
      overload;
  public
    // Скрывает TObject.Create на тот случай, если кто-то
    // попытается создать экземпляр класса TPrivateHelper.
    constructor Create;
      reintroduce; overload;
  end;
  TPublic = class
  private
    fHelper: TPrivateHelper;
  public
    constructor Create;
    destructor Destroy;
  end;

constructor TPrivateHelper.Create;
begin
  raise Exception.Create('Ошибка программирования')
end;

constructor TPublic.Create;
begin
  // This is the only place where
  // TPrivateHelper is created.
  // Единственное место, где создается TPrivateHelper
  fHelper := TPrivateHelper.Create(Self);
end;
```

```
    function IsEqual(const R: TRectEx): Boolean;
    procedure Offset(X, Y: Integer);
    procedure Union(const R: TRectEx);

    property TopLeft: TPoint read R.TopLeft write R.TopLeft;
    property BottomRight: TPoint read R.BottomRight write R.BottomRight;
    property Rect: TRect read R write R;
    property Height: Integer read GetHeight write SetHeight;
    property Width: Integer read GetWidth write SetWidth;
  published
    property Left: Integer read R.Left write R.Left default 0;
    property Right: Integer read R.Right write R.Right default 0;
    property Top: Integer read R.Top write R.Top default 0;
    property Bottom: Integer read R.Bottom write R.Bottom default 0;
  end;
```

Свойства-массивы

Свойства могут быть либо скалярами, либо массивами. Свойство-массив не может быть публикуемым (published), но может использоваться во многих других ситуациях. Индекс массива может иметь любой тип, допускаются также многомерные свойства-массивы. Для свойств-массивов следует применять методы-считыватели и методы-установщики – вы не можете установить прямое соответствие между свойством типа «массив» и полем типа «массив».

Одно из свойств-массивов можно назначить как свойство по умолчанию. На свойство по умолчанию можно ссылаться с помощью указания объекта и индекса массива без указания имени свойства, как в примере 2-10.

Пример 2-10. Использование свойства-массива по умолчанию

```
type
  TExample = class
    ...
    property Items[I: Integer]: Integer read GetItem write SetItem;
    property Chars[C: Char]: Char read GetChar write SetChar; default;
  end;
var
  Example: TExample;
  I: Integer;
  C: Char;
begin
  Example := TExample.Create;
  I := Example.Items[4];     // Имя свойства должно быть указано явно
  C := Example['X'];         // Свойство-массив по умолчанию
  C := Example.Chars['X'];   // Аналогично предыдущей строке
```

Индексированные свойства

Вы можете связать несколько свойств с одним считывателем или установщиком путем указания номера индекса для каждого из таких

свойств. Значение индекса передается в считыватель и установщик для указания различия между свойствами.

Допускается совместное использование индексов массивов и индексов свойств. Считыватели и установщики принимают в качестве первого аргумента индекс массива, а в качестве второго – индекс свойства.

Значения по умолчанию

Свойства могут также иметь директивы `stored` и `default`. Эта информация не имеет никакого семантического значения в языке Delphi Pascal, но используется IDE для хранения описания форм. Значения директивы `stored` – это логическая константа, поле типа `Boolean` или метод, не имеющий аргументов и возвращающий логическое значение. Значение директивы `default` – константа того же типа, что и тип свойства. Значение по умолчанию могут иметь только свойства перечисляемого, целого или множественного типов. Директивы `stored` и `default` имеют смысл только для публикуемых свойств.

Чтобы отличить массив по умолчанию от значения по умолчанию, директиву `default` для массива располагают после точки с запятой, завершающей объявление свойства. Значение по умолчанию записывается как часть объявления свойства. Обратитесь к главе 5 за более подробной информацией о директиве `default`.

Использование свойств

Обычный подход при написании классов в Delphi – это объявление всех полей закрытыми и создание для доступа к полям открытых свойств. В Delphi применение свойств для непосредственного обращения к полям не приводит к уменьшению производительности. Используя свойства, вы получаете возможность в будущем изменить внутреннюю реализацию класса, например добавить проверку на правильность устанавливаемых значений поля. При помощи свойств также можно устанавливать ограничения доступа, например, использовать свойство только для чтения для доступа к полю, значение которого нельзя изменять. В примере 2-11 показано несколько способов объявления и применения свойств.

Пример 2-11. Объявление и использование свойств

```
type
  TCustomer = record
    Name: string;
    TaxIDNumber: string[9];
  end;
  TAccount = class
  private
    fCustomer: TCustomer;
    fBalance: Currency;
    fNumber: Cardinal;
    procedure SetBalance(NewBalance: Currency);
```

```
  published
    property Balance: Currency read fBalance write SetBalance;
    property Number: Cardinal read fNumber;   // Номер счета нельзя менять
    property CustName: string read fCustomer.Name;
  end;
  TSavingsAccount = class(TAccount)
  private
    fInterestRate: Integer;
  published
    property InterestRate: Integer read fInterestRate
        write fInterestRate default DefaultInterestRate;
  end;
  TLinkedAccount = class(TObject)
  private
    fAccounts: array[0..1] of TAccount;
    function GetAccount(Index: Integer): TAccount;
  public
    // Два метода доступа свойств к массиву: с помощью индекса
    // и ссылки на элемент массива.
    property Checking: TAccount index 0 read GetAccount;
    property Savings:  TAccount read fAccounts[1];
  end;
  TAccountList = class
  private
    fList: TList;
    function GetAccount(Index: Integer): TAccount;
    procedure SetAccount(Index: Integer; Account: TAccount);
    function GetCount: Integer;
  protected
    property List: TList read fList;
  public
    property Count: Integer read GetCount;
    property Accounts[Index: Integer]: TAccount read GetAccount
        write SetAccount; default;
  end;

procedure TAccount.SetBalance(NewBalance: Currency);
begin
  if NewBalance < 0 then
    raise EOverdrawnException.Create;
  fBalance := NewBalance;
end;

function TLinkedAccount.GetAccount(Index: Integer): TAccount;
begin
  Result := fAccounts[Index]
end;

function TAccountList.GetCount: Integer;
begin
  Result := List.Count
end;
```

```
function TAccountList.GetAccount(Index: Integer): TAccount;
begin
  Result := List[Index]
end;

procedure TAccountList.SetAccount(Index: Integer; Account: TAccount);
begin
  fList[Index] := Account
end;
```

Объекты-свойства

Свойства, содержащие значение типа «объект», требуют дополнительного рассмотрения. Лучше всего работать со объектами-свойствами таким образом, чтобы объект-владелец сам занимался их управлением. Другими словами, не сохраняйте ссылки на другие объекты, а держите собственную копию объекта-свойства. Используйте установщик для сохранения объекта путем его копирования. IDE Delphi требует такого поведения от публикуемых свойств, то же самое имеет смысл и для всех остальных объектов-свойств.

Единственное исключение из правила работы с объектами-свойствами – это использование их для хранения ссылки на компонент из формы. В этом случае свойство должно содержать ссылку на объект, а не копию компонента.

IDE Delphi хранит ссылки на компоненты в *dfm*-файле в виде имени компонента. Когда загружается *dfm*-файл, Delphi ищет имя компонента для восстановления ссылки на объект. Если вы должны хранить весь компонент внутри другого компонента, придется перечислить все свойства внутреннего компонента.

Наследуйте класс объекта-свойства от `TPersistent` и переопределите метод `Assign`. Реализуйте установщик вашего свойства с помощью вызова метода `Assign`. (Использование класса `TPersistent` из модуля `Classes` – это не обязательный, но наиболее простой способ копирования объекта. Иначе вам придется продублировать метод `Assign` во всех используемых классах.) Считыватель может обеспечить непосредственный доступ к полю. Если класс объекта-свойства имеет событие `OnChange`, есть смысл переписать его так, чтобы объект-хозяин узнавал обо всех изменениях. В примере 2-12 показана типичная схема работы с объектом-свойством. В примере определяется графический элемент, который периодически выводит картинку внутри себя, при необходимости дублируя ее в виде мозаики. В свойстве `Bitmap` хранится объект `TBitmap`.

Пример 2-12. Объявление и использование объекта-свойства

```
unit Tile;

interface
```

```
uses SysUtils, Classes, Controls, Graphics;

type
  // Создание мозаики
  TTile = class(TGraphicControl)
  private
    fBitmap: TBitmap;
    procedure SetBitmap(NewBitmap: TBitmap);
    procedure BitmapChanged(Sender: TObject);
  protected
    procedure Paint; override;
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
  published
    property Align;
    property Bitmap: TBitmap read fBitmap write SetBitmap;
    property OnClick;
    property OnDblClick;
    // Многие другие свойства также могут быть полезны, но здесь пропущены
    // для экономии места. Полный список можно найти в описании TControl.
  end;

implementation

{ TTile }

// Создание TBitmap при формировании управляющего элемента.
constructor TTile.Create(Owner: TComponent);
begin
  inherited;
  fBitmap := TBitmap.Create;
  fBitmap.OnChange := BitmapChanged;
end;

// Освобождение TBitmap при уничтожении управляющего элемента.
destructor TTile.Destroy;
begin
  FreeAndNil(fBitmap);
  inherited;
end;

// При изменении картинки управляющий элемент перерисовывается.
procedure TTile.BitmapChanged(Sender: TObject);
begin
  Invalidate;
end;

// Рисование управляющего элемента путем создания мозаики из картинки.
// Если картинки нет, ничего не рисуется.
procedure TTile.Paint;
```

```
var
  X, Y: Integer;
begin
  if (Bitmap.Width = 0) or (Bitmap.Height = 0) then
    Exit;

  Y := 0;
  while Y < ClientHeight do
  begin
    X := 0;
    while X < ClientWidth do
    begin
      Canvas.Draw(X, Y, Bitmap);
      Inc(X, Bitmap.Width);
    end;
    Inc(Y, Bitmap.Height);
  end;
end;

// Создание новой картинки копированием объекта TBitmap.
procedure TTile.SetBitmap(NewBitmap: TBitmap);
begin
  fBitmap.Assign(NewBitmap);
end;

end.
```

Интерфейсы

Интерфейс определяет тип, состоящий из абстрактных виртуальных методов[1]. И хотя класс может быть наследником только одного базового класса, он может реализовывать любое количество интерфейсов. Интерфейс имеет некоторое сходство с абстрактным классом (т. е. классом без полей, все методы которого абстрактные), но в Delphi есть особые приемы для работы с интерфейсами. Интерфейсы Delphi в чем-то напоминают интерфейсы COM (Component Object Model, объектная модель компонентов), но для использования интерфейсов Delphi не требуется знания COM, и их можно применять для многих других целей. Вы можете объявить новый интерфейс как наследник существующего. Объявление интерфейса может содержать описания методов и свойств, но не может содержать описания полей. Как все классы в Delphi являются потомками `TObject`, так и все интерфейсы в Delphi явля-

[1] Помимо методов, интерфейс может содержать также и свойства (интерфейс не может содержать поля). Следует заметить, что поскольку интерфейс лишь объявляет, но не реализует методы, применение директивы `abstract` в объявлении методов интерфейса лишено смысла и не допускается. – *Примеч. науч. ред.*

ются расширениями IUnknown. В интерфейсе IUnknown объявлено три метода: _AddRef, _Release и QueryInterface. Если вы знаете COM, они вам знакомы. Первые два метода управляют счетчиком ссылок для определения времени жизни объекта, реализующего интерфейс. Третий метод обеспечивает доступ к другим интерфейсам, которые реализует объект.

Если вы объявляете класс, реализующий один и более интерфейсов, вы должны обеспечить реализацию всех методов, объявленных во всех интерфейсах. Класс может либо реализовать методы интерфейса, либо делегировать реализацию свойству, значение которого является интерфейсом. Простейший способ реализации методов _AddRef, _Release и QueryInterface – это унаследовать их от TInterfacedObject или одного из его производных классов, но вы также можете использовать другие базовые классы, если хотите создать эти методы самостоятельно.

Класс реализует каждый из методов интерфейса путем объявления метода с тем же именем, аргументами и соглашением о вызове. Delphi автоматически ставит методы класса в соответствие с методами интерфейса. Если вы хотите дать методу другое имя, вы можете перенаправить на него метод интерфейса. Он должен иметь те же аргументы и соглашение о вызове, что и метод интерфейса. Эта возможность особенно важна, когда класс реализует несколько интерфейсов, содержащих методы с одинаковыми именами. Подробнее о перенаправлении методов смотрите описание ключевого слова class в главе 5 .

Класс может делегировать реализацию интерфейса свойству, содержащему директиву implements. Тип значения свойства должен соответствовать интерфейсу, который класс собирается реализовать. Когда объект преобразуется к этому типу интерфейса, Delphi автоматически извлекает значение свойства и возвращает соответствующий интерфейс. Смотрите описание директивы implements в главе 5.

Для каждого неделегированного интерфейса компилятор создает скрытое поле, в котором хранится указатель на VMT интерфейса. Поле или поля интерфейса следуют сразу за скрытыми VMT-полями объекта. Так же как ссылка на объект является указателем на скрытое VMT-поле объекта, так и ссылка на интерфейс является указателем на скрытое VMT-поле интерфейса. Delphi автоматически инициализирует скрытые поля при создании объекта. В главе 3 обсуждается, как компилятор использует RTTI для отслеживания VMT и скрытого поля.

Счетчик ссылок

Компилятор генерирует для управления временем жизни интерфейсных объектов вызовы методов _AddRef и _Release. Для использования автоматического счетчика ссылок объявите переменную интерфейсного типа. Когда происходит присваивание интерфейсной переменной, Delphi автоматически вызывает метод _AddRef, а когда переменная выходит из области действия, – метод _Release.

Поведение _AddRef и _Release полностью зависит от вас. Если вы наследуете объект от TInterfacedObject, они реализуют счетчик ссылок, причем _AddRef увеличивает счетчик ссылок, а _Release – уменьшает его. Когда счетчик уменьшается до 0, _Release освобождает объект. Если вы используете другой класс в качестве базового, то эти методы можно применять для любых целей. Однако вы должны корректно реализовать метод QueryInterface, так как Delphi использует его для реализации работы оператора as.

Преобразования типов

Delphi вызывает метод QueryInterface как часть реализации оператора as для интерфейса. При помощи оператора as можно преобразовать один интерфейс к любому другому интерфейсу. Для получения ссылки на новый интерфейс Delphi вызывает QueryInterface. Если в ходе выполнения последнего возникает ошибка, оператор as генерирует ошибку времени выполнения. (Модуль SysUtils преобразует ошибку времени выполнения в исключительную ситуацию EIntfCastError.)

Вы можете реализовать QueryInterface по собственному усмотрению, но вероятно последуете тому подходу, который использован в TInterfacedObject. В примере 2-13 приведен класс, который правильно реализует метод QueryInterface, но для _AddRef и _Release в нем оставлены «заглушки». Далее в этой главе вы убедитесь, насколько полезен может быть такой класс.

Пример 2-13. Базовый интерфейсный класс без счетчика ссылок

```
type
  TNoRefCount = class(TObject, IUnknown)
  protected
    function QueryInterface(const IID:TGUID; out Obj):HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;

function TNoRefCount.QueryInterface(const IID:TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := 0
  else
    Result := Windows.E_NoInterface;
end;

function TNoRefCount._AddRef: Integer;
begin
  Result := -1
end;

function TNoRefCount._Release: Integer;
begin
  Result := -1
end;
```

Интерфейсы и объектно-ориентированное программирование

Самый полезный вариант применения интерфейсов – это отделение наследования типов от наследования классов. Наследование классов – это эффективное средство повторного использования кода. Производный класс может наследовать поля, методы и свойства базового класса и, следовательно, не реализовывать общие методы заново. В строго типизированном языке, таком как Delphi Pascal, компилятор рассматривает класс как тип, и, таким образом, наследование классов становится синонимом наследования типов. Но в лучшем из возможных миров, однако, типы и классы должны быть полностью разделены.

В учебниках по объектно-ориентированному программированию наследование часто описывается как отношение «is» («это»), например, `TSavingsAccount` «это» `TAccount`. Та же идея прослеживается и в операторе `is` (это), с помощью которого вы можете проверить, что переменная `Account – is TSavingsAccount`.

Но за пределами учебников, простая идея отношения «это» не всегда работает. Квадрат «это» прямоугольник, но это не значит, что вы захотите создать `TSquare` как класс, производный от `TRectangle`. Прямоугольник «это» многоугольник, но вы не обязательно создадите класс `TRectangle` на основе `TPolygon`. Наследование классов требует, чтобы в производном классе были все поля, объявленные в базовом, но в описанных случаях производному классу это не требуется. Объект `TSquare` может обойтись без хранения значений длины каждой из своих сторон. Однако объект `TRectangle` должен иметь две длины. Объекту `TPolygon` требуется хранить информацию о многих сторонах и вершинах. Решение состоит в отделении наследования типов (квадрат «это» прямоугольник, прямоугольник «это» многоугольник) от наследования классов (класс C наследует поля и методы класса B, который наследует поля и методы класса A). Для наследования типов можно использовать интерфейсы и оставить наследованию классов то, что у него получается лучше всего – наследование полей и методов.

Другими словами, `ISquare` является наследником `IRectangle`, который является наследником `IPolygon`. Для наследования интерфейсов выполняется отношение «это». Независимо от наследования интерфейсов, класс `TSquare` реализует `ISquare`, `IRectangle` и `IPolygon`. `TRectangle` реализует `IRectangle` и `IPolygon`.

В программировании технологии COM принято соглашение, что названия интерфейсов начинаются с буквы `I`. Delphi следует этому соглашению для всех интерфейсов. Заметьте, что это лишь полезное соглашение, а не требование языка.

Что касается реализации повторного использования кода, то вы можете объявить для этого дополнительные классы. Например, класс `TBaseShape` реализует методы и поля, общие для всех фигур. `TRectangle` является наследником `TBaseShape` и реализует методы тем способом, кото-

рый имеет смысл для прямоугольников. TPolygon также является наследником TBaseShape и реализует свои методы способом, подходящим для всех остальных видов многоугольников. Программа рисования может использовать эти классы, работая через интерфейс IPolygon. В примере 2-14 показаны упрощенные классы и интерфейсы, описанные этой схемой. Заметьте, что каждый интерфейс имеет в своем описании GUID (Globally Unique Identifier – глобально уникальный идентификатор). GUID необходим для использования метода QueryInterface. Если вам потребуется GUID (например, для непосредственного вызова QueryInterface), можно использовать имя интерфейса. Delphi автоматически преобразует имя интерфейса в его GUID.

Пример 2-14. Разделение иерархий типов и классов

```
type
  IShape = interface
  ['{50F6D851-F4EB-11D2-88AC-00104BCAC44B}']
    procedure Draw(Canvas: TCanvas);
    function GetPosition: TPoint;
    procedure SetPosition(Value: TPoint);
    property Position: TPoint read GetPosition write SetPosition;
  end;

  IPolygon = interface(IShape)
  ['{50F6D852-F4EB-11D2-88AC-00104BCAC44B}']
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  IRectangle = interface(IPolygon)
  ['{50F6D853-F4EB-11D2-88AC-00104BCAC44B}']
  end;
  ISquare = interface(IRectangle)
  ['{50F6D854-F4EB-11D2-88AC-00104BCAC44B}']
    function Side: Integer;
  end;

  TBaseShape = class(TNoRefCount, IShape)
  private
    fPosition: TPoint;
    function GetPosition: TPoint;
    procedure SetPosition(Value: TPoint);
  public
    constructor Create; virtual;
    procedure Draw(Canvas: TCanvas); virtual; abstract;
    property Position: TPoint read fPosition write SetPosition;
  end;
  TPolygon = class(TBaseShape, IPolygon)
  private
    fVertices: array of TPoint;
```

```
  public
    procedure Draw(Canvas: TCanvas); override;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  TRectangle = class(TBaseShape, IPolygon, IRectangle)
  private
    fRect: TRect;
  public
    procedure Draw(Canvas: TCanvas); override;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
  TSquare = class(TBaseShape, IPolygon, IRectangle, ISquare)
  private
    fSide: Integer;
  public
    procedure Draw(Canvas: TCanvas); override;
    function Side: Integer;
    function NumVertices: Integer;
    function NumSides: Integer;
    function SideLength(Index: Integer): Integer;
    function Vertex(Index: Integer): TPoint;
  end;
```

Производный класс наследует интерфейсы, реализованные родительскими классами. То есть, если `TRectangle` является наследником `TBaseShape`, и `TBaseShape` реализует `IShape`, то `TRectangle` реализует `IShape`. Наследование интерфейсов работает несколько по-другому, это лишь средство для сокращения набора текста, чтобы не приходилось заново перепечатывать длинные описания методов. Если класс реализует интерфейс, это не значит, что он автоматически реализует все его родительские интерфейсы. Класс реализует только те интерфейсы, которые перечислены в его описании (и описаниях классов-предков). Поэтому хотя `IRectangle` — наследник `IPolygon`, если вы хотите, чтобы класс `TRectangle` реализовывал интерфейсы `IRectangle` и `IPolygon`, в объявлении класса `TRectangle` должны быть явно указаны и `IRectangle`, и `IPolygon`.

При создании иерархии типов вы, возможно, не захотите использовать счетчик ссылок. Вместо этого можно положиться на явное управление памятью, как для обычных объектов Delphi. В этом случае лучше всего реализовать методы `_AddRef` и `_Release` как заглушки, как это сделано в классе `TNoRefCount` в примере 2-13. Только не храните в других переменных устаревшие ссылки. Переменная, которая ссылается на уже уничтоженный объект, наверняка создаст вам проблемы, когда

Delphi автоматически вызовет для нее метод _Release. Другими словами, в программе не должно быть переменных, содержащих неверные указатели, а при работе с интерфейсами, не использующими средства подсчета ссылок, вы просто вынуждены так поступать.

COM и CORBA

Интерфейсы Delphi также полезны для использования и реализации объектов COM и CORBA. Вы можете создать сервер COM, реализующий много интерфейсов, и Delphi автоматически будет управлять агрегированием. Библиотека Delphi содержит много классов, облегчающих создание серверов COM, «фабрик классов» и прочего. Поскольку эти классы не являются частью языка Delphi Pascal, они не описываются в этой книге. Смотрите фирменную документацию по Delphi об информации по этим классам.

Счетчики ссылок

В предыдущем разделе мы обсуждали управление жизненным циклом интерфейсов при помощи счетчика ссылок. Строки и динамические массивы также используют для этого счетчики ссылок. Компилятор генерирует необходимый код, отслеживающий создание ссылок на интерфейсы, строки и динамические массивы и выход их из области действия, когда они должны быть уничтожены.

Обычно компилятор управляет счетчиками ссылок автоматически, и все работает именно так, как можно было бы ожидать. Иногда, однако, вы должны дать компилятору указание. Например, когда вы объявляете запись, которая содержит поле, имеющее счетчик ссылок, и используете процедуру GetMem для создания экземпляра записи, следует вызвать процедуру Initialize, передав ей данную запись в качестве аргумента. Перед вызовом FreeMem следует вызвать Finalize.

Возможно, вы захотите сохранить ссылку на строку или интерфейс после выхода за границы ее действия, т. е. вне блока, где была объявлена переменная. Например, связать интерфейс с каждым элементом TListView. Это можно сделать, явно управляя счетчиком ссылок. При сохранении интерфейса преобразуйте его в тип IUnknown, вызовите _AddRef и преобразуйте ссылку на IUnknown в простой тип указателя. При извлечении данных преобразуйте указатель в IUnknown. Таким образом, вы сможете использовать оператор as для преобразования интерфейса к любому типу, а также дать возможность Delphi освободить интерфейс. Для удобства создайте пару подпрограмм, выполняющих всю «грязную» работу, и пользуйтесь ими всегда, когда вам потребуется восстановить ссылку на интерфейс. В примере 2-15 приведен пример сохранения ссылки на интерфейс как информации, привязанной к элементу списка.

Пример 2-15. Сохранение интерфейсов вместе со списком

```
// Преобразование интерфейса в Pointer, чтобы увеличить
// счетчик ссылок и не освобождать интерфейс до вызова
// метода ReleaseIUnknown.
function RefIUnknown(const Intf: IUnknown): Pointer;
begin
  Intf._AddRef;               // Увеличение счетчика ссылок.
  Result := Pointer(Intf);    // Сохранение указателя на интерфейс.
end;

// Освобождение интерфейса, указатель на который хранится в P.
procedure ReleaseIUnknown(P: Pointer);
var
  Intf: IUnknown;
begin
  Pointer(Intf) := P;
  // Delphi освобождает интерфейс, когда Intf выходит из области действия
end;

// Когда пользователь нажимает на кнопку, интерфейс добавляется в список.
procedure TForm1.Button1Click(Sender: TObject);
var
  Item: TListItem;
begin
  Item := ListView1.Items.Add;
  Item.Caption := 'Stuff';
  Item.Data := RefIUnknown(GetIntf as IUnknown);
end;

// Когда по какой-либо причине уничтожается список или один из его элементов,
// интерфейс также освобождается.
procedure TForm1.ListView1Deletion(Sender: TObject; Item: TListItem);
begin
   ReleaseIUnknown(Item.Data);
end;

// Когда пользователь выбирает элемент из списка, выполняется некоторое
// действие со связанным с этим элементом интерфейсом.
procedure TForm1.ListView1Click(Sender: TObject);
var
  Intf: IMyInterface;
begin
  Intf := IUnknown(ListView1.Selected.Data) as IMyInterface;
  Intf.DoSomethingUseful;
end;
```

Вы можете также сохранять и строки. Вместо вызова `_AddRef` **преобразуйте строку к типу «указатель», чтобы сохранить ссылку на строку, затем заставьте переменную «забыть» об этой строке. Когда переменная выйдет из своей области действия, Delphi не освободит строку,**

поскольку переменная о ней «не помнит». После получения указателя присвойте его строковой переменной, преобразованной к типу «указатель». При возврате из подпрограммы Delphi автоматически освободит память этой строки. Убедитесь, что в программе не осталось указателей на освобождаемую память. Кроме того, служебные процедуры упростят вашу задачу. В примере 2-16 показан один из способов хранения строк.

Пример 2-16. Хранение строк вместе со списком

```
// Сохранение ссылки на строку и возвращение простого указателя
// на строку.
function RefString(const S: string): Pointer;
var
  Local: string;
begin
  Local := S;                // Увеличение счетчика ссылок.
  Result := Pointer(Local);  // Сохранение указателя на строку.
  Pointer(Local) := nil;     // Предотвращение уменьшения значения
     // счетчика.
end;

// Уничтожение строки, на которую ссылается RefString.
procedure ReleaseString(P: Pointer);
var
  Local: string;
begin
  Pointer(Local) := P;
  // Delphi уничтожает строку, когда заканчивается область действия Local.
end;

// При нажатии пользователем кнопки добавляем элемент в список
// и сохраняем дополнительную скрытую строку.
procedure TForm1.Button1Click(Sender: TObject);
var
  Item: TListItem;
begin
  Item := ListView1.Items.Add;
  Item.Caption := Edit1.Text;
  Item.Data := RefString(Edit2.Text);
end;

// Уничтожает строку при уничтожении элемента списка.
procedure TForm1.ListView1Deletion(Sender: TObject; Item: TListItem);
begin
  ReleaseString(Item.Data);
end;

// Извлечение строки при выборе пользователем элемента списка.
procedure TForm1.ListView1Click(Sender: TObject);
var
```

```
  Str: string;
begin
  if ListView1.Selected <> nil then
  begin
    Str := string(ListView1.Selected.Data);
    ShowMessage(Str);
  end;
end;
```

Сообщения

Вы, должно быть, знакомы с сообщениями Windows: действия пользователя и другие события генерируют сообщения, которые Windows посылает приложению. Приложение по очереди обрабатывает сообщения, реагируя на происходящие события. Каждый тип сообщения имеет уникальный номер и два целых параметра. Иногда параметр фактически является указателем на строку или структуру данных, содержащую более подробную информацию о событии. Сообщения формируют ядро архитектуры Windows, и в Delphi имеется уникальный способ работы с сообщениями Windows. В Delphi любой объект, а не только управляющие элементы, может реагировать на сообщения. Каждое сообщение имеет уникальный целый идентификатор и может содержать любое количество дополнительной информации. В VCL (Visual Component Library) сообщения Windows получает объект `Application`, который преобразует их в эквивалентные сообщения Delphi. Другими словами, сообщения Windows являются особым случаем более общих сообщений Delphi.

Сообщенит Delphi – это запись, первые два байта которой содержат целый идентификатор сообщения, а остаток записи определяется программистом. Диспетчер сообщений Delphi никогда не обращается к оставшейся части сообщения, идущей после номера, поэтому вы можете хранить в этой записи любое количество и тип информации. По соглашению, VCL всегда использует аналогичный Windows формат сообщения (`TMessage`), но если вы сможете использовать сообщения Delphi и для других целей, не ограничивайте себя.

Чтобы послать сообщение объекту, заполните поле идентификатора сообщения и оставшуюся часть записи, а затем вызовите метод `Dispatch` этого объекта. Delphi ищет номер сообщения в таблице сообщений объекта. Эта таблица содержит указатели на все обработчики событий, которые определены в классе. Если для этого сообщения обработчик отсутствует, Delphi продолжает поиск в таблице родительского класса. Поиск продолжается до тех пор, пока не будет найден обработчик, либо поиск не дойдет до класса `TObject`. Если в классе объекта и в его родительских классах не определены обработчики для данного типа сообщений, Delphi вызывает метод `DefaultHandler`. Управляющие элементы из VCL переопределяют метод `DefaultHandler` для передачи

сообщений процедуре окна; другие классы обычно игнорируют неизвестные им сообщения. Вы можете переопределить DefaultHandler для выполнения любых действий, которые вы хотите, например, для вызова исключительной ситуации.

Для объявления обработчика сообщений используется директива message. Подробнее о директиве message смотрите в главе 5.

Обработчики сообщений используют ту же таблицу сообщений и диспетчер, что и динамические сообщения. Каждому методу, который определен с директивой dynamic, присваивается 16-разрядное отрицательное число, которое фактически является номером сообщения. При вызове динамического метода используется тот же алгоритм поиска, что и для обработчиков событий, но если динамический метод не найден, это значит, что он является абстрактным, и Delphi вызывает процедуру AbstractErrorProc для выдачи сообщения о вызове абстрактного метода.

Поскольку динамические методы используют отрицательные индексы, вы не можете написать обработчик сообщения с отрицательным номером, т. е. для номеров сообщений, у которых старший бит равен 1. Это ограничение не должно создать проблем для обычных приложений. Если вам потребуется определить собственные сообщения, для этого имеется пространство от WM_USER ($0F00) до $7FF. Delphi ищет динамические методы и сообщения в одной таблице, используя линейный поиск, и потому ваше приложение будет тратить много времени на нахождение метода в больших таблицах.

Система сообщений Delphi абсолютно универсальна, и можно творчески подойти к ее применению. Обычно интерфейсы предоставляют вам те же возможности, но при лучшей производительности и повышенной безопасности типов.

Управление распределением памяти

Delphi автоматически управляет распределением памяти и жизненным циклом строк, переменных типа Variant, динамических массивов и интерфейсов. Для всех остальных динамических структур данных за это отвечает программист. Здесь легко запутаться, т. к. может показаться, что Delphi также автоматически распределяет память и для компонентов, но это лишь прием VCL.

Управление распределением памяти работает и в многопоточных приложениях в том случае, если вы используете для создания потоков классы или функции Delphi. Если вы будете непосредственно пользоваться средствами Windows API и функцией CreateThread, установите значение переменной IsMultiThread в True. Дополнительную информацию смотрите в главе 4 «Создание многопоточных приложений».

Компоненты и объекты

Класс TComponent из VCL имеет два интересных механизма для управления жизненным циклом объектов, и это может запутать новичка в Delphi, заставляя думать, что Delphi делает это автоматически. Важно понимать, как именно работают компоненты, чтобы не оказаться введенным в заблуждение.

У каждого компонента есть владелец. Когда владелец уничтожается, вместе с ним уничтожаются все имеющиеся у него компоненты. Форма является владельцем всех компонентов, находящихся в ней, и при освобождении она автоматически освобождает все свои компоненты. Поэтому вам обычно не надо беспокоиться об управлении жизненным циклом форм и компонентов.

Когда форма освобождает компонент, которым она владеет, выполняется проверка, нет ли публикуемых полей с тем же именем, что и этот компонент. Если таковые есть, значение поля устанавливается в nil. Поэтому, если форма динамически добавляет и удаляет компоненты, ее поля всегда либо содержат корректную ссылку на объект, либо равны nil. Не думайте, что Delphi делает это для любого другого поля или ссылки на объект. Это делается только для публикуемых полей (например, для тех, которые автоматически создаются при переносе компонента на форму в редакторе форм IDE), и только когда имя поля соответствует имени компонента.

Обычно при создании объекта Delphi вызывает для его распределения и инициализации метод NewInstance. Вы можете переопределить NewInstance для изменения алгоритма выделения памяти для объекта. Предположим, например, что ваша программа часто использует двусвязные списки. Вместо использования универсального менеджера памяти для каждого узла списка значительно эффективнее иметь цепочку доступных для использования узлов и применять менеджер памяти Delphi только тогда, когда список узлов будет исчерпан. Если ваша программа часто выделяет и освобождает узлы, этот специальный менеджер памяти может работать быстрее универсального. В примере 2-17 показана простая реализация этой схемы. (В главе 4 имеется версия этого класса для многопоточного приложения.)

Пример 2-17. Специальный менеджер памяти для связанных списков

```
type
  TNode = class
  private
    fNext, fPrevious: TNode;
  protected
    // Узлы управляются объектом типа TLinkedList.
```

```
    procedure Relink(NewNext, NewPrevious: TNode);
    constructor Create(Next: TNode = nil; Previous: TNode = nil);
    procedure RealFree;

public
    destructor Destroy; override;
    class function NewInstance: TObject; override;
    procedure FreeInstance; override;
    property Next: TNode read fNext;
    property Previous: TNode read fPrevious;
end;

// Односвязный список узлов, доступных для использования.
// Для работы со списком используются только поля Next.
var
  NodeList: TNode;

// Новый узел создается из начального узла NodeList.
// Не забудьте вызвать InitInstance для инициализации узла,
// взятого из Nodelist.
// Если NodeList пустой, создаем узел обычным образом.
class function TNode.NewInstance: TObject;
begin
  if NodeList = nil then
    Result := inherited NewInstance
  else
  begin
    Result := NodeList;
    NodeList := NodeList.Next;
    InitInstance(Result);
  end;
end;

// Так как NodeList использует только поле Next, установите поле
// Previous равным некоторому специальному значению. Если программа
// по ошибке обратится к полю Previous свободного узла, вы, проверив
// это специальное значение, найдете причину ошибки.
const
  BadPointerValueToFlagErrors = Pointer($F0EE0BAD);

// Освобождаемый узел добавляется в начало NodeList.
// Это ЗНАЧИТЕЛЬНО быстрее, чем использовать универсальный
// менеджер памяти.
procedure TNode.FreeInstance;
begin
  fPrevious := BadPointerValueToFlagErrors;
  fNext := NodeList;
  NodeList := Self;
end;

// Если вы хотите корректно очистить список по завершении работы
```

```
// программы, вызовите RealFree для каждого элемента списка.
// Унаследованный метод FreeInstance освобождает и очищает
// узел фактически.
procedure TNode.RealFree;
begin
  inherited FreeInstance;
end;
```

Вы также можете целиком заменить систему управления распределением памяти, используемую Delphi. Новый менеджер памяти устанавливается вызовом процедуры `SetMemoryManager`. Например, вы можете заменить упрощенную версию процедуры выделения памяти на полную версию, осуществляющую дополнительные проверки. В примере 2-18 показан специальный менеджер памяти, который поддерживает список выделенных программе указателей и явно проверяет все попытки их освобождения по этому списку. Любая попытка освободить неверный указатель отклоняется, и Delphi выдает ошибку времени выполнения (которую `SysUtils` заменяет на исключительную ситуацию). Заодно менеджер памяти проверяет, является ли этот список пустым по окончании работы приложения. Если список не пустой, происходит утечка памяти.

Пример 2-18. Установка специального менеджера памяти

```
unit CheckMemMgr;
interface

uses Windows;

function CheckGet(Size: Integer): Pointer;
function CheckFree(Mem: Pointer): Integer;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;

var
  HeapFlags: DWord;       // В однопоточном приложении можно установить
                        // эту переменную в Heap_No_Serialize.
implementation

const
  MaxSize = MaxInt div 4;
type
  TPointerArray = array[1..MaxSize] of Pointer;
  PPointerArray = ^TPointerArray;
var
  Heap: THandle;             // Идентификатор «кучи» Windows для
                             // списка указателей
  List: PPointerArray;       // Список распределенных указателей
  ListSize: Integer;         // Количество указателей в списке
  ListAlloc: Integer;        // Емкость списка указателей

// Если список распределенных указателей не пустой после завершения
```

```
// работы программы, это означает утечку памяти. Работа с утечкой
// памяти оставляется читателю в качестве упражнения.
procedure MemoryLeak;
begin
  // Выдача пользователю сообщения об утечке памяти. Так как программа
  // завершает свою работу, вы ограничены интерфейсом Windows API
  // и не можете использовать VCL.
end;

// Добавление указателя в список.
procedure AddMem(Mem: Pointer);
begin
  if List = nil then
  begin
    // Создание списка.
    ListAlloc := 8;
    List := HeapAlloc(Heap, HeapFlags, ListAlloc * SizeOf(Pointer));
  end
  else if ListSize >= ListAlloc then
  begin
    // Увеличение размера списка. Попытайтесь подойти к этому разумно.
    if ListAlloc < 256 then
      ListAlloc := ListAlloc * 2
    else
      ListAlloc := ListAlloc + 256;
    List := HeapRealloc(Heap, HeapFlags, List,
                        ListAlloc * SizeOf(Pointer));
  end;
  // Добавление указателя в список.
  Inc(ListSize);
  List[ListSize] := Mem;
end;

// Поиск указателя в списке и его удаление. Возвращает истину
// в случае успеха и ложь в случае отсутствия указателя в списке.
function RemoveMem(Mem: Pointer): Boolean;
var
  I: Integer;
begin
  for I := 1 to ListSize do
    if List[I] = Mem then
    begin
      MoveMemory(@List[I], @List[I+1], (ListSize-I) * SizeOf(Pointer));
      Dec(ListSize);
      Result := True;
      Exit;
    end;

  Result := False;
end;
```

```
// Замена процедуры выделения памяти.
function CheckGet(Size: Integer): Pointer;
begin
  Result := SysGetMem(Size);
  AddMem(Result);
end;

// Если указателя нет в списке, реальная функция Free не
// вызывается. Возвращает 0 в случае успеха и ненулевое значение
// в случае ошибки.
function CheckFree(Mem: Pointer): Integer;
begin
  if not RemoveMem(Mem) then
    Result := 1
  else
    Result := SysFreeMem(Mem);
end;

// Удаление старого указателя и добавление нового, который может
// либо быть тем же, либо отличаться. В случае ошибки возвращает nil,
// и Delphi вызывает исключительную ситуацию.
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;
begin
  if not RemoveMem(Mem) then
    Result := nil
  else
  begin
    Result :=SysReallocMem(Mem, Size);
    AddMem(Result);
  end;
end;

procedure SetNewManager;
var
  Mgr: TMemoryManager;
begin
  Mgr.GetMem := CheckGet;
  Mgr.FreeMem := CheckFree;
  Mgr.ReallocMem := CheckRealloc;
  SetMemoryManager(Mgr);
end;

initialization
  Heap := HeapCreate(0, HeapFlags, 0);
  SetNewManager;
finalization
  if ListSize <> 0 then
    MemoryLeak;
  HeapDestroy(Heap);
end.
```

Если вы определяете собственный менеджер памяти, вы должны быть уверены, что он используется для всех операций по распределению памяти. Простейший вариант состоит в установке менеджера памяти в разделе инициализации модуля, как показано в примере 2-18. Модуль управления памятью должен быть первым из перечисленных в разделе `uses` проекта.

Обычно, если модуль выполняет какие-то глобальные изменения в разделе инициализации, он должен вернуть изменения в исходное состояние в разделе завершения. Модуль, находящийся в пакете, может загружаться и выгружаться приложением несколько раз, и поэтому возвращение в исходное состояние имеет большое значение. Но менеджер памяти – это другое дело. Память, распределенная одним менеджером памяти, не может быть освобождена другим, и вы должны гарантировать, что в приложении работает только один менеджер, и он действует в течение всей работы приложения. Это значит, что менеджер памяти нельзя поместить в пакет, хотя, как описывается в следующем разделе, вы можете использовать DLL.

Память и DLL

Если вы используете DLL и хотите передавать объекты между двумя DLL или DLL и приложением, возникает несколько проблем. Во-первых, каждая DLL и исполняемый модуль имеют собственные копии таблиц классов. Операторы `is` и `as` не работают корректно для объектов, передаваемых между DLL и исполняемым модулем. Для решения этой проблемы используйте пакеты (смотрите главу 1). Другая проблема состоит в том, что любая память, выделенная DLL, принадлежит этой библиотеке. Когда Windows выгружает DLL, вся память, выделенная из этой библиотеки, освобождается, даже если в исполняемом модуле или другой DLL имеется указатель на эту память. Это может вызвать серьезные проблемы при использовании строк, динамических массивов и переменных типа `Variant`, так как вы никогда не знаете, когда Delphi автоматически выделит им память.

Решение состоит в использовании модуля `ShareMem` в качестве первого модуля проекта и всех DLL. Модуль устанавливает специальный менеджер памяти, который перенаправляет все запросы на выделение памяти в специальную DLL, *BorlndMM.dll*. Приложение не выгрузит *BorlndMM* до своего завершения. Работа с DLL происходит прозрачным образом, и вам не придется заботиться о деталях. Только не забудьте использовать модуль `ShareMem` и поместить его первым в списке `uses` всех ваших модулей и библиотек. Когда вы будете передавать ваше приложение клиентам или заказчикам, включите в поставку библиотеку *BorlndMM.dll*.

Если вы создаете собственный менеджер памяти и используете DLL, вам придется повторить трюки, выполняемые модулем `ShareMem`. Вы можете также заменить этот модуль на собственный, перенаправля-

ющий запросы к DLL, которая работает с вашим менеджером памяти. В примере 2-19 показан один из способов создания замены модулю ShareMem.

Пример 2-19. Создание менеджера общей памяти

```
unit CheckShareMem;

// Этот модуль должен быть первым, чтобы все запросы к памяти поступали
// к менеджеру общей памяти. Этот модуль должно использовать приложение и
// все DLL. Применение пакетов не допускается, так как эти DLL используют
// стандартный менеджер общей памяти Borland.

interface

function CheckGet(Size: Integer): Pointer;
function CheckFree(Mem: Pointer): Integer;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;

implementation

const
  DLL = 'CheckMM.dll';

function CheckGet(Size: Integer): Pointer; external DLL;
function CheckFree(Mem: Pointer): Integer; external DLL;
function CheckRealloc(Mem: Pointer; Size: Integer): Pointer;
    external DLL;

procedure SetNewManager;
var
  Mgr: TMemoryManager;
begin
  Mgr.GetMem := CheckGet;
  Mgr.FreeMem := CheckFree;
  Mgr.ReallocMem := CheckRealloc;
  SetMemoryManager(Mgr);
end;

initialization
  SetNewManager;
end.
```

Библиотека CheckMM использует ваш менеджер памяти и экспортирует свои функции так, что с ними может работать модуль CheckShareMem. В примере 2-20 приведен исходный текст библиотеки CheckMM.

Пример 2-20. Создание DLL-библиотеки менеджера общей памяти

```
library CheckMM;

// Замена для BorlndMM.dll, использующая собственный менеджер памяти.
```

```
uses
  CheckMemMgr;

exports
  CheckGet, CheckFree, CheckRealloc;

begin
end.
```

Ваши программы и библиотеки начинаются модулем `CheckShareMem`, и все запросы к памяти направляются в библиотеку *CheckMM.dll*, которая использует менеджер памяти с проверкой ошибок. Вам вряд ли потребуется часто менять менеджер памяти Delphi, но, как видите, это не трудно сделать.

Менеджер памяти, имеющийся в Delphi, работает хорошо для большинства приложений, но он может не слишком подходить для вашей задачи. Среднее приложение выделяет и освобождает память участками переменного размера. Если к вашей программе это не относится, и она выделяет память во все увеличивающемся объеме (например, если у вас имеется динамический массив, который вырастает небольшими этапами до очень большого размера), производительность может упасть. Менеджер памяти Delphi будет запрашивать больше памяти, чем требуется приложению. Одно из решений состоит в переработке программы с тем, чтобы она по-другому использовала память (например, заранее выделяла большой динамический массив). Другое решение – написать собственный менеджер памяти, который будет соответствовать специфическим потребностям вашего приложения. Например, новый менеджер памяти может использовать Windows API (`HeapAllocate` и другие подобные процедуры).

Ключевое слово «object»

Кроме классов, Delphi также поддерживает устаревший тип, использующий ключевое слово `object`. Эти объекты сохранены для обратной совместимости с Turbo Pascal, но в последующих версиях Delphi они могут быть исключены.

«Старые» объекты больше похожи на записи, чем на новые объекты, принятые в Delphi Pascal. Поля старых объектов хранятся так же, как в записях. Если объект не имеет виртуальных методов, то он и не имеет скрытого поля для указателя на VMT. В отличие от записей, «старые» объекты также поддерживают наследование. Новые поля следуют за полями, унаследованными от родительских объектов. Если в объекте объявляется виртуальный метод, первым его полем становится указатель на VMT, который идет сразу за унаследованными объектами. (В отличие от современных классов, в которых указатель на

VMT всегда идет в начале, так как в TObject объявлены виртуальные методы.)

«Старые» объекты могут иметь секции private, protected и public, но не имеют секций published и automated. Так как в них нет секции published, «старые» типы объектов не могут иметь информации о типе времени выполнения. «Старые» объектные типы не могут реализовывать интерфейсы.

Конструкторы и деструкторы в «старых» объектных типах работают не так, как новые классы. Чтобы создать экземпляр «старого» объектного типа, вызывается процедура New. Поля только что созданного объекта инициализируются нулями. Если вы объявили конструктор, вы можете вызвать его как часть вызова процедуры New. Укажите имя конструктора и его аргументы в качестве второго аргумента процедуры New. Аналогично, вы можете использовать деструктор при вызове Dispose для освобождения экземпляра объекта. Имя деструктора и его аргументы указываются вторыми аргументами процедуры Dispose.

«Старые» объекты не обязательно создавать динамически. Вы можете рассматривать тип object как записи и объявлять объектные переменные как на уровне модуля, так и локально. Delphi автоматически инициализирует поля строкового типа, поля-динамические массивы и поля типа Variant, но не инициализирует другие поля экземпляра объектного типа.

В отличие от новых классов, исключения, возникающие в конструкторе «старого» объекта, не приводят к автоматическому освобождению динамически созданного объекта или вызову деструктора.