Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Московский государственный университет печати имени Ивана Федорова

А.А. Калинин

РАЗРАБОТКА АЛГОРИТМОВ С ИСПОЛЬЗОВАНИЕМ ПРИНЦИПОВ ООП НА ЯЗЫКЕ C#

Учебно-методическое пособие

для студентов, обучающихся по направлению
230100.62 — Информатика и вычислительная техника

Москва
2014

УДК 004.43
ББК 32.973.26-018.1
К37

А.А. Калинин

К37 Разработка алгоритмов с использованием принципов ООП на языке C# : учебно-методическое пособие / А.А. Калинин ; Моск. гос. ун-т печати имени Ивана Федорова. — М. : МГУП имени Ивана Федорова, 2014. — 106 с.

ISBN 978-5-8122-1301-5

Целью данного учебно-методического пособия, предназначенного для лабораторных занятий, является изучение дисциплины Б3.В.ОД.1 «Объектно-ориентированное программирование» (ООП) на базе *.NET Framework* — программной технологии, реализуемой в MS Visual Studio 2010, и языка программирования C#.

Учебное пособие состоит из двух частей. В 1-й части излагаются неразрывно связанные между собой основы языка C# и принципы ООП в среде MS Visual Studio 2010. Во 2-й части приводятся задания для лабораторных работ с использованием консольных программ.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-8122-1301-5

Содержание

Введение

Целями освоения дисциплины Б3.В.ОД.1 «Объектно-ориентированное программирование» (ООП) является изучение технологии объектно-ориентированного программирования, платформы .NET и языка программирования C#, а также новых концепций развития программного обеспечения.

В результате освоения дисциплины студент должен:

- **знать** основные концепции объектно-ориентированного программирования, основы платформы .NET, основы языка программирования C#;
- **уметь** работать со средой разработки Microsoft Visual Studio, разрабатывать алгоритмы с использованием принципов объектно-ориентированного программирования в C#;
- **иметь навыки** (приобрести опыт) решения поставленных задач, используя объектно-ориентированный подход.

Процесс изучения дисциплины направлен на формирование следующих компетенций:

- владеть основными методами, способами и средствами получения, хранения, переработки информации, иметь навыки работы с компьютером как средством управления информацией (ОК-12);
- осваивать методики использования программных средств для решения практических задач (ПК-2);
- разрабатывать компоненты программных комплексов и баз данных, использовать современные инструментальные средства и технологии программирования (ПК-5).

Профессиональные разработчики программного обеспечения .NET располагают самым серьезным в этой сфере продуктом производства Microsoft, который называется Visual Studio 2010/2012.

В данном учебно-методическом пособии, предназначенном для лабораторных занятий по разработке алгоритмов с использованием

принципов ООП на языке C#, предполагается использование платформы MS Visual Studio 2010, которая поддерживает *.NET Framework* — программную технологию от компании Microsoft версии 4.0.

Visual Studio 2010 поставляется с графическими конструкторами, поддержкой использования отдельных фрагментов кода, средствами для работы с базами данных, утилитами для просмотра объектов и проектов, а также встроенной справочной системой. Кроме того, предлагается множество дополнительных возможностей, наиболее важные из которых перечислены ниже:

- графические редакторы и конструкторы XML;
- поддержка разработки программ Windows, ориентированных на мобильные устройства:
- поддержка разработки программ Microsoft Offlce;
- поддержка разработки проектов Windows Workflow Foundation;
- встроенная поддержка рефакторинга кода;
- инструменты визуального конструирования классов.

Разработка приложений в среде Microsoft Visual Studio 2010 имеет две стороны. С одной стороны, использование языка C# как такового, определяемого его синтаксисом, с другой стороны — приемов работы в интегрированной среде разработки Visual Studio 2010.

Следует понимать, что C# является не единственным языком, который может применяться для построения .NET-приложений. При установке доступного для бесплатной загрузки комплекта разработки программного обеспечения Microsoft .NET 4.0 Framework Software Development Kit (SDK), равно как и при установке Visual Studio 2010, для выбора становятся доступными пять управляемых языков: C#, Visual Basic, C++/CLI, JScript.NET и F#. Помимо управляемых языков, предлагаемых Microsoft, существуют компиляторы, которые предназначены для таких языков, как Smalltalk, COBOL и Pascal (и это далеко не полный перечень).

В настоящем пособии все внимание уделяется языку C#, причем для работы с ним можно также использовать платформу Visual C# 2010 Express.

Для более подробного изучения платформы .NET и языка C# следует обратиться к литературе [1–3].

Часть I. ОСНОВЫ ЯЗЫКА C#

1. Платформа .NET Framework

.NET Framework — это программная технология от компании Microsoft, предназначенная для создания обычных программ и веб-приложений. Одной из основных идей Microsoft, которая используется в .NET, является совместимость различных служб, написанных на разных языках.

Платформа *.NET Framework* предлагает следующие возможности:

- предоставление среды выполнения кода, которая уменьшает конфликты между разными версиями языков, гарантирует безопасное выполнение кода, устраняет проблемы, связанные с переносом сред;
- обеспечивает работу с различными типами приложений, размещенных как в Интернете, так и на локальном компьютере;
- установление стандартов для поддержки платформы *.NET Framework* другими языками;
- создание непротиворечивой объектно-ориентированной среды программирования, в которой код объекта может храниться и выполняться локально или выполняться локально, но быть распределенным в Интернете, или выполняться удаленно.

Из-за того что платформа .NET столь радикально отличается от предыдущих технологий, в Microsoft разработали специально под нее новый язык программирования C#. Ниже приведен неполный список ключевых функциональных возможностей языка C#, которые присутствуют во всех его версиях:

- никаких указателей использовать не требуется. В программах на C# обычно не возникает необходимости в манипулировании указателями напрямую;

- управление памятью осуществляется автоматически посредством сборки мусора. По этой причине ключевое слово delete в C# не поддерживается;
- предлагаются формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов;
- предоставляется аналогичная C++ возможность перегружать операции для пользовательских типов, но без лишних сложностей;
- предлагается поддержка для программирования с использованием атрибутов. Такой подход в сфере разработки позволяет снабжать типы и их членов аннотациями и тем самым еще больше уточнять их поведение.

Возможно, наиболее важным моментом, о котором следует знать, программируя на C#, является то, что с помощью этого языка можно создавать только такой код, который будет выполняться в исполняющей среде .NET (использовать C# для построения «классического» COM-сервера или неуправляемого приложения с вызовами API-интерфейса и кодом на C и C++ нельзя).

Примечание. C# является чувствительным к регистру языком программирования. Поэтому необходимо запомнить, что все ключевые слова в C# вводятся в нижнем регистре (например, public, lock, class, dynamic), а названия пространств имен, типов и членов всегда начинаются (по соглашению) с заглавной буквы, равно как и любые вложенные в них слова (как, например, Console.WriteLine, System.Windows.Forms. MessageBox и System. Data.SqlClient).

2. Общеязыковая исполняющая среда(CLR)

Уровень исполняющей среды называется *общеязыковой исполняющей средой* (Common Language Runtime) или, сокращенно, средой CLR. С точки зрения программирования под термином *исполняющая среда* может пониматься коллекция внешних служб, которые требуются для выполнения скомпилированной единицы программного кода.

Официально код, ориентируемый на выполнение в исполняющей среде .NET, называется:

- *управляемым кодом* (managed code);

- двоичной единицей (*.dll или *.exe), в которой содержится такой управляемый код — *сборкой* (assembly);
- кодом, который не может обслуживаться непосредственно в исполняющей среде .NET — *неуправляемым кодом* (unmanaged code).

Главной задачей CLR является автоматическое обнаружение, загрузка и управление типами .NET (вместо программиста). Кроме того, среда CLR заботится о ряде низкоуровневых деталей, таких как управление памятью, обслуживание приложения, обработка потоков и выполнение различных проверок, связанных с безопасностью. Исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения, который способны использовать все совместимые с .NET языки и платформы.

Сама CLR состоит из двух основных компонентов.

- *Первый компонент* — это ядро среды выполнения, которое реализовано в виде библиотеки mscoree.dll (и также называется общим механизмом выполнения исполняемого кода объектов — Common Object Runtime Execution Engine). При обращении к приложению .NET библиотека mscoree.dll автоматически загружается в память, и затем она управляет процессом загрузки в память сборки данного приложения. Ядро среды выполнения ответственно за множество задач: сначала, что наиболее важно, оно отвечает за определение места расположения сборки и обнаружение запрашиваемого типа в двоичном файле за счет считывания содержащихся там метаданных. Затем оно размещает тип в памяти, преобразует CIL-код в соответствующие платформе инструкции, производит любые необходимые проверки на предмет безопасности и после этого, наконец, непосредственно выполняет сам запрашиваемый программный код.
- *Второй главный компонент* CLR — это библиотека базовых классов. Помимо загрузки пользовательских сборок и создания пользовательских типов, механизм CLR при необходимости будет взаимодействовать и с типами, содержащимися в библиотеках базовых классов .NET. Сама библиотека разбита на множество отдельных сборок, однако главная сборка библиотеки базовых классов представлена файлом mscorlib.dll. В этой сборке

содержится большое количество базовых типов, охватывающих широкий спектр типичных задач программирования, а также базовых типов данных, применяемых во всех языках .NET. При построении .NET-решений доступ к этой конкретной сборке будет предоставляться автоматически. В библиотеке базовых классов содержится огромное количество типов для решения распространенных задач при создании приложения. Приложение .NET будет обязательно использовать сборку mscorlib.dll и по мере необходимости — другие сборки (как встроенные, так и создаваемые).

Хотя двоичные .NET-единицы имеют такое же файловое расширение, как и двоичные единицы COM-серверов и неуправляемых программ Win32 (*.dll или *.exe), внутренне они устроены абсолютно по-другому. Они содержат:

- независящие от платформы инструкции на *промежуточном языке* (Intermediate Language — IL или CIL (Common Intermediate Language — общий промежуточный язык));
- исчерпывающие и точные метаданные, которые описывают каждый определенный в двоичном файле тип (например, класс, структуру или перечисление), а также всех его членов (например, свойства, методы или события). Метаданные .NET являются настолько детальными, что сборки представляют собой полностью самоописываемые (self-describing) сущности. За генерацию новейших и наилучших метаданных по типам всегда отвечает компилятор, а не программист;
- помимо CIL и метаданных типов, сами сборки тоже описываются с помощью метаданных, которые официально называются *манифестом* (manifest). В каждом таком манифесте содержится информация о текущей версии сборки, сведения о культуре (применяемые для локализации строковых и графических ресурсов) и перечень ссылок на все внешние сборки, которые требуются для правильного функционирования.

Из-за того что в сборках содержатся CIL-инструкции, а не инструкции, ориентированные на конкретную платформу, CIL-код перед использованием должен обязательно компилироваться на лету в соответствующий машинный код. Объект, который отвечает за компи-

ляцию CIL-кода в понятные ЦП инструкции, называется *оперативным* (Qust-in-time — JIT) *компилятором*. Иногда его «по-дружески» называют Jitter.

Поскольку сборки описывают себя сами с помощью метаданных, то для их установки не требуется никакой внешней информации, например, в системном реестре. Это делает .NET-компоненты намного более простыми и менее подверженными ошибкам при установке и деинсталляции, чем традиционные компоненты, построенные на основе модели компонентных объектов Microsoft (COM), которые имели обширные требования к информации в системном реестре.

Может возникнуть вопрос о том, какую выгоду приносит компиляция исходного кода в CIL, а не напрямую в набор ориентированных на конкретную платформу инструкций. Одним из самых важных преимуществ такого подхода является интеграция языков. Как уже можно было увидеть, все компиляторы .NET генерируют примерно одинаковые CIL-инструкции. Благодаря этому все языки могут взаимодействовать в рамках четко обозначенной двоичной «арены».

Процесс *развертывания* (в рассматриваемом здесь случае) включает следующее:

- установку — копирование файлов приложения с физического или электронного носителя в рабочую систему, а также настройку операционной системы для запуска приложения пользователями;
- обновление — изменение, добавление, удаление или замену существующих файлов приложения в системе для обновления версии и добавления новых или исправленных функций;
- удаление — удаление файлов приложений, а также настройка операционной системы таким образом, чтобы приложения больше не было в списке доступных для запуска программ.

Сборки положены в основу технологии компонентов .NET. Сборка — основная единица развертывания и управления разрешениями защиты, версиями, а также повторным использованием двоичного кода. Сборка может быть составлена из одного или нескольких физических файлов на диске, но это — все равно одна *логическая единица развертывания*. В большом количестве случаев между сбор-

ками .NET и файлами двоичного кода (*.dll или *.exe) соблюдается простое соответствие «один к одному». Следовательно, получается, что при построении *.dll-библиотеки .NET можно спокойно полагать, что файл двоичного кода и сборка представляют собой одно и то же, и что, аналогичным образом, при построении исполняемого приложения для настольной системы на файл .exe можно ссылаться как на саму сборку.

Сборка — единица развертывания, пространства имен поддерживают иерархическую систему именования, а динамически подключаемая библиотека (DLL) или исполняемый файл — есть единица упаковки функциональных возможностей в пределах файловой системы. Важно также понимать разницу между сборкой, которая является единицей развертывания, и приложением, которое является единицей конфигурации.

Сборка без подписи идентифицируется просто удобочитаемым названием (именем), а также номером версии. Сборка с цифровой подписью идентифицируется также по имени ее создателя, однозначно определяемому по криптографической паре (ключей). Простая проверка версии гарантирует, что клиентская программа правильно использует общедоступные сборки с учетом тех их версий, которые были определены при первоначальной компиляции и тестировании клиента.

Благодаря этому можно на самом деле устранить неприятную проблему «ада динамически подключаемых библиотек (DLL)», где клиенты и компоненты, построенные на основе модели компонентных объектов Microsoft (COM), могли легко «рассинхронизироваться» друг с другом, если старая версия заменялась более новой, — это нарушало работу существующих клиентов.

Сборки содержат хэш-код, представляющий двоичное содержимое сборки. Он используется, чтобы подтвердить подлинность или обнаружить искажение сборок с цифровой подписью. *Хеширование* — преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины.

В составе платформы .NET поставляется *библиотека базовых классов,* которая является доступной для всех языков программирования .NET. В этой библиотеке не только содержатся определения различных примитивов, таких как потоки, файловый ввод-вывод,

системы графической визуализации и механизмы для взаимодействия с различными внешними устройствами, но также предоставляется поддержка для целого ряда служб, требуемых в большинстве реальных приложений.

В библиотеке базовых классов содержатся определения типов, которые способны упрощать процесс получения доступа к базам данных, манипулирования XML-документами, обеспечения программной безопасности и создания веб-, а также обычных настольных и консольных интерфейсов. На высоком уровне взаимосвязь между CLR, CTS, CLS и библиотекой базовых классов выглядит так, как показано в табл. 2.1.

Таблица 2.1

Отношения между CLR, CTS, CLS и библиотеками базовых классов

<table>
<tr><td>Доступ к базе данных</td><td>Настольные графические API-интерфейсы</td><td>Безопасность</td><td>API-интерфейсы для удаленной работы</td></tr>
<tr><td>Организация потоковой обработки</td><td>Файловый ввод-вывод</td><td>API-интерфейсы для работы с веб-содержимым</td><td>и другие</td></tr>
<tr><td colspan="4">Общеязыковая исполняющая среда (CLR)</td></tr>
<tr><td colspan="4">Общая система типов (CTS)</td></tr>
<tr><td colspan="4">Общеязыковая спецификация (CLS)</td></tr>
</table>

Язык C#, однако, не поставляется с какой-либо специфической библиотекой кода. Вместо этого от разработчиков, использующих C#, требуется применять нейтральные к языкам библиотеки, которые поставляются в .NET. Для поддержания всех *типов* в библиотеках базовых классов в хорошо организованном виде в .NET широко применяется понятие *пространства имен* (namespace). Под пространством имен понимается группа связанных между собой с семантической точки зрения типов, которые содержатся в сборке. В C# ключевое слово using упрощает процесс добавления ссылок на типы, содержащиеся в определенном пространстве имен.

Отличие между таким подходом и зависящими от конкретного языка библиотеками состоит в том, что он обеспечивает использование во всех языках, ориентированных на среду выполнения .NET, *одних и тех же* пространств имен и *одних и тех же* типов.

3. Общая система типов (CTS)

Общая система типов (Common Type System) или, сокращенно, система CTS представляет собой формальную спецификацию, в которой описано, как должны быть определены типы для обслуживания в CLR-среде.

В мире .NET «тип» представляет собой просто общий термин, который применяется для обозначения любого элемента из множества (класс, интерфейс, структура, перечисление, делегат).

В спецификации CTS представлено полное описание всех возможных типов данных и программных конструкций, поддерживаемых исполняющей средой, как эти сущности могут взаимодействовать друг с другом, как они могут представляться в формате метаданных .NET.

Тип класс

В каждом совместимом с .NET языке поддерживается, как минимум, понятие *типа класса* (class type), которое играет центральную роль в объектно-ориентированном программировании (ООП). Каждый класс может включать любое количество членов (таких как конструкторы, свойства, методы и события) и точек данных (полей). В C# классы объявляются с помощью ключевого слова *class*.

Характеристика классов:

- экземпляры *абстрактных* (abstract) классов не могут создаваться напрямую, и предназначены для определения общих аспектов поведения для производных типов. Экземпляры же *конкретных* (concrete) классов могут создаваться напрямую;
- *запечатанные* (sealed), или *герметизированные*, классы не могут выступать в роли базовых для других классов, т. е. не допускают наследования;
- каждый класс должен конфигурироваться с атрибутом *видимости* (visibility). По сути, этот атрибут указывает, должен ли класс быть доступным для использования внешним сборкам или только изнутри определяющей сборки.

Тип интерфейс

Интерфейсом (interface) называется коллекция абстрактных членов, которые обеспечивают возможность взаимодействия между

объектом и пользователем этого объекта. CTS позволяет реализовать в классе любое количество интерфейсов.

Интерфейсы представляют собой не более чем просто именованную коллекцию определений абстрактных членов, которые могут поддерживаться (т. е. реализоваться) в данном классе или структуре. В C# типы интерфейсов определяются с помощью ключевого слова *interface*.

Тип интерфейса в C# обычно объявляется общедоступным, чтобы позволить типам в других сборках реализовать его поведение.

Тип структура

Структуры лучше всего подходят для моделирования геометрических и математических данных, и в C# они создаются с помощью ключевого слова *struct*. Они представляют собой типы, способные содержать любое количество полей данных и членов, выполняющих над ними какие-то операции.

Можно считать структуры «облегченными классами», поскольку они тоже предоставляют возможность определять тип, поддерживающий инкапсуляцию, но не могут применяться для построения семейства взаимосвязанных типов. Когда есть потребность в создании семейства взаимосвязанных типов через наследование, нужно применять типы классов.

Тип перечисление

Перечисления (enum) представляют собой удобную программную конструкцию, которая позволяет группировать данные в пары «имя-значение».

Тип делегат

Делегаты (delegate) являются .NET-эквивалентом безопасных в отношении типов указателей функций в стиле C. Главное отличие заключается в том, что делегат в .NET представляет собой *класс,* который наследуется от System.MulticastDelegate, а не просто указатель на какой-то конкретный адрес в памяти. В C# делегаты объявляются с помощью ключевого слова *delegate*.

Встроенные типы данных

И, наконец, последним, что следует знать о спецификации CTS, является то, что в ней содержится четко определенный набор фун-

даментальных типов данных. Хотя в каждом отдельно взятом языке для объявления того или иного встроенного типа данных из CTS обычно предусмотрено свое уникальное ключевое слово, все эти ключевые слова в конечном итоге соответствуют одному и тому же типу в сборке mscorlib.dll.

4. Общеязыковая спецификация (CLS)

Важно понимать, что любая из определенных в CTS функциональных возможностей может не поддерживаться в отдельно взятом языке, совместимом с .NET. Поэтому существует еще *общеязыковая спецификация* (Common Language Specification) или, сокращенно, спецификация CLS, в которой описано лишь то подмножество общих типов и программных конструкций, какое способны воспринимать абсолютно все поддерживающие .NET языки программирования. Следовательно, в случае построения типов .NET только с функциональными возможностями, которые предусмотрены в CLS, можно оставаться полностью уверенным в том, что все совместимые с .NET языки смогут их использовать. И, наоборот, в случае применения такого типа данных или конструкции программирования, которой нет в CLS, рассчитывать на то, что каждый язык программирования .NET сможет взаимодействовать с подобной библиотекой кода .NET, нельзя.

4.1. Системные типы данных в C#

Как и в любом языке программирования, в C# поставляется собственный набор основных типов данных, которые должны применяться для представления локальных переменных, переменных экземпляра, возвращаемых значений и входных параметров.

В табл. 4.1 перечислены эти системные типы данных вместе с охватываемыми ими диапазонами значений, соответствующими ключевыми словами на C# и сведениями о том, отвечают ли они требованиям общеязыковой спецификации CLS (Common Language Specification)

Таблица 4.1

Внутренние типы данных в C#

C#	CLS	Системный тип	Диапазон значений	Описание
bool	Да	System. Boolean	true или false	истина или ложь
sbyte	Нет	System. SByte	от –128 до 127	8-битное число со знаком
byte	Да	System. Byte	от 0 до 255	8-битное число без знака
short	Да	System. Intl6	от –32 768 до 32 767	16-битное число со знаком
ushort	Нет	System. UIntl6	от 0 до 65 535	16-битное число без знака
int	Да	System. Int32	от –2 147 483 648 до 2 147 483 647	32-битное число со знаком
uint	Нет	System. UInt32	от 0 до 4 294 967 295	32-битное число без знака
long	Да	System. Int64	от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-битное число со знаком
ulong	Нет	System. UInt64	от 0 до 18 446 744073 709 551 615	64-битное число без знака
char	Да	System. Char	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
float	Да	System. Single	от $+1{,}5\times10^{-45}$ до $3{,}4\times10^{38}$	32-битное число с плавающей точкой
double	Да	System. Double	от $5{,}0\times10^{-324}$ до $1{,}7\times10^{308}$	64-битное число с плавающей точкой
deci-mal	Да	System. Decimal	от $\pm1{,}0\times10e^{-28}$ до $\pm7{,}9\times10^{28}$	96-битное число со знаком
string	Да	System. String	Ограничивается объемом системной памяти	Представляет ряд символов в формате Unicode
object	Да	System. Object	Позволяет сохранять любой тип в объектной переменной	Служит базовым классом для всех типов в мире .NET

4.2. Числовые данные

Очень интересно отметить то, что даже элементарные типы данных в .NET имеют вид иерархии классов. Каждый из этих типов в конечном итоге наследуется от класса System.Object, в котором содержится набор методов (таких как ToString (), Equals () и GetHashCode ()), являющихся общими для всех поставляемых в библиотеках базовых классов .NET типов.

Важно отметить, что многие из числовых типов данных унаследованы от класса System.ValueType. Потомки ValueType автоматически размещаются в стеке и потому обладают очень предсказуемым временем жизни и являются довольно эффективными. Стек-структура данных представляет собой список элементов, организованных по принципу «последним пришел — первым вышел».

Типы, у которых в цепочке наследования не присутствует класс System. ValueType (вроде System.Type, System.String, System.Array, System.Exception и System. Delegate), в стеке не размещаются, а попадают в кучу и подвергаются автоматической сборке мусора.

Продолжая обсуждение встроенных типов данных в C#, нельзя не упомянуть о том, что числовые типы в .NET поддерживают свойства MaxValue и MinValue, которые позволяют получать информацию о диапазоне значений, хранящихся в данном типе. Помимо свойств MinValue/MaxValue, каждый числовой тип может иметь и другие полезные члены. Например, тип System.Double позволяет получать значения эпсилон (бесконечно малое) и бесконечность (представляющие интерес для тех, кто занимается решением математических задач).

Для нас существенно, что тип System.Double, являясь структурой, содержит поле NaN, которое используется в методах класса System.Math.

public const double NaN	Метод или оператор возвращают значение NaN, если не указан результат операции
Double.NaN — поле	Представляет значение, не являющееся числом (NaN). Данное поле является константой
public static bool IsNaN(double d)	Возвращает значение true, если значение параметра d равно значению NaN; в противном случае — значение false

Класс System.Math (public static class Math) предоставляет *константы и статические методы* для тригонометрических, логарифмических и иных общих математических функций. При обращении к ним необходимо через точку использовать в качестве префикса имя класса Math.

Таблица 4.2

Избранные члены класса System.Math для данных типа двойной точности с плавающей запятой

Свойства	**Описание**
public const double E	e = 2,7182818284590452354
public const double PI	π = 3,14159265358979323846
Методы	**Описание**
public static double Abs(double d)	Возвращает абсолютное значение числа d
public static double Acos(double d)	Возвращает угол, θ, измеренный в радианах, такой, что $0 \le \theta \le \pi$ -или-значение NaN, если $d < -1$ или $d > 1$
public static double Asin(double d)	Возвращает угол, θ, измеренный в радианах, такой, что $-\pi/2 \le \theta \le \pi/2$ -или- значение NaN, если $d < -1$ или $d > 1$
public static double Atan(double d)	Возвращает угол, θ, измеренный в радианах, такой, что $-\pi/2 < \theta < \pi/2$, -или- значение NaN, если $d = \|\pi/2\|$
public static double Cos(double d)	Возвращает косинус указанного угла d, измеряемого в радианах
public static double Exp(double d)	Возвращает e, возведенное в указанную степень d
public static double Log(double d)	Возвращает натуральный логарифм d (с основанием e), если $d > 0$
public static double Log10(double d)	Возвращает логарифм d с основанием 10, если $d > 0$
public static double Pow(double x, double y)	Возвращает число x, возведенное в степень y
public static double Round(double d)	Округляет заданное число d до ближайшего целого
public static double Sin(double d)	Возвращает синус указанного угла d, измеряемого в радианах

Окончание табл. 4.2

Методы	Описание
public static double Sqrt(double d)	Возвращает квадратный корень d, если 0 ≤ d
public static double Tan(double d)	Возвращает тангенс указанного угла d, измеряемого в радианах
public static double Truncate(double d)	Вычисляет целую часть заданного числа d

4.3. Текстовые данные

Большинство современных языков программирования (как и некоторые старые) проводят ясную черту между двоичным (бинарным) контентом и символьным (или текстовым) контентами.

Бинарные (двоичные) данные являются последовательностью байт (или октетов), которые состоят из 8 бит, без всякого придаваемого им естественного значения, или интерпретации. Символьные (текстовые) данные являются последовательностью символов.

Юникóд (Unicode) — стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков. Большинство современных операционных систем в той или иной степени обеспечивают поддержку Юникода.

Текстовые данные в C# представляются с помощью ключевых слов *string* и *char*, которые являются сокращенными вариантами обозначения типов System.String и System.Char.

Объект String является упорядоченной коллекцией структур Char, представляющей строку текста.

NET Framework использует структуру Char, чтобы представить символ как элемент кода UTF-16 (англ. *Unicode Transformation Format*). В информатике это один из способов кодирования символов из Unicode в виде последовательности 16-битовых слов. Стандарт Юникода идентифицирует каждый символ уникальным 21-битовым скалярным числом, называемым кодовой точкой, и определяет форму кодировки UTF-16, показывающую, как выполняется кодирование кодовой точки последовательностью из одного или более 16-битовых значений. Каждое 16-битовое значение лежит в диапазоне, ограниченном шестнадцатеричными значениями 0x0000 и

0xFFFF и хранится в структуре Char. Значением объекта Char является 16-битовое числовое (порядковое) значение.

Большинство символов Юникода могут быть представлены одним объектом Char, но символы, кодируемые как базовые, суррогатные пары и (или) последовательности несамостоятельных знаков, представлены несколькими объектами Char. По этой причине структура Char необязательно должна быть эквивалентна отдельному символу Юникода в объекте String

Помимо возможности хранить один элемент символьных данных, структура System.Char обладает массой других функциональных возможностей. В частности, с помощью ее статических методов можно определять, является данный символ цифрой, буквой, знаком пунктуации или чем-то еще.

В классе System. String предоставляется набор свойств и методов для определения длины символьных данных, поиска подстроки в текущей строке, преобразования символов из верхнего регистра в нижний и наоборот, и т. д.

Абстрактный класс System.Text.Encoding предоставляет средства конвертации массива байт в массив символов или в строку, а также наоборот.

Таблица 4.3

Избранные члены класса System.String

Свойства	**Описание**
public char this[int index] { get; }	Получает объект Char в указанной позиции в текущем объекте String (только для чтения). Квадратные скобки [] служат для доступа к отдельным знакам в объекте string, но при этом возможен доступ только для чтения
public int Length { get; }	Свойство Length возвращает число объектов Char в данном экземпляре (только для чтения), а не число знаков Юникода. Причина в том, что знак Юникода может быть представлен несколькими Char
Методы	**Описание**
public static int Compare(string strA,string strB)	Сравнивает два указанных объекта String и возвращает целое число, которое показывает их относительное положение в порядке сортировки по словам

Окончание табл. 4.3

Методы	Описание
public bool Contains(string value)	Возвращает значение, указывающее, содержит ли данная строка заданный объект типа String
public bool Equals(string value)	Определяет, равны ли значения этого экземпляра и указанного объекта String
public static string Format( )	см. разделы I, 6.2
public string Insert(int startIndex, string value)	Возвращает новую строку, в которой указанная строка вставляется на указанной позиции индекса, отсчитываемой от нуля, в данном экземпляре
public string PadLeft(int totalWidth)	Возвращает новую строку, в которой знаки данного экземпляра выровнены по правому (левому) краю путем добавления слева (справа) пробелов до указанной общей длины
public string PadRight(int totalWidth)	
public string Remove(int startIndex)	Возвращает новую строку, в которой были удалены все символы, начиная с указанной позиции, и до конца в текущем экземпляре
public string Replace(char oldChar,char newChar)	Возвращает новую строку, в которой все вхождения заданного знака Юникода в текущем экземпляре заменены другим заданным знаком Юникода
public string Replace(string oldValue,string newValue)	Возвращает новую строку, в которой все вхождения заданной строки в текущем экземпляре заменены другой заданной строкой
public string Trim()	Удаляет все начальные и конечные знаки пробела из текущего объекта String
public string ToUpper()	Возвращает копию этой строки, переведенную в верхний (нижний) регистр
public string ToLower()	

Пример:

```
//Используем свойства класса String (только для //чтения)
string str1 = "Test";
for (int ctr = 0; ctr <= str1.Length — 1; ctr + + )
Console.Write("{0} ", str1[ctr]);
// Вывод на консоль:
// T e s t
```

Пример:

```
// Используем метод Raplace() класса String
public static void Main() {
String str = "1 2 3 4 5 6 7 8 9";
Console.WriteLine("Оригинальная строка:\"{0}\"",str);
Console.WriteLine("Строка результата:\"{0}\"",
str.Replace(' ', ','));
}
//\" — управляющий символ для ввода кавычки в строку
// Вывод на консоль:
// Оригинальная строка: "1 2 3 4 5 6 7 8 9"
// Строка результата: "1,2,3,4,5,6,7,8,9"
//
```

4.4. Управляющие символы

При выводе на экран строк в C# можно использовать управляющие последовательности, позволяющие управлять процессом вывода текстовой информации на экран. Основной сферой применения этих последовательностей является вывод тех символов, которые невозможно вывести на экран при помощи обычной стандартной строки, заключенной в кавычки.

Кроме управляющих последовательностей в C# предусмотрен специальный префикс @, предназначенный для дословного вывода строки, вне зависимости от наличия в ней управляющих символов. Можно сказать, что этот префикс отключает присутствующие в строке управляющие символы.

Таблица 4.4

Управляющие символы

Символ	Имя символа	Кодировка Юникода
\'	Одинарная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратная косая черта	0x005C
\0	Null	0x0000
\a	Системное оповещение	0x0007
\b	Backspace	0x0008
\f	Начало следующей страницы	0x000C

Окончание табл. 4.4

Символ	Имя символа	Кодировка Юникода
\n	Новая строка	0x000A
\v	Вертикальная табуляция	0x000B
\t	Горизонтальная табуляция	0x0009

5. Метод Main(). Методы и модификаторы параметров

В отличие от многих других языков, в C# не допускается создавать ни глобальных функций, ни глобальных элементов данных. Вместо этого требуется, чтобы все члены данных и методы содержались внутри определения типа.

Класс, определяющий метод Main(), должен обязательно присутствовать в каждом исполняемом приложении на C# (будь то консольная программа, настольная программа для Windows или служба Windows), поскольку он применяется для обозначения точки входа в приложение. Формально класс, в котором определяется метод Main(), называется объектом приложения.

Создавая консольный проект с именем ConsoleApplication в Visual Studio 2010, классу, в котором определяется метод Main(),по умолчанию назначается имя Program. Класс содержит статический метод Main() с возвращаемым значением void и массивом типа string в качестве единственного входного параметра.

Область действия статических (static) членов класса охватывает уровень всего класса (а не уровень отдельного объекта) и потому они могут вызываться без предварительного создания нового экземпляра класса.

По умолчанию в результате создания проекта ConsoleApplication, имеем:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Далее, при написании программы метод Main() модифицируется добавлением программного кода, а также в класс Program могут быть добавлены и другие статические методы.

Методы могут как принимать, так и не принимать параметров, а также возвращать или не возвращать значения вызывающей стороне. Методы могут быть реализованы в контексте классов или структур (а также прототипированы внутри типов интерфейсов) и снабжаться различными ключевыми словами (internal, virtual, public, new и т. д.) для уточнения их поведения. Хотя определение метода в C# выглядит довольно понятно, существует несколько ключевых слов, с помощью которых можно управлять способом передачи аргументов интересующему методу.

Таблица 5.1

Модификаторы параметров в C#

Модификатор параметра	Описание
(отсутствует)	По умолчанию, если параметр не сопровождается модификатором, предполагается, что он должен передаваться по значению, т. е. вызываемый метод должен получать копию исходных данных
out	Выходные параметры должны присваиваться вызываемым методом (и, следовательно, передаваться по ссылке). Если параметрам out в вызываемом методе значения не присвоены, компилятор сообщит об ошибке
ref	Это значение первоначально присваивается вызывающим кодом и при желании может повторно присваиваться в вызываемом методе (поскольку данные также передаются по ссылке). Если параметрам ref в вызываемом методе значения не присвоены, компилятор никакой ошибки генерировать не будет

Окончание табл. 5.1

Модификатор параметра	Описание
params	Этот модификатор позволяет передавать в виде одного логического параметра переменное количество аргументов. В каждом методе может присутствовать только один модификатор params и он должен обязательно указываться последним в списке параметров

Как и в других современных объектно-ориентированных языках программирования, в C# можно перегружать (overload) методы. Перегруженными называются методы, одинаково именованные, отличающиеся друг от друга количеством (или типом) параметров или типом возвращаемого значения, т. е. сигнатурой.

6. Ввод и вывод класса System.Console

Хотя консольный пользовательский интерфейс (Console User Interface — CUI) не является настолько привлекательным как графический (Graphical User Interface — GUI) или веб-интерфейс, однако, ограничение консольными программами позволяет уделить больше внимания синтаксису C# и ключевым характеристикам платформы .NET, а не сложным деталям построения графических пользовательских интерфейсов или веб-сайтов.

В классе Console имеются методы, которые позволяют захватывать ввод и вывод; все они являются *статическими* (static) и потому вызываются за счет добавления к имени метода в качестве префикса имени самого класса (Console).

К их числу относится уже показанный ранее метод WriteLine(), который позволяет вставлять в поток вывода строку текста (вместе с символом возврата каретки); метод Write(), который позволяет вставлять в поток вывода текст без символа возврата каретки; метод ReadLine(), который разрешает получать информацию из потока ввода вплоть до нажатия клавиши <Enter>; и метод Read(), позволяющий захватывать из потока ввода одиночный символ.

При определении строкового литерала с сегментами данных, значения которых остаются неизвестными до этапа выполнения, внутри него допускается указывать метку-заполнитель с использовани-

ем синтаксиса в виде фигурных скобок. Во время выполнения на месте каждой такой метки-заполнителя подставляется передаваемое в Console.WriteLine () значение (или значения).

В качестве первого параметра методу WriteLine() всегда передается строковый литерал, в котором могут содержаться метки-заполнители. Другими словами, строка *составного формата* состоит из фиксированного текста, перемежаемого индексированными метками-заполнителями, являющимися *элементами формата*, которые соответствуют объектам в списке.

Каждый элемент формата принимает следующую форму и состоит из следующих компонентов:

{ Индекс[,Выравнивание][:formatString]}

Парные фигурные скобки "{" и "}" являются обязательными.

Следует запомнить, что отсчет обязательного параметра *Индекса* в окружаемых фигурными скобками метках-заполнителях всегда начинается с нуля.

Необязательный параметр *Выравнивание* определяет желательную ширину поля форматирования. Если значение параметра *Выравнивание* меньше длины форматируемой строки, параметр *Выравнивание* игнорируется, и в качестве значения ширины поля используется длина форматируемой строки.

Необязательный параметр *formatString* является строкой формата. Если *formatString* не указан, используется описатель общего формата ("G") для типа числовых значений даты и времени или перечисления. Двоеточие является обязательным, если *formatString* указан.

Остальными передаваемыми WriteLine() параметрами являются просто значения, которые должны подставляться на месте соответствующих меток — заполнителей.

> *Примечание.* Если количество пронумерованных уникальным образом заполнителей превышает число необходимых для их заполнения аргументов, во время выполнения будет генерироваться исключение, связанное с форматом.

Пример:

```
//Отображение значений переменных userName и
//userAge в окне консоли.
```

```
Console.WriteLine("Hello {0}! You are {1} years
old.", userName, userAge);
```

Метка-заполнитель может повторяться в пределах одной и той же строки.

Пример:

```
// Вывод строки "9, Number 9, Number 9"
Console.WriteLine("{0}, Number {0}, Number {0}",9);
```

Каждый заполнитель допускается размещать в любом месте внутри строкового литерала, и вовсе не обязательно, чтобы следующий после него заполнитель имел более высокий номер.

Пример:

```
// Отображает: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

6.1. Форматирование числовых данных

Строки стандартных числовых форматов служат для форматирования стандартных числовых типов. Стандартная строка числового формата имеет вид Axx, где A — буквенный символ, называемый *спецификатором формата*, а xx — необязательное целое число, называемое *спецификатором точности.* Спецификатор точности находится в диапазоне от 0 до 99 и влияет на число цифр в результате. Он не округляет число.

Таблица 6.1

Символы для форматирования числовых данных в .NET

Спецификатор формата	Спецификатор точности	Описание
C или c	Количество цифр дробной части	Результат: значение валюты
D или d	Минимальное число цифр. По умолчанию: минимальное необходимое число цифр	Результат: целочисленные цифры с необязательным отрицательным знаком. Поддерживается: только целочисленными типами данных

Окончание табл. 6.1

Спецификатор формата	Спецификатор точности	Описание
E или e	Количество цифр дробной части. По умолчанию: 6	Результат: экспоненциальная нотация. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (E) или в нижнем (e)
F или f	Количество цифр дробной части	Результат: цифры целой и дробной частей с необязательным отрицательным знаком
G или g	Количество значащих цифр. По умолчанию определяется численным типом	Результат: наиболее компактная запись из двух вариантов: экспоненциального и с фиксированной запятой
N или n	Желаемое число знаков дробной части	Результат: цифры целой и дробной частей, разделители групп и разделитель целой и дробной частей с необязательным отрицательным знаком
P или p	Желаемое число знаков дробной части	Результат: число, умноженное на 100 и отображаемое с символом процента
X или x	Число цифр в результирующей строке	Результат: шестнадцатеричная строка. Поддерживается: только целочисленными типами данных. В случае использования символа X в верхнем регистре, в шестнадцатеричном представлении будут содержаться символы верхнего регистра

6.2. Форматирование данных в приложениях, отличных от консольных

Составное форматирование может использоваться не только в консольных приложениях. Тот же синтаксис форматирования можно применять и в вызове статического метода string.Format(), который возвращает новый объект типа string, форматируемый в соответствии

с предоставляемыми флагами, описанными выше. После этого текстовые данные могут использоваться любым желаемым образом.

Это может быть удобно при генерации во время выполнения текстовых данных, которые должны использоваться в приложении любого типа (например, в настольном приложении с графическим пользовательским интерфейсом, в веб-приложении ASP.NET или веб-службах XML).

Пример:

Требуется создать графическое настольное приложение и применить форматирование к строке, отображаемой в окне сообщения внутри него:

```
static void DisplayMessage()
{
// Использование string.Format() для форматирования
//строкового литерала.
string userMessage = string.Format("100000 in hex
is {0:x}", 100000);
// Для компиляции этой строки кода требуется
// ссылка на System.Windows.Forms.dll
System.Windows.Forms.MessageBox.Show(userMessage);
}
```

7. Конструкции в C#

7.1. Итерационные конструкции

Все языки программирования предлагают способы для повторения блоков кода до тех пор, пока не будет соблюдено какое-то условие завершения. Какой бы язык не использовался ранее, операторы, предлагаемые для осуществления итераций в C#, не должны вызывать особых недоумений и требовать особых объяснений.

Цикл for

Если требуется проходить по блоку кода фиксированное количество раз, приличную гибкость демонстрирует оператор for. Он позволяет указать, сколько раз должен повторяться блок кода, а также задать конечное условие, при котором его выполнение должно быть завершено.

Пример:

```
for(int i=0; i<4; i++){ }
```

Как и в других языках, в C# можно создавать сложные конечные условия, определять бесконечные циклы и использовать ключевые слова *goto, continue* и *break*.

Цикл foreach

Ключевое слово *foreach* в C# позволяет проходить в цикле по всем элементам массива (или коллекции) без проверки его верхнего предела.

Пример:

```
int[] myInts = { 10, 20, 30, 40 };
foreach (int i in myInts){ }
//in — ключевое слово, i — параметр цикла
```

Конструкции while и do/while

Конструкцию while удобно применять, когда требуется, чтобы блок операторов выполнялся до тех пор, пока не будет удовлетворено какое-то конечное условие. Обозначим его в угловых скобках в примере. Естественно, нужно позаботиться о том, чтобы это условие когда-нибудь действительно достигалось, иначе получится бесконечный цикл.

Пример:

```
while( <условие>){ }
do{ } while( <условие>);
```

7.2. Конструкции принятия решений и операции сравнения

Оператор if/else

В C# этот оператор может работать только с булевскими выражениями. С учетом этого, для получения литеральных булевских значений, в операторах if/else обычно применяются операции, перечисленные в табл. 7.1.

Таблица 7.1

Операции сравнения в C#

Операция сравнения	Описание
==	Возвращает true, только если выражения одинаковы
! =	Возвращает true, только если выражения разные
<	Возвращает true, только если выражение слева меньше, больше, меньше или равно либо больше или равно выражению справа
>	
<=	
>=	

При построении сложных выражений в C# используется вполне ожидаемый набор условных операций, описанный в табл. 7.2.

Таблица 7.2

Условные операции в C#

Операция	Описание
&&	Условная операция AND (И). Возвращает true, если все выражения истинны
‖	Условная операция OR (ИЛИ). Возвращает true, если истинно хотя бы одно из выражений
!	Условная операция NOT (НЕ). Возвращает true, если выражение ложно, или false, если истинно

Оператор switch

Как и в остальных C-подобных языках, в C# этот оператор позволяет организовать выполнение программы на основе заранее определенного набора вариантов.

Пример:

```
switch (strChoice)
{
case "C#": Console.WriteLine("Выбор языка C#"); break;
case "VB": Console.WriteLine("Выбор языка Visual Basic"); break;
default: Console.WriteLine("Выбор языка программирования
не состоялся"); break;
}
```

Блок default обрабатывает неверный выбор.

8. Массивы в C#

8.1. Объявление массивов

Массивом (array) называется набор элементов данных, доступ к которым получается по числовому индексу. Если говорить более конкретно, то любой массив, по сути, представляет собой ряд связанных между собой элементов данных одинакового типа.

Пример:

```
string[] booksOnDotNet = new string[100];
```

При объявлении массива с помощью такого синтаксиса указываемое в объявлении число обозначает общее количество элементов, а не верхнюю границу. Кроме того, нижняя граница в массиве всегда начинается с 0.

Помимо заполнения массива элемент за элементом, можно также заполнять его с использованием специального синтаксиса *инициализации массивов*. Для этого необходимо перечислить включаемый в массив элемент в фигурных скобках { }.

Пример:

```
bool[] boolArray = { false, false, true };
```

В дополнение к одномерным массивам, которые демонстрировались до сих пор, в C# также поддерживаются две разновидности многомерных массивов.

Многомерные массивы первого вида называются *прямоугольными массивами* и содержат несколько измерений, где все строки имеют одинаковую длину. Объявляются и заполняются такие массивы следующим образом.

Пример:

```
int[,] myMatrix;
myMatrix = new int[6,7];
```

Многомерные массивы второго вида называются *зубчатыми (Jagged) массивами,* они содержат некоторое количество внутренних массивов, каждый из которых может иметь собственный уникальный верхний предел.

Пример:

```
int[][] myJagArray = new int [5][ ];
// Создание зубчатого массива.
for (int i = 0; i < myJagArray.Length; i++)
myJagArray[i] = new int[i+7];
```

Массив можно передавать как аргумент метода или получать в виде возвращаемого значения метода.

Абстрактный класс System.Array (public abstract class Array) предоставляет свойства и методы для создания, изменения, поиска и сортировки массивов, т. е. выступает в роли базового класса для всех массивов в среде CLR.

8.2. Класс System.Random

Класс System.Random (public class Random) реализует алгоритм генератора случайных чисел Кнута.

Псевдослучайные числа выбираются с равной вероятностью из конечного набора чисел. Выбранные числа не являются строго случайными, так как для их выборки используется четкий математический алгоритм, но они достаточно случайны для практического применения.

Генерация случайных чисел начинается с начального значения. При повторном использовании того же начального значения создается та же последовательность чисел. Одним из способов получения различных последовательностей является выбор зависимого от времени начального значения, что позволяет создавать различные последовательности для каждого нового экземпляра Random.

По умолчанию в лишенном параметров конструкторе класса Random для генерации начального значения используются системные часы, в то время как параметризованный конструктор данного класса может принимать значение типа Int32, зависящее от количества тактов в текущем времени. Однако вследствие конечности разрешающей способности часов, использование конструктора без параметров, при создании различных объектов Random в быстрой последовательности, приводит к созданию генераторов случайных чисел, производящих идентичные числовые последовательности. Эту проблему можно устранить, создав единый объект Random вместо нескольких генераторов случайных чисел.

Для повышения производительности также нужно создать один объект Random для генерации множества случайных чисел с течением времени, отказавшись от создания новых генераторов Random для каждого случайного числа.

Таблица 8.1

Избранные члены класса System.Random

Конструкторы	**Описание**
public Random()	Инициализирует новый экземпляр класса Random с помощью зависимого от времени начального значения по умолчанию
public Random(int Seed)	Инициализирует новый экземпляр класса Random с помощью указанного начального значения
Методы	**Описание**
public virtual int Next()	Возвращает неотрицательное случайное число
public virtual int Next(int maxValue)	Возвращает 32-разрядное целое число со знаком больше или равно нулю и меньше, чем значение maxValue
public virtual int Next(int minValue, int maxValue)	Возвращает 32-разрядное целое число со знаком большее или равно minValue и меньше, чем maxValue; т. е., диапазон возвращаемого значения включает minValue, не включает maxValue. В отличие от других перегрузок метода Next, возвращающих только неотрицательные значения, этот метод может также возвращать отрицательные случайные целые числа
public virtual double NextDouble()	Возвращает число двойной точности с плавающей запятой, которое больше или равно 0,0, и меньше 1,0

Пример:

```
Random rand = new Random();
// Generate and display 5 random floating point //values from 0 to 1.
Console.WriteLine("Five Doubles.");
for (int ctr = 0; ctr <= 4; ctr++)
Console.Write("{0,8:N3}", rand.NextDouble());
Console.WriteLine();
```

Метод NextDouble() дает возможность генерировать случайные числа в интервале $\xi \in [0, 1)$. Чтобы с помощью этого метода класса

Random генерировать случайные числа в интервале $\chi \in [A, B)$, где $A < B$, следует использовать линейное преобразование

$$\chi = (B - A)\xi + A,$$

т. е. следует в приведенном выше примере сделать замену:
rand.NextDouble() ⇒ (B – A)* rand.NextDouble() + A.

Пример:

```
Random rand = new Random();
//Случайные числа в интервале [A, B)
Console.WriteLine("Five Doubles.");
for (int ctr = 0; ctr <= 4; ctr++)
Console.Write("{0,8:N3}", ((B — A)*rand.NextDouble() + A));
Console.WriteLine();
```

9. Тип класс C#. Инкапсуляция

Что касается платформы .NET, то наиболее фундаментальной программной конструкцией является *тип класса*. Класс определяется в C# с помощью ключевого слова *class*. Формально класс — это определяемый пользователем тип, который состоит из *полей* данных (часто именуемых *переменными-членами*) и членов, оперирующих этими данными (конструкторов, свойств, методов, событий и т. п.). Все вместе поля данных класса представляют «состояние» экземпляра класса (иначе называемого *объектом*).

При желании можно распределить единственный класс, представленный в виде одного файла *.cs, на несколько файлов C#, чтобы изолировать рутинный код от более ценных полезных членов. Первый шаг состоит в добавлении *ключевого* слова *partial* к текущему определению класса и вырезании кода, который должен быть помещен в новый файл. Далее можно переместить оставшиеся члены (например, поля данных и конструкторы) в другой новый файл посредством простой операции вырезания и вставки.

При компиляции модифицированного проекта не должна быть заметна разница. Основная идея, положенная в основу *частичного* класса, реализуется только во время проектирования. Как только приложение скомпилировано, в сборке оказывается один цельный класс.

Единственное требование при определении *частичных* типов связано с тем, что разные части должны иметь одно и то же имя и находиться в пределах одного и того же пространства имен .NET.

> *Примечание.* Помните, что каждый аспект определения частичного класса должен быть помечен ключевым словом *partial.*

Первый основной принцип ООП называется *инкапсуляцией*. Концепция инкапсуляции основывается на положении, гласящем, что внутренние данные объекта не должны быть напрямую доступны через экземпляр объекта. Вместо этого, если вызывающий код желает изменить состояние объекта, то должен делать это через методы доступа (accessor, или метод get) и изменения (mutator, или метод set).

При работе с инкапсуляцией всегда следует принимать во внимание то, какие аспекты типа видимы различным частям приложения.

В частности, *типы*:

- классы,
- интерфейсы,
- структуры,
- перечисления,
- делегаты,

а также их члены:

- свойства,
- методы,
- конструкторы,
- поля,

которые определяются с использованием определенного ключевого слова *модификатора доступа*, управляющего «видимостью» элемента другим частям приложения.

Хотя в C# определены многочисленные ключевые слова для управления доступом, их значение может отличаться в зависимости от места применения (к типу или члену). В табл. 9.1, приведенной ниже, описаны роли и применение *модификаторов доступа*.

Таблица 9.1

Модификаторы доступа

Модификатор доступа	К чему может быть применен	Назначение
public	Типы или члены типов	Общедоступные (public) элементы не имеют ограничений доступа. Общедоступный член может быть доступен как из объекта, так и из любого производного класса. Общедоступный тип может быть доступен из других внешних сборок
private	Члены типов или вложенные типы	Приватные (private) элементы могут быть доступны только в классе (или структуре), в котором они определены
protected	Члены типов или вложенные типы	Защищенные (protected) элементы могут использоваться классом, который определил их, и любым дочерним классом. Однако защищенные элементы не доступны внешнему миру через операцию точки (.)
internal	Типы или члены типов	Внутренние (internal) элементы доступны только в пределах текущей сборки. Таким образом, если в библиотеке классов .NET определен набор внутренних типов, то другие сборки не смогут ими пользоваться
protected internal	Члены типов или вложенные типы	Когда ключевые слова protected и internal комбинируются в объявлении элемента, такой элемент доступен внутри определяющей его сборки, определяющего класса и всех его наследников

9.1. Определения вложенных типов

В C# (как и в других языках .NET) допускается определять тип (перечисление, класс, интерфейс, структуру или делегат) непосредственно внутри контекста класса или структуры. При этом вложенный (или «внутренний») тип считается членом охватывающего (или «внешнего») класса, и в глазах исполняющей системы им можно манипулировать как любым другим членом (полем, свойством, методом и событием). Поскольку вложенный тип является членом вклю-

чающего класса, он может иметь доступ к приватным членам включающего класса.

Когда тип включает другой тип класса, он может создавать переменные-члены этого типа, как любой другой элемент данных. Однако если вложенный тип нужно использовать вне включающего типа, его понадобится квалифицировать именем включающего типа.

9.2. Способы инкапсуляции

Инкапсуляция предоставляет способ предохранения целостности данных о состоянии объекта. Члены класса, представляющие *состояние объекта*, не должны помечаться как public. В то же время вполне допускается иметь общедоступные константы и поля только для чтения. Вместо определения общедоступных (public) полей (которые легко приводят к повреждению данных), необходимо использовать определения *приватных данных*, управление которыми осуществляется опосредованно, с применением одной из двух техник:

- введением пары методов доступа (метод get) и изменения (метод set) для каждого приватного данного;
- введением свойств класса.

Методы доступа

Роль метода get состоит в возврате вызывающему коду значения лежащих в основе статических данных. Метод set позволяет вызывающему коду изменять текущее значение лежащих в основе статических данных при условии соблюдения используемых ограничений.

Пример:

```
class Employee {
// Поля данных.
private string empName;
// Метод доступа (метод get) .
public string GetName()
{
return empName;
}
// Метод изменения (метод set) .
```

```
public void SetName(string name)
{
// Перед присваиванием проверить входное значение
//на длину имени.
if (name.Length > 15)
Console.WriteLine("Error! Name must be less than 16 characters!");
else
empName = name;
}
}
```

Эта техника требует наличия двух уникально именованных методов для управления единственным элементом данных. Приватное поле empName инкапсулировано с использованием двух методов GetName() и SetName(). Благодаря коду в методе SetName(), попытка присвоить строку длиннее 15 символов приводит к выводу на консоль жестко закодированного сообщения об ошибке.

Свойство класса

В языках .NET имеется более предпочтительный способ инкапсуляции данных с помощью свойств. Свойство позволяет вводить данные при условии соблюдения используемых ограничений в отличие от полей с модификатором public.

Прежде всего, нужно иметь в виду, что свойства — это всего лишь упрощенное представление «реальных» методов доступа и изменения. Это значит, что разработчик класса по-прежнему может реализовать любую внутреннюю логику, которую нужно выполнить перед присваиванием значения (например, преобразовать в верхний регистр, очистить от недопустимых символов, проверить границы числовых значений и т. д.).

Пример:

```
class Employee {
// Поля данных.
private string empName;
private int empID;
private float currPay;
// Свойства.
```

```
public string Name {
get { return empName; }
set {
if (value.Length > 15)
Console.WriteLine("Error! Name must be less than 16 characters!");
else
empName = value;
}
}
```

В свойстве для set используется лексема value, которая представляет входное значение, присваиваемое свойству вызывающим кодом. Эта лексема *не* является настоящим ключевым словом C#, а представляет собой то, что называется *контекстуальным ключевым словом*. Когда лексема value находится внутри контекста set, она всегда обозначает значение, присваиваемое вызывающим кодом, и всегда имеет тип, совпадающий с типом самого свойства.

При наличии свойств вызывающему коду кажется, что он имеет дело с общедоступным элементом данных; однако «за кулисами» при каждом обращении вызывается корректный get или set, обеспечивая инкапсуляцию.

> *Примечание.* В библиотеках базовых классов .NET всегда отдается предпочтение использованию для инкапсуляции свойств, а не традиционных методов доступа и изменения. Поэтому при построении специальных классов, которые интегрируются с платформой .NET, избегайте объявления традиционных методов get и set.

При инкапсуляции данных может понадобиться сконфигурировать *свойство, доступное только для чтения*. Для этого нужно просто опустить блок set. Аналогично, если требуется создать *свойство, доступное только для записи,* следует опустить блок get.

В C# также поддерживаются статические свойства. Статические члены доступны на уровне класса, а не на уровне экземпляра (объекта) этого класса. Например, предположим, что в классе Employee определен статический элемент данных для представления названия организации, нанимающей сотрудников. Инкапсулировать статическое свойство можно следующим образом.

Пример:

```
// Статические свойства должны оперировать
//статическими данными!
class Employee {
...
private static string companyName;
public static string Company {
get{ return companyName;}
set{ companyName = value;}
...
}
```

Класс C# может определять любое количество *статических членов* с использованием ключевого слова static. При этом соответствующий член должен вызываться непосредственно на уровне класса, а не на объектной ссылке. Статические члены — это элементы, задуманные (проектировщиком класса) как общие, так что нет нужды создавать экземпляр типа при их вызове.

> *Примечание.* Следует повторить: статические члены могут оперировать только статическими данными и вызывать статические методы определяющего их класса. Попытка использования нестатических данных класса или вызова нестатического метода класса внутри реализации статического члена приводит к ошибке во время компиляции.

В классе (или структуре) также могут быть определены статические поля, т. е. переменные-члены. Когда класс определяет нестатические данные (правильно называемые данными экземпляра), то каждый объект этого типа поддерживает независимую копию поля. Статические данные, с другой стороны, распределяются однажды и разделяются всеми объектами того же класса.

При проектировании любого класса C# одна из задач связана с выяснением того, какие части данных должны быть определены как статические члены, а какие — нет. Хотя на этот счет не существует строгих правил, помните, что поле статических данных разделяется между всеми объектами данного класса. Поэтому, если необходимо, чтобы часть данных совместно использовалась всеми объектами, статические члены будут самым подходящим вариантом.

10. Тип класс C#. Конструкторы класса

В C# поддерживается механизм конструкторов, которые позволяют устанавливать состояние объекта в момент его создания. Конструктор (constructor) — это специальный метод класса, который вызывается неявно при создании объекта с использованием ключевого слова new. Однако в отличие от «нормального» метода, конструктор никогда не имеет возвращаемого значения (даже void) и всегда именуется идентично имени класса, который он конструирует.

10.1. Определение конструкторов

Каждый класс C# снабжается конструктором по умолчанию. По определению такой конструктор никогда не принимает аргументов. После размещения нового объекта в памяти конструктор по умолчанию гарантирует установку всех полей в соответствующие стандартные значения.

Конструктор по умолчанию при необходимости может быть переопределен как специальный конструктор по умолчанию. В подавляющем большинстве случаев реализация конструктора класса по умолчанию намеренно остается пустой, поскольку все, что требуется — это создание объекта со значениями всех полей по умолчанию.

Обычно, помимо конструкторов по умолчанию, в классах определяются дополнительные специальные конструкторы. При этом пользователь объекта обеспечивается простым и согласованным способом инициализации состояния объекта непосредственно в момент его создания.

Пример:

```
class Car
{
// 'Состояние' объекта Car.
public string petName;
public int currSpeed;
// Специальный конструктор по умолчанию.
public Car()
{
petName = "Chuck";
```

```
currSpeed = 10;
}
// Здесь currSpeed получает значение
// по умолчанию типа int (0) .
public Car(string pn)
{
petName = pn;
}
// Позволяет вызывающему коду установить полное
//состояние Car.
public Car(string pn, int cs)
{
petName = pn;
currSpeed = cs;
}
}
```

Следует иметь в виду, что один конструктор отличается от другого (с точки зрения компилятора C#) количеством и типом аргументов. Определение методов с одним и тем же именем, но разным количеством и типами аргументов, называется *перегрузкой*. Таким образом, класс Car имеет перегруженный конструктор, чтобы предоставить несколько способов создания объекта во время объявления. В любом случае, теперь можно создавать объекты Car, используя любой из его общедоступных конструкторов.

Однако как только определен специальный конструктор, конструктор по умолчанию *удаляется* из класса и становится недоступным! Воспринимайте это так: если вы не определили специального конструктора, компилятор C# снабжает класс конструктором по умолчанию, чтобы позволить пользователю объекта размещать его в памяти с набором данных, имеющих значения по умолчанию. В случае же, когда определяется уникальный конструктор, компилятор предполагает, что вы решили взять власть в свои руки. Таким образом, чтобы позволить пользователю объекта создавать экземпляр типа посредством конструктора по умолчанию, а также специального конструктора, понадобится *явно* переопределить конструктор по умолчанию.

Пример:

```
class Motorcycle {
public int driverIntensity;
// Вернуть конструктор по умолчанию, который будет
//устанавливать
// для всех членов данных значения по умолчанию.
public Motorcycle() {}
// Специальный конструктор.
public Motorcycle(int intensity)
{ driverIntensity = intensity; }
}
}
```

10.2. Роль ключевого слова this

В языке C# имеется ключевое слово *this*, которое обеспечивает доступ к текущему экземпляру класса. Одно из возможных применений ключевого слова *this* состоит в том, чтобы разрешать неоднозначность контекста, которая может возникнуть, когда входящий параметр назван так же, как поле данных данного типа.

Пример:

```
public void SetDriverName(string name)
{ this.name = name; }
```

Для информирования компилятора необходимо установить значение поля данных текущего объекта, просто используется *this* для разрешения этой неоднозначности. Если неоднозначности нет, то использовать ключевое слово *this*, когда классу нужно обращаться к собственным данным или членам, необязательно.

> *Примечание.* Применение ключевого слова *this* внутри реализации статического члена приводит к ошибке компиляции. Статические члены оперируют на уровне класса (а не объекта), а на этом уровне нет текущего объекта, потому и не существует *this*.

Другое применение ключевого слова *this* состоит в проектировании класса, использующего технику под названием *сцепление конструкторов* или *цепочка конструкторов* (constructor chaining). Этот шаблон проектирования полезен, когда имеется класс, определяющий несколько конструкторов.

Подход предусматривает назначение конструктора, который принимает *максимальное количество аргументов*, в качестве «ведущего конструктора», с реализацией внутри него необходимой логики проверки достоверности. Остальные конструкторы смогут использовать ключевое слово *this*, чтобы передать входные аргументы ведущему конструктору и при необходимости предоставить любые дополнительные параметры. В результате, беспокоиться придется только о поддержке единственного конструктора для всего класса, в то время как остальные конструкторы остаются в основном пустыми. При связывании конструкторов в цепочку обратите внимание, что ключевое слово *this* располагается вне тела конструктора (и отделяется от его имени двоеточием).

Пример:

```
class Motorcycle {
public int driverIntensity;
public string driverName;
// Связывание конструкторов в цепочку.
public Motorcycle() {}
public Motorcycle(int intensity): this(intensity, "") {}
public Motorcycle(string name): this (0, name) {}
// Это ведущий конструктор , выполняющий всю
//реальную работу.
public Motorcycle(int intensity, string name)
{
if (intensity > 10)
{
intensity = 10;
}
driverIntensity = intensity;
driverName = name;
}
}
```

В построении цепочки конструкторов замечательно то, что этот шаблон программирования работает с любой версией языка C# и платформы .NET.

10.3. Использование свойств внутри определения класса

В этом разделе (в примерах для простоты) были использованы поля с модификатором public. Но в реальном случае дело обстоит иначе, и поля используются с модификатором private, поскольку свойства позволяют вводить данные при условии соблюдения используемых ограничений, в отличие от общедоступных полей.

В то же время, конструктор должен, получая входные параметры, проверять корректность данных и только после этого выполнять присваивания внутренним приватным полям. Тем самым, свойства и ведущий конструктор предпринимают одну и ту же проверку ошибок. В результате получается дублирование кода.

Чтобы упростить код и разместить всю проверку ошибок в одном месте, для установки и получения данных внутри класса разумно *всегда* использовать свойства вместо приватных полей. Во многих случаях единственное место, где можно напрямую обращаться к приватным данным — это внутри самого свойства.

11. Тип класс C#. Наследование

Второй принцип ООП — наследование, есть аспект ООП, облегчающий повторное использование кода. При установке между классами отношения «является» строится зависимость между двумя или более типами классов. Базовая идея, лежащая в основе классического наследования, заключается в том, что новые классы могут создаваться с использованием существующих классов в качестве отправной точки.

Отношение «является» (формально называемое *классическим наследованием*) позволяет строить новые определения классов, расширяющие функциональность существующих классов. Существующий класс, который будет служить основой для нового класса, называется *базовым* или *родительским* классом. Назначение базового класса состоит в определении всех общих данных и членов для классов, которые расширяют его. Расширяющие классы формально называются *производными* или *дочерними* классами. В C# для установки между классами отношения «является» используется операция двоеточия в определении класса.

Примечание. Хотя конструкторы обычно определяются как общедоступные, производный класс никогда не наследует конструкторы своего родительского класса.

Язык C# требует, чтобы любой конкретный класс имел в *точности* один непосредственный базовый класс. В C# невозможно создать тип класса, который напрямую унаследован от двух и более базовых классов (эта техника, поддерживаемая в неуправляемом C++, называется *множественным наследованием*). Попытка создать класс, в котором указано два непосредственных родительских класса, приводит к ошибке компиляции.

В C# поддерживается еще одно ключевое слово — *sealed*, которое *предотвращает* наследование. Если класс помечен как *sealed* (запечатанный), компилятор не позволяет наследовать от него. В пространстве имен *System* определено множество запечатанных классов.

Примечание. Структуры C# всегда неявно запечатаны. Поэтому ни унаследовать одну структуру от другой, ни класс от структуры, ни структуру от класса, не получится. Структуры могут использоваться только для моделирования отдельных атомарных, определенных пользователем типов. Для реализации отношения «является» необходимо применять классы.

Пример:

```
class Employee { // Базовый класс
// Поля данных.
private string empName;
private int empID;
private float currPay;
private int empAge;
private string empSSN;
// Свойства остаются прежними...
public string SocialSecurityNumber {
get { return empSSN; }
}
// Конструкторы.
public Employee() {}
public Employee(string name, int id, float pay)
```

```
:this(name, 0, id, pay, ""){}
public Employee(string name, int age, int id, float pay, string ssn)
{
Name = name;
Age = age;
ID = id;
Pay = pay;
SocialSecuntyNumber = ssn;
}
// Методы.
public void GiveBonus(float amount)
{ Pay + = amount; }
public void DisplayStats()
{
Console.WriteLine("Name: {0}", Name);
Console.WriteLine (11 ID : {0}", ID);
Console.WriteLine("Age: {0}", Age);
Console.WriteLine("Pay: {0}", Pay);
Console.WriteLine("SSN: {0}", SocialSecuntyNumber);
}
}
// Менеджерам нужно знать количество их опционов на акции.
class Manager : Employee {// Дочерний класс
public int StockOptions { get; set; }
}
// Продавцам нужно знать количество продаж.
class SalesPerson : Employee {// Дочерний класс
public int SalesNumber { get; set; }
}
```

Теперь, после установки отношения «является», SalesPerson и Manager автоматически наследуют все общедоступные члены базового класса Employee.

Модифицируем специальный конструктор класса Manager, применив ключевое слово *base*.

Пример:

```
public Manager(string fullName, int age, int empID, float currPay,
string ssn, int numbOfOpts)
```

```
: base (fullName, age, empID, currPay, ssn)
{
// Это свойство определено в классе Manager.
StockOptions = numbOfOpts;
}
```

Здесь ключевое слово *base* ссылается на сигнатуру конструктора (подобно синтаксису, используемому для сцепления конструкторов на единственном классе с использованием ключевого слова *this*), что всегда указывает на то, что производный конструктор передает данные конструктору непосредственного родителя. В данной ситуации явно вызывается конструктор с пятью параметрами, определенный в Employee, что избавляет от излишних вызовов во время создания экземпляра базового класса.

Специальный конструктор SalesPerson выглядит в основном идентично.

Пример:

```
// В качестве общего правила, все подклассы должны
//вызывать соответствующий конструктор базового
//класса.

public SalesPerson(string fullName, int age, int empID, float currPay,
string ssn, int numbOfSales)
: base(fullName, age, empID, currPay, ssn)
{
// Это касается нас!
SalesNumber = numbOfSales;
}
```

> *Примечание*. Ключевое слово *base* можно использовать везде, где подкласс желает обратиться к общедоступному или защищенному члену, определенному в родительском классе. Применение этого ключевого слова не ограничивается логикой конструктора.

Как только в определении класса появляется специальный конструктор, конструктор по умолчанию из класса удаляется. Следовательно, нужно переопределить конструктор по умолчанию для типов SalesPerson и Manager.

```
//Вернуть классу Manager конструктор по умолчанию.
public Manager() {}
```

Когда базовый класс определяет защищенные данные или защищенные члены, он устанавливает набор элементов, которые могут быть доступны непосредственно любому наследнику с помощью ключевого слова *protected* (защищенный).

Преимущество определения защищенных членов в базовом классе состоит в том, что производным типам больше не нужно обращаться к данным опосредованно, используя общедоступные методы и свойства.

> *Примечание*. Хотя protected-поля данных могут нарушить инкапсуляцию, объявлять protected-методы достаточно безопасно (и полезно). При построении иерархий классов очень часто приходится определять набор методов, которые используются только производными типами.

12. Тип класс C#. Полиморфизм

Третий принцип ООП — полиморфизм, есть аспект ООП, предоставляющий подклассу способ определения собственной версии метода, определенного в его базовом классе, с использованием процесса, который называется *переопределением методов и свойств* (индексаторы и события здесь не рассматриваются). Для этого нужно понять значение ключевых слов *virtual* и *override*. Если базовый класс желает определить метод или свойство, которые *могут быть* (но не обязательно) переопределены в подклассе, он должен пометить этот член класса ключевым словом *virtual*:

> *Примечание.* Методы и свойства, помеченные ключевым словом *virtual*, называются виртуальными методами и виртуальными свойствами.

> *Примечание.* Поля не могут быть виртуальными.

Когда класс желает изменить реализацию деталей виртуального метода или свойства, он делает это с помощью ключевого слова *override*.

Пример:

```
partial class Employee {
// Этот метод теперь может быть переопределен
//производным классом.
public virtual void GiveBonus(float amount)
{
currPay + = amount;
}
}

class SalesPerson : Employee
{
// Бонус продавца зависит от количества продаж.
public override void GiveBonus(float amount)
{
int salesBonus = 0;
if (numberOfSales > = 0 && numberOfSales < = 100) salesBonus = 10;
else
{
if (numberOfSales > = 101 && numberOfSales < = 200) salesBonus =
15; else salesBonus = 20;
}
base.GiveBonus(amount * salesBonus);
}
}
```

Обратите внимание на использование каждым переопределенным методом поведения по умолчанию через ключевое слово *base*. Таким образом, полностью повторять реализацию логики GiveBonus() вовсе не обязательно, а вместо этого можно повторно использовать (и, возможно, расширять) поведение по умолчанию родительского класса.

Ключевое слово *sealed* применяется к типу класса для предотвращения расширения другими типами его поведения через наследование. Иногда требуется не запечатывать класс целиком, а просто предотвратить переопределение некоторых виртуальных методов в производных типах.

Пример:

```
// SalesPerson запечатал метод GiveBonus()
class SalesPerson : Employee
{
public override sealed void GiveBonus(float amount)
{...
}
}
```

Здесь SalesPerson действительно переопределяет виртуальный метод GiveBonus(), определенный в классе Employee, однако он явно помечен как sealed. Поэтому попытка переопределения этого метода в классе-наследнике приведет к ошибке во время компиляции.

12.1. Абстрактные классы

В C# можно создать *абстрактный базовый класс*, используя ключевое слово *abstract* в определении класса. Попытка создать экземпляр *абстрактного* класса приведет к ошибке во время компиляции.

Следует понимать, что хотя *непосредственно* создать абстрактный класс нельзя, он все же присутствует в памяти, когда создан экземпляр его производного класса. Таким образом, совершенно нормально (и принято) для абстрактных классов определять любое количество *конструкторов*, вызываемых опосредованно при размещении в памяти *экземпляров производных классов*, *полей, неабстрактных членов* (с реализацией) и т. п.

Когда класс определен как *абстрактный базовый* (с помощью ключевого слова *abstract*), в нем может определяться любое количество *абстрактных членов*.

Для пометки метода как абстрактного в C# служит ключевое слово *abstract*. Методы, помеченные как *abstract*, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости).

Аналогично абстрактным методам определяются абстрактные свойства.

Пример:

```
abstract class clsBase
{
```

```
//абстрактный метод
abstract public void Describe();
//абстрактные свойства
abstract public double DoubleProp{
get;}//только для чтения
abstract public int IntProp{
set;}// только для записи
abstract public string StringProp{
get;set;}// для чтения и записи
...
}
```

Абстрактные члены могут использоваться везде в абстрактном классе, где необходимо определить член, который не предлагает реализации по умолчанию (т. е. в абстрактном классе). За счет этого вы навязываете *полиморфный интерфейс* каждому наследнику, возлагая на них задачу реализации конкретных деталей абстрактных методов.

При реализации в унаследованном классе абстрактных свойств и методов перед их заголовками нужно ставить ключевое слово *override*.

12.2. Полиморфный интерфейс

Полиморфный интерфейс абстрактного базового класса просто ссылается на его набор виртуальных и абстрактных методов. Методы, помеченные как abstract, являются чистым протоколом. Они просто определяют имя, возвращаемый тип (если есть) и набор параметров (при необходимости).

> *Примечание.* Абстрактные методы могут определяться только в абстрактных классах. Попытка поступить иначе приводит к ошибке во время компиляции.

Ниже в примере абстрактный класс Shape информирует типов-наследников о том, что у него есть метод по имени Draw(), который не принимает аргументов и ничего не возвращает. О необходимых деталях должен позаботиться наследник.

Пример:

```
abstract class Shape {
public Shape(string name = "NoName")
```

```
{ PetName = name; }
public string PetName { get; set; }
// Вынудить все дочерние классы определить свою
//визуализацию.
public abstract void Draw();
}
//С учетом этого метод Draw() в классе Circle
//теперь должен быть обязательно переопределен,
//поскольку класс Circle абстрактным типом не //является.
class Circle : Shape {
public Circle () {}
public Circle(string name) : base(name) {}
public override void Draw()
{
Console.WriteLine("Drawing {0} the Circle", PetName);
}
class Hexagon : Shape {
public Hexagon () {}
public Hexagon (string name) : base(name) {}
public override void Draw()
{
Console.WriteLine("Drawing {0} the Hexagon", PetName);
}
}
```

Хотя невозможно *напрямую* создавать экземпляры абстрактного базового класса (Shape), можно свободно сохранять ссылки на объекты любого подкласса (т. е. класса наследника) в абстрактной базовой переменной. Таким образом, созданный массив объектов Shape может хранить объекты, унаследованные от базового класса Shape (попытка поместить в массив объекты, несовместимые с Shape, приводит к ошибке во время компиляции).

Предположим, что от абстрактного базового класса Shape унаследовано еще пять классов (Triangle, Square и т. д.). Учитывая, что все элементы в массиве myShapes действительно наследуются от Shape, известно, что все они поддерживают один и тот же полиморфный интерфейс (или, говоря конкретно — все они имеют метод Draw()). Выполняя итерацию по массиву ссылок Shape, исполняющая система сама определяет, объект какого конкретного класса

наследника содержит каждый элемент массива. И в этот момент вызывается реализованная в данном классе-наследнике версия метода Draw(), рисующая либо Triangle, либо Square и т. д.

Ниже метод Main() иллюстрирует использование полиморфизма в чистом виде.

Пример:

```
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Polymorphism *****\n");
// Создать массив совместимых с Shape объектов.
Shape[] myShapes = {new Hexagon(), new Circle(), new
Hexagon("Mick"), new Circle("Beth"), new Hexagon("Linda")};
// Пройти циклом по всем элементам и
//взаимодействовать с полиморфным интерфейсом.
foreach (Shape s in myShapes)
{
s.Draw ();
}
Console.ReadLine ();
}
```

Ниже показан результат работы метода Main():

```
***** Fun with Polymorphism *****
Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

13. Тип интерфейс C#

В C# *интерфейс* (*interface*) как тип представляет собой не более чем просто именованный набор *абстрактных членов*. Абстрактные методы являются чистым протоколом, поскольку не имеют никакой стандартной реализации.

На синтаксическом уровне любой интерфейс определяется с помощью ключевого слова *interface*. В отличие от классов, базовый класс (даже System.Object) для интерфейсов никогда не указывает-

ся (хотя базовые интерфейсы указываться могут). Более того, модификаторы доступа для членов интерфейсов тоже никогда не указываются (поскольку в отличие от абстрактных классов все члены интерфейсов всегда являются по умолчанию общедоступными и абстрактными).

> *Примечание*. По соглашению имена всех интерфейсов в .NET сопровождаются префиксом в виде заглавной буквы «I». При создании собственных специальных интерфейсов рекомендуется тоже следовать этому соглашению.

Содержание конкретных членов данного класса или структуры, реализующих интерфейс, зависит от того, какое поведение моделируется с его помощью данным классом или структурой. Другими словами, интерфейс (или абстрактный метод) выражает *поведение*, которое данным классом или структурой выбрано для его поддержки. Каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений. В библиотеках базовых классов .NET поставляются сотни предопределенных типов интерфейсов, которые реализуются в различных классах и структурах.

13.1. Сравнение интерфейсов и абстрактных базовых классов

Полиморфный интерфейс, предоставляемый абстрактным родительским классом, обладает одним серьезным ограничением: определяемые в абстрактном родительском классе члены поддерживаются *только в производных (наследуемых) типах*. Поэтому настраивать типы в разных иерархиях классов так, чтобы они поддерживали один и тот же полиморфный интерфейс, невозможно. В то же время интерфейс может быть реализован в любом типе или структуре, в любой иерархии и внутри любого пространства имен или сборки (написанной на любом языке программирования .NET). Очевидно, что это делает интерфейсы *чрезвычайно* полиморфными.

Рассмотрим стандартный интерфейс .NET по имени ICloneable, определенный в пространстве имен System. Этот интерфейс имеет единственный метод Clone ():

Пример:

```
public interface ICloneable {
object Clone ();
}
```

Если заглянуть в документацию .NET Framework 4.0 SDK, можно обнаружить, что очень многие по виду несвязанные типы (System.Array, System.Data.SqlClient.SqlConnection, System.OperatingSystem, System.String и т. д.) реализуют этот интерфейс. В результате, хотя у этих типов нет общего родителя (кроме System.Object), с ними все равно можно работать полиморфным образом через интерфейс ICloneable.

Другим ограничением традиционных абстрактных базовых классов является то, что в каждом производном типе должен обязательно поддерживаться соответствующий набор абстрактных членов и предоставляться их реализация, даже если для некоторых производных типов в этом нет никакого смысла.

Предположим, что в базовом классе Shape определен новый абстрактный метод по имени GetNumberOfPoints (), позволяющий производным типам возвращать информацию о количестве вершин, которое требуется для визуализации фигуры.

Пример:

```
abstract class Shape {
// Каждый производный класс теперь должен
// обязательно поддерживать такой метод!
public abstract byte GetNumberOfPoints();
}
```

Это не имеет смысла для наследников Shape, таких как Circle или ThreeDCircle, у которых нет вершин.

13.2. Реализация интерфейса

Чтобы расширить функциональность какого-то класса или структуры за счет обеспечения в нем поддержки интерфейсов, необходимо в его определении предоставить список соответствующих интерфейсов, разделенных запятыми.

Следует иметь в виду, что непосредственный базовый класс должен быть обязательно перечислен в этом списке первым, сразу же после двоеточия. Когда тип класса наследуется прямо от System.Object, допускается перечислять только лишь поддерживаемый им интерфейс или интерфейсы, поскольку компилятор C# автоматически расширяет типы классов возможностями System.Object в случае, если не было указано иначе. Из-за того, что структуры всегда наследуются от класса System.ValueType, интерфейсы просто должны перечисляться после определения структуры.

Определить, поддерживает ли данный тип тот или иной интерфейс, можно с использованием ключевого слова *as* или *is*:

- в первом случае, если объект удается интерпретировать как указанный интерфейс, то возвращается ссылка на интересующий интерфейс, а если нет, то ссылка null;
- во втором случае, если запрашиваемый объект несовместим с указанным интерфейсом, возвращается значение false, а если совместим, то true.

Благодаря тому, что интерфейсы являются допустимыми типами .NET, можно создавать методы, принимающие интерфейсы в качестве параметров, а также использовать и в качестве возвращаемых значений методов.

Поскольку один и тот же интерфейс может быть реализован во множестве типов, даже если они находятся не в одной и той же иерархии классов и не имеют никакого общего родительского класса, помимо System. Object, можно выполнять проход по массиву объектов, совместимых с этим интерфейсом, несмотря на разницу между иерархиями классов.

Пример:

```
// Этот интерфейс определяет поведение "наличие вершин".
public interface IPointy
{
byte GetNumberOfPoints();
}
```

Создадим в текущем проекте три новых типа класса, два из которых (Knife (нож) и Fork (вилка)) будут представлять кухонные принадлежности, а третий (PitchFork (вилы)) — инструмент для работы в саду.

Имея определения типов PitchFork, Fork и Knife, можно определить массив объектов, совместимых с IPointy. Поскольку все эти члены поддерживают один и тот же интерфейс, можно выполнять проход по массиву и интерпретировать каждый его элемент как совместимый с IPointy объект, несмотря на разницу между иерархиями классов.

Пример:

```
static void Main(string[] args)
{
// В этом массиве могут содержаться только типы,
// которые реализуют интерфейс IPointy.
IPointy[] myPointyObjects = {new Hexagon(),
new Knife(), new Triangle(), new Fork(), new PitchFork() };
```

Интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, в иерархии интерфейсов, когда какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своего родителя (или родителей). Конечно, в отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов.

Использовать иерархию интерфейсов может быть удобно, когда нужно расширить функциональность определенного интерфейса без нарушения уже существующих кодовых баз.

В заключение важно уяснить, что интерфейсы являются фундаментальным компонентом .NET Framework. Какого бы типа приложение не разрабатывалось (веб-приложение, приложение с настольным графическим интерфейсом, библиотека доступа к данным и т. п.), работа с интерфейсами будет обязательной частью этого процесса.

14. Файловый ввод-вывод и сериализация объектов

14.1. Исследование пространства имен System.IO

Пространство имен System.IO в .NET — это область библиотек базовых классов, относящаяся к службам файлового ввода-вывода, а также ввода-вывода из памяти. Многие типы из пространства имен

System.IO сосредоточены на программных манипуляциях физическими каталогами и файлами. Дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также области памяти.

В табл. 14.1 кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности System.IO.

Таблица 14.1

Ключевые члены пространства имен System.IO

Неабстрактные классы ввода-вывода	Назначение
BinaryReader	Позволяют сохранять и извлекать элементарные типы данных (целочисленные, булевские, строковые и т. п.) в двоичном виде
BinaryWriter	
BufferedStream	Предоставляет временное хранилище для потока байтов, которые могут затем быть перенесены в постоянные хранилища
Directory	Используются для манипуляций структурой каталогов машины. Тип Directory представляет функциональность, используя статические члены. Тип DirectoryInfo обеспечивает аналогичную функциональность через действительную объектную ссылку
DirectoryInfo	
DriveInfo	Предоставляет детальную информацию относительно дисковых устройств, используемых данной машиной
File	Служат для манипуляций множеством файлов данной машины. Тип File представляет функциональность через статические члены. Тип FileInfo обеспечивает аналогичную функциональность через действительную объектную ссылку
FileInfo	
FileStream	Обеспечивает произвольный доступ к файлу (т. е. возможности поиска) с данными, представленными в виде потока байт
FileSystemWatcher	Позволяет отслеживать модификации внешних файлов в определенном каталоге
MemoryStream	Обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле

Окончание табл. 14.1

Неабстрактные классы ввода-вывода	Назначение
Path	Выполняет операции над типами System.Strmg, содержащими информацию о пути к файлу или каталогу в независимой от платформы манере
StreamWriter	Используются для хранениия (и извлечения) текстовой информации из файла. Не поддерживают произвольного доступа к файлу
StreamReader	
StringWriter	Подобно классам StreamWriter/StreamReader, эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл
StringReader	

В дополнение к этим конкретным типам классов в Systera.IO определено несколько перечислений, а также набор абстрактных классов (т. е. Stream, TextReader и TextWriter), которые определяют разделяемый полиморфный интерфейс для всех наследников.

Перечисление FileMode (описание членов дано в табл. 14.2)

Пример:

```
public enum FileMode
{
CreateNew,
Create,
Open,
OpenOrCreate,
Truncate,
Append
}
```

Таблица 14.2

Члены перечисления FileMode

Член	Назначение
CreateNew	Информирует операционную систему о создании нового файла. Если файл уже существует, генерируется исключение lOException
Create	Информирует операционную систему о создании нового файла. Если файл уже существует, он будет перезаписан

Окончание табл. 14.2

Член	Назначение
Open	Открывает существующий файл. Если файл не существует, генерируется исключение FileNotFoundException
OpenOrCreate	Открывает файл, если он существует; в противном случае создает новый
Truncate	Открывает файл и усекает его до нулевой длины
Append	Открывает файл, переходит в его конец и начинает операции чтения (этот флаг может быть использован только с потоками, доступными лишь для чтения). Если файл не существует, то создается новый

Перечисление FileAccess — используется для определения поведения чтения/записи лежащего в основе потока.

Пример:

```
public enum FileAccess {
Read,
Write,
ReadWrite
}
```

Перечисление FileShare — указывает, как файл может быть разделен с другими файловыми дескрипторами.

Пример:

```
public enum FileShare {
Delete,
Inheritable,
None,
Read,
ReadWrite,
Write
}
```

На этом знакомство с фундаментальными операциями ввода-вывода, предлагаемыми платформой .NET, завершим. Для более подробного их изучения следует обратиться к литературе [1–3].

14.2. Сериализация объектов

В мире манипуляций вводом-выводом *поток* (stream) представляет порцию данных, протекающую от источника к цели. Потоки

предоставляют общий способ взаимодействия с *последовательностью байт* независимо от того, какого рода устройство (файл, сеть, соединение, принтер и т. п.) хранит или отображает эти байты.

> *Примечание.* Концепция потока не ограничена файловым вводом-выводом. Точности ради, следует отметить, что библиотеки .NET предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Термин *сериализация* описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (т. е. файловом потоке и потоке в памяти). Последовательность сохраняемых данных содержит всю необходимую информацию, необходимую для реконструкции (или десериализации) состояния объекта, с целью последующего использования. Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями. Во многих случаях сохранение данных приложения с использованием служб сериализации выливается в код меньшего объема, чем применение классов для чтения/записи из пространства имен System.IO

Метаданные и атрибуты

Одной из особенностей платформы .NET является то, что программы, написанные под нее, не превращаются в монотонные последовательности машинных инструкций, разобрать которые — весьма нетривиальная задача в общем случае, а наоборот, сохраняют первоначальную структуру (включая иерархию классов и т. п.), а также снабжаются подробной информацией о том, где и что лежит и что из себя представляет.

Такая информация, такие данные о данных (о программе), называются *метаданными*. Часть метаданных генерируется компилятором. Часть можно задавать самому в программном коде при помощи *атрибутов*. Используется она тоже всюду: и средой исполнения, и программистом (по мере необходимости) с помощью отражения (reflection) или рефлексии.

Атрибуты — это средство привязки информации к различным сущностям: методам, свойствам, типам и даже целым сборкам. По своей сути атрибут — это экземпляр класса унаследованного от

System.Attribute. Можно создавать свои атрибуты, с помощью наследников от класса System.Attribute.

Присоединить атрибут к сущности можно путем добавления его имени перед ней в квадратных скобках. Назначаемый атрибут инициализируется вызовом конструктора с соответствующими параметрами. Класс атрибута должен иметь хотя бы один public-конструктор. Если атрибут принимает некоторые значения в качестве параметров, то в таком случае эти значения нужно указать в круглых скобках после имени атрибута.

Чтобы запросить тип или член о прикрепленных к ним атрибутах, нужно применить *отражение* (reflection). Отражение — это функция, позволяющая в период выполнения программы динамически определять характеристики типов в приложении, в том числе и заданных атрибутами.

Для реализации отражения библиотека NET Framework предоставляет несколько классов, базовым для которых служит класс отражения System.Reflection.

Основные методы отражения, используемые для запроса атрибутов, предоставляются классами System.Reflection.MemberInfo и System.Reflection.Assembly. Так, метод GetCustomAttributes позволяет определить атрибуты, присущие данному объекту.

Конфигурирование объектов для сериализации

Чтобы сделать объект доступным для служб сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом [Serializable]. Если выясняется, что некоторый тип имеет члены-данные, которые не должны (или не могут) участвовать в схеме сериализации, можно пометить такие поля атрибутом [NonSerialized]. Это помогает сократить размер хранимых данных, при условии, что в сериализуемом классе есть переменные-члены, которые не следует «запоминать» (например, фиксированные значения, случайные значения, кратковременные данные и т. п.).

Когда объект сериализуется, среда CLR учитывает все связанные объекты, чтобы гарантировать корректное сохранение данных. Этот набор связанных объектов называется *графом объектов,* графы объектов представляют простой способ документирования набора отношений между объектами, и эти отношения не обязательно

отображаются на классические отношения ООП (вроде отношений «является» и «имеет»), хотя достаточно хорошо моделируют эту парадигму.

Каждый объект в графе получает уникальное числовое значение. Нужно иметь в виду, что числа, назначенные объектам в графе, являются произвольными и не имеют никакого значения для внешнего мира. Как только каждому объекту присвоено числовое значение, граф объектов может записать все наборы зависимостей каждого объекта.

Атрибут [Serializable] не может наследоваться от родительского класса. Поэтому при наследовании типа, помеченного [Serializable], дочерний класс также должен быть помечен [Serializable] или же его нельзя будет сохранить в потоке. Фактически все объекты в графе объектов должны быть помечены атрибутом [Serializable]. Попытка сериализовать несериализуемый объект с использованием BinaryFormatter или SoapFormatter приводит к исключению SerializationException во время выполнения.

Рассмотрим класс по имени Radio, помеченный атрибутом [Serializable], у которого исключается одна переменная-член (radioID), помеченная атрибутом [NonSerialized] и потому не сохраняемая в специфицированном потоке данных, и два дополнительных типа, представляющих базовые классы JamesBondCar и Car (оба они также помечены атрибутом [Serializable]).

Пример:

```
//Обратите внимание, что в каждом из этих классов поля данных
// определены как public, это сделано для упрощения примера.
[ Senalizable]
public class Radio {
public bool hasTweeters;
public bool hasSubWoofers;
public double[] stationPresets;
[NonSerialized]
public string radioID = "XF-552RR6";
}
//
[Senalizable]
public class Car {
```

```
public Radio theRadio = new Radio();
public bool isHatchBack;
}
//
[Senalizable]
public class JamesBondCar : Car {
public bool canFly;
public bool canSubmerge;
}
```

Какого определения полей данных типа требуют различные форматеры, чтобы сериализовать их в поток? Если вы сохраняете состояние объекта, используя BinaryFormatter или SoapFormatter, то разницы никакой.

Эти типы запрограммированы для сериализации всех сериализуемых полей типа, независимо от того, представлены они общедоступными полями, приватными полями или приватными полями с соответствующими общедоступными свойствами. Если есть элементы данных, которые не должны сохраняться в графе объектов, можно выборочно пометить общедоступные или приватные поля атрибутом [NonSerialized], как сделано со строковыми полями в типе Radio.

Сериализация объектов

Службы сериализации .NET позволяют сохранять граф объектов в различных форматах. Перечисленные возможности представлены следующими классами:

- BinaryFormatter;
- SoapFormatter;
- XmlSerializer.

Независимо от того, какой форматер выбран, каждый из них наследуется непосредственно от System.Object, так что они не разделяют общего набора членов от какого-то базового класса сериализации. Однако типы BinaryFormatter и SoapFormatter поддерживают общие члены через реализацию интерфейсов IFormatter и IRemotingFormatter.

Каким образом форматеры добиваются *точности типов* (type fidelity)? Когда используется тип BinaryFormatter, он сохраняет не

только данные полей объектов из графа, но также полное квалифицированное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер общедоступного ключа и культура). Эти дополнительные элементы данных делают BinaryFormatter идеальным выбором, когда необходимо передавать объекты по значению (т. е. полные копии) между границами машин для использования в .NET-приложениях.

Этот тип определен в пространстве имен System.Runtime. Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll. Таким образом, чтобы получить доступ к этому типу, необходимо указать следующую директиву using.

Пример:

```
// Получить доступ к BinaryFormatter в mscorlib.dll.
using System.Runtime.Serialization.Formatters. Binary;
```

Тип SoapFormatter сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб). Этот тип определен в пространстве имен System.Runtime. Serialization.Formatters.Soap, находящемся в *отдельной сборке*. Поэтому для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на System.Runtime.Serialization.Formatters. Soap.dll, используя диалоговое окно Add Reference (Добавить ссылку) в Visual Studio 2010, и затем указать следующую директиву using.

Пример:

```
// Необходима ссылка на System.Runtime.
//Serialization.Formatters.Soap.dll
using System.Runtime.Serialization.Formatters.Soap;
```

Для сохранения дерева объектов в документе XML имеется тип XmlSerializer. Чтобы использовать этот тип, нужно указать директиву using для пространства имен System.Xml.Serialization и установить ссылку на сборку System.Xml.dll.

Пример:

```
// Определено внутри System.Xml.dll.
using System.Xml.Serialization;
```

Если необходимо сохранить состояние объекта так, чтобы его можно было использовать в любой операционной системе (Windows XP, Mac OS X и различных дистрибутивах Linux), на любой платформе приложений (.NET, Java Enterprise Edition, COM и т. п.) или в любом языке программирования, придерживаться полной точности типов не следует, поскольку нельзя рассчитывать, что все возможные адресаты смогут понять специфичные для .NET типы данных. Учитывая это, SoapFormatter и XmlSerializer являются идеальным выбором, когда требуется гарантировать как можно более широкое распространение объектов.

Сериализация объектов с использованием BinaryFormatter

Тип BinaryFormatter сериализует состояние объекта в поток, используя компактный двоичный формат.

Двумя ключевыми методами типа BinaryFormatter, о которых следует знать, являются Serialize() и Deserialize():

- Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;
- Deserialize() преобразует сохраненную последовательность байт в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить этот экземпляр в файле *.dat. Первая задача — создание самого файла *.dat. Для этого можно создать экземпляр типа System. IO.FileStream. Затем следует создать экземпляр BinaryFormatter и передать ему FileStream и граф объектов для сохранения.

Пример:

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Binary и System.IO'
static void Main(string[] args)
{
Console.WnteLine("***** Fun with Ob]ect Serialization *****\n");
// Создать JamesBondCar и установить состояние.
JamesBondCar jbc = new JamesBondCar();
jbc.canFly = true;
jbc.canSubmerge = false;
jbc.theRadio.stationPresets = new double[](89.3, 105.1, 97.1);
jbc.thePadio.hasTweeters = true;
```

```
// Сохранить объект в указанном файле в двоичном формате.
SaveAsBmaryFormat (jbc,"CarData.dat" ) ;
Console.ReadLine();
}
```

Метод SaveAsBinaryFormat() реализован следующим образом:

```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
// Сохранить объект в файл CarData.dat в двоичном виде.
BinaryFormatter binFormat = new BmaryFormatter();
using(Stream fStream = new FileStream(fileName,
FileMode.Create, FileAccess.Write, FileShare.None))
{
binFormat.Serialize(fStream, objGraph);
}
Console.WriteLine(" => Saved car in binary format");
}
```

Десериализация объектов с использованием BinaryFormatter

Теперь предположим, что необходимо прочитать сохраненный объект JamesBondCar из двоичного файла обратно в объектную переменную. После открытия файла CataData.dat (методом File.OpenRead()) просто вызвать метод Deserialize() класса BinaryFormatter. Нужно иметь в виду, что Deserialize() возвращает объект общего типа System.Object, так что понадобится применить явное приведение, как показано ниже:

Пример:

```
static void LoadFromBinaryFile(string fileName)
{
BinaryFormatter binFormat = new BinaryFormatter();
// Прочитать JamesBondCar иа двоичного файла.
using(Stream fStream = File.OpenRead(fileName))
{
JamesBondCar carFromDisk =
(JamesBondCar)binFormat.Deserialize(fStream);
Console.WriteLine("Can this car fly? : (0)", carFromDisk.canFly);
}
}
```

При вызове Deserialize() ему передается тип-наследник Stream, представляющий местоположение сохраненного графа объектов. Приведя возвращенный объект к правильному типу, получается объект в том состоянии, в каком он был на момент сохранения.

Часть II. ЛАБОРАТОРНЫЕ РАБОТЫ

Задания лабораторных работ, которые приводятся в части II, используют консольные программы и, соответственно, *форматированный* ввод-вывод осуществляется с помощью класса System.Console (см. раздел I–6).

Нумерация лабораторных работ в части II сквозная.

> *Примечание*. Для ввода и вывода в окно (консоль) запущенной программы в качестве десятичного разделителя используется символ (,), в то время как только символ (.) воспринимается как правильный десятичный разделитель для кода программы в редакторе программы.

Поскольку программы разрабатываются в среде Visual Studio 2010, то при работе с ней следует помнить следующее:

- *решения и проекты* — это контейнеры, которые Visual Studio 2010 использует для размещения и группировки кода, который пишется в интегрированной среде;
- *контейнеры* — это объекты, внутри которых размещены другие объекты;
- *решения* — это виртуальные контейнеры; они группируют свойства, относящиеся к одному (или нескольким), содержащимся в решении, проектам. Внутри интегрированной среды можно настроить несколько свойств на уровне решения.

Решения не обрабатываются компилятором — *программой,* переводящей текст программы на языке высокого уровня в эквивалентную программу на машинном языке.

- *проекты* имеют и виртуальный, и физический характеры. Помимо функционирования в качестве организационных единиц для вашего кода они также однозначно соответствуют результатам, получаемым на выходе компилятора. Иначе говоря, Visual Studio 2010 превращает проекты в *откомпилированный код.* Каждый проект приводит к созданию .NET-компонента (такого, как файл

с расширением dll — динамическая библиотека, или exe — исполняемый файл приложения).

С точки зрения программирования — все, что делается внутри Visual Studio 2010, происходит на уровне решения. Как мы уже упоминали, сами по себе решения не делают ничего, кроме того, что служат контейнерами высокого уровня для прочих элементов. Проекты — это самые очевидные элементы, которые можно поместить внутрь решений, но решения также могут содержать разнообразные файлы, которые необходимы для самого решения (такие, как документы Readme и диаграммы проекта). Фактически в решение может быть добавлен файл любого типа. Однако решения не могут содержать *другие решения*. Кроме того, Visual Studio 2010 может открыть только *одно решение* единовременно. Если нужно работать над несколькими решениями одновременно, то придется запустить еще один экземпляр Visual Studio 2010.

Поскольку мы знаем, что решения не могут компилироваться, то совершенно очевидно, что добавленные на уровне решения файлы не имеют практического значения (с точки зрения компиляции). Однако имеются различные причины, по которым можно добавлять в решение элементы. Например, это удобный способ хранения документации, которая относится ко всему решению в целом. Поскольку можно добавить в решение файл любого типа, то это может быть и документация, и заметки для других разработчиков, и проектные спецификации, и даже файлы исходного кода из других решений (которые могут иметь отношение к текущей работе).

Для разработки решения сначала создают проект. Поскольку проекты в Visual Studio 2010 независимо от решения загружаться не могут, то создание проекта приведет к одновременному созданию и решения.

1. КОНСТРУКЦИИ ПРОГРАММИРОВАНИЯ НА C#

В этом разделе приводятся лабораторные работы № 1–4, задания которых используют конструкции языка C# безотносительно к объектно-ориентированному программированию. Подобные схожие конструкции есть во многих языках программирования высокого уровня, в частности, в Си-подобных языках, Паскале и других.

Как уже говорилось в части I раздела 5, в языке C# требуется, чтобы все члены данных и методы содержались внутри определения типа.

Поскольку программы разрабатываются в среде Visual Studio 2010, то программируя задания лабораторных работ в части II раздела 1, для этой цели необходимо создать консольный проект C# и использовать класс Program вместе с методом Main(). Если требуется, то в класс Program можно дополнительно ввести другие статические методы (в частности, для большей модульности программного кода).

В арифметических выражениях используются свойства и методы класса System.Math, в операциях со строками — свойства и методы класса System.String (см. часть I, раздел 4).

Примечание. При написании программы для лабораторного задания следует взвешенно выбирать используемые типы переменных. В частности, для численных расчетов рекомендуется использовать переменные типа double. Целочисленные типы следует использовать только в том случае, когда это вызвано спецификой конкретных данных или синтаксисом языка C#.

Лабораторная работа № 1

Алгоритмизация линейных вычислительных процессов

Задание 1. Стандартные типы данных и выражения

Разработать программу для нахождения параметра фигуры по заданным вещественным значениям параметров, вводимым с клавиатуры.

Вариант 1. Составить программу для нахождения полной поверхности пирамиды, в основании которой лежит квадрат со стороной b. Высота пирамиды b.

Вариант 2. Составить программу для нахождения объема пирамиды, в основании которой лежит квадрат со стороной b. Высота пирамиды h.

Вариант 3. Составить программу для нахождения объема пирамиды, в основании которой лежит равносторонний треугольник со стороной b. Высота пирамиды h.

Вариант 4. Составить программу для нахождения площади треугольника по двум сторонам a, b и углу α между ними.

Вариант 5. Составить программу для нахождения объема и полной поверхности цилиндра. Высота цилиндра h, радиус основания r.

Вариант 6. Составить программу для нахождения объема прямой призмы, в основании которой лежит прямоугольный треугольник с катетами a и b. Высота призмы h.

Вариант 7. Составить программу для нахождения полной поверхности прямой призмы, в основании которой лежит квадрат со стороной b. Высота призмы h.

Вариант 8. Составить программу для нахождения полной поверхности прямой призмы, в основании которой лежит равносторонний треугольник со стороной b. Высота призмы h.

Вариант 9. Составить программу для нахождения объема прямой призмы, в основании которой лежит квадрат со стороной b. Высота призмы h.

Вариант 10. Составить программу для нахождения объема прямой призмы, в основании которой лежит равносторонний треугольник со стороной a. Высота призмы h.

Вариант 11. Составить программу для нахождения объема и полной поверхности куба, ребро которого b.

Вариант 12. Составить программу для нахождения площади кольца, внутренний диаметр которого d, а внешний D.

Вариант 13. Составить программу для нахождения поверхности сферы и объема шара. Радиус шара r.

Вариант 14. Составить программу для нахождения полной поверхности прямой призмы, в основании которой лежит прямоугольный треугольник с катетами a и b. Высота призмы h.

Вариант 15. Составить программу для нахождения объема цилиндрической трубы со стенками толщиной t. Внешний диаметр d, высота трубы h.

Задание 2. Арифметические выражения

Разработать программу вычисления по приведенной формуле $F(x)$ и вывести на экран результат в указанном формате. Функцию $F(x)$ запрограммировать в виде статического метода в классе Program. Исходные данные вводятся с клавиатуры. Сверить полученный результат с ответом.

Вариант 1

$$F = \left(\sqrt{\frac{x+1}{x-1}}\right)^3 + 2a\ln(\cos\sqrt[5]{4-x}) + \operatorname{ctg} a - \frac{\lg x^2}{5!},$$

где $5! = 1\cdot 2\cdot 3\cdot 4\cdot 5$, $\lg x = \dfrac{\ln x}{\ln 10}$.

Проверка: $x = 3{,}8 \quad a = -4{,}82 \quad F = 5{,}13.$

Вариант 2

$$F = \left(\sin^2 x + \left(\cos x - \frac{\sin^4 x^2 + x^2}{x} - \frac{x^4 + 0{,}96x - 1}{x^4 + x^3 - 2{,}37}\right)^3 + \sqrt{x}\right)^4.$$

Проверка: $x = 0{,}85 \quad F = 4{,}844.$

Вариант 3

$$F = \left(a + x + \sin^2 a\right)\cos x^{-1} + \frac{a}{\sin a + x\cos x^{-2}} - e^{a-x}.$$

Проверка: $x = 4{,}66; \quad a = 0{,}83; \quad F = 6{,}028.$

Вариант 4

$$F = \sin\left(\frac{\pi + x}{20}\right) + \sqrt{x + \sqrt[a]{x+1}} - 7{,}2xa\ln\left(\sin^2 x^{-3}\right).$$

Проверка: $x = 4{,}87; \quad a = 3; \quad F = 1{,}0021\cdot 10^3.$

Вариант 5

$$F = x\left(b\left(\frac{x\cdot a}{x+a}\right)^b\right)^x - \frac{\lg a - 1}{\sqrt{a^2 - x^2 + b^2}} - \operatorname{ctg}\sqrt{x^2 - b},$$

где $\lg x = \dfrac{\ln x}{\ln 10}$.

Проверка: $x = 1{,}15; \quad a = 3{,}71; \quad b = 0{,}89; \quad F = -2{,}9615\cdot 10^{-1}.$

Вариант 6

$$F = 8{,}36 \cdot 10^{3{,}5} + 2\sin^2(\pi + a)^{-3} + \sqrt{|x|}\,\frac{\sin x^2}{x + e^x} - \operatorname{ctg}\left|\frac{x}{a}\right|.$$

Проверка: $x = -0{,}17; \quad a = 3{,}81; \quad F = 2{,}64 \cdot 10^4.$

Вариант 7

$$F = 0{,}836 + \left(\frac{a}{\sqrt{\cos^3 a}} - 1\right)\operatorname{arctg} a^4 + 3a - \left(\frac{a}{a-8} + 2{,}306a^2\right).$$

Проверка: $a = 0{,}672; \quad F = 1{,}896.$

Вариант 8

$$F = e^{-2\pi \cdot a\sqrt{x^2+1{,}2}} + 13x^{\frac{3}{2}} + \sqrt[a]{xa} + \ln\left|\operatorname{tg}\frac{x}{2}\right|.$$

Проверка: $x = 0{,}85; \quad a = 2{,}34; \quad F = 1{,}0736 \cdot 10^1.$

Вариант 9

$$F = \left(\sqrt{a\sqrt{ax}} - (ax)^{3/4}\right)e^{-\left(a^2+x\right)} + \ln\ \operatorname{tg}\frac{a}{x}.$$

Проверка: $x = 1{,}87; \quad a = 0{,}53; \quad F = -1{,}265.$

Вариант 10

$$F = \frac{\ln\sqrt{\left|\dfrac{a^3+a}{a+2} - a^2\right|} + e^{-\left(\sqrt{2^{\sqrt{a}-2}}+2\right)}}{\sin\left(\dfrac{\pi a}{12}\right) + 1}.$$

Проверка: $a = 1{,}87; \quad F = 1{,}367 \cdot 10^{-1}.$

Вариант 11

$$F=\frac{\left(e^{x}\right)^{a}+2,73\left(a^{3}\sqrt[3]{x}+1,78\left|x\right|^{1,4}\right)}{\left|x+a\right|+tg\left(\frac{\pi}{2}a\right)}.$$

Проверка: $x=1,25;\quad a=-2,14;\quad F=-3,31\cdot10^{1}.$

Вариант 12

$$F=3\sin x+x^{4}\left(1+\frac{x-a^{x}}{x+\frac{a}{x}}\right)\operatorname{tg}^{2}x-\ln\left|\frac{x}{a}\right|.$$

Проверка: $x=-0,25;\quad a=0,31;\quad F=-5,266\cdot10^{-1}.$

Вариант 13

$$F=\sqrt{x+\sqrt[3]{x+\sqrt{a\ln x}}}+\frac{\sqrt{x}\sin x^{-2}}{x+e^{ax}}.$$

Проверка: $x=3,89;\quad a=4,01;\quad F=2,394.$

Вариант 14

$$F=\frac{\ln x^{2}}{\cos x}+\sqrt{\lg\left|tg\frac{x}{2}\right|+\frac{a-\sqrt{x^{a}}}{\left|a-x\right|}},$$

где $\lg x=\dfrac{\ln x}{\ln 10}.$

Проверка: $x=0,97;\quad a=2,34;\quad F=7,44\cdot10^{-1}.$

Вариант 15

$$F=\left(\sqrt{\frac{1-a}{1+a}}\right)^{2,9a+7}-\frac{7,2\cdot10^{1,3}\sqrt{a}}{\cos e^{\sin x}}-\operatorname{tg}\left(\frac{\pi}{30}-x\right).$$

Проверка: $x=2,17;\quad a=0,11;\quad F=7,144\cdot10^{1}.$

Вариант 16

$$F = 1{,}736\left|\ \ln(tg\,\frac{x}{2})\ \right| - \sqrt{\operatorname{arctg}\,\frac{e^{x} - e^{-x}}{2}}.$$

Проверка: $x = 2{,}4; \quad F = 4{,}61 \cdot 10^{-1}.$

Лабораторная работа № 2

Разветвляющиеся вычислительные процессы

Задание 1

Ввести с клавиатуры 3 целых числа. Определить и выдать на экран те числа, которые попадают в диапазон от 2 до 5. Если число попадает на границу интервала, то сообщить об этом.

Задание 2

Дано натуральное число n ($n \le 100$), определяющее возраст человека (в годах). Дать для этого числа наименования «год», «года», «лет». Например, 1 год, 23 года, 46 лет и т. д.

Задание 3

Написать программу решения квадратного уравнения $ax^2 + b + c = 0$ по введенным с клавиатуры вещественным коэффициентам, используя условный оператор.

В зависимости от введенных коэффициентов уравнение может иметь 2 действительных корня, 2 комплексных корня, один действительный корень, множество решений и не иметь решения.

При $a = b = c = 0$ корней бесчисленное множество (x — любое).

При $a = b = 0$, $c \ne 0$ уравнение не имеет корней.

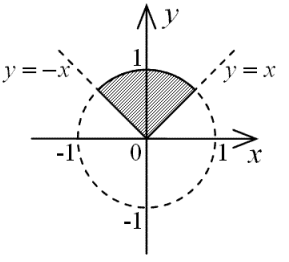
При $a = 0$, $b \ne 0$, (c — любое) квадратное уравнение превращается в линейное и имеет один корень $x = -c/b$.

При $a \ne 0$ надо вычислить дискриминант $d = b^2 - 4ac$. Если $d < 0$, то уравнение не имеет действительных корней, а имеет комплексные корни. Если $d \ge 0$, то уравнение имеет два действительных корня, которые вычисляются по формулам

$$x_1 = \frac{-b + \sqrt{d}}{2a}; \qquad x_2 = \frac{-b - \sqrt{d}}{2a}.$$

Задание 4

Пусть D — заштрихованная часть плоскости



и пусть u определяется по x и y следующим образом:

$$u = \begin{cases} \sqrt{|x^2 - 1|}, & \text{если } (x, y) \in D, \\ x + y & \text{в противном случае,} \end{cases}$$

где x, y — вещественные числа, вводимые с клавиатуры.

Написать программу, вычисления u, используя условный оператор.

Вывести результат с 5 знаками после запятой.

Задание 5

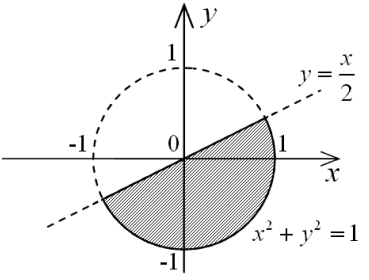
Написать программу вычисления y (используя условный оператор и оператор switch) по формуле

$$y = \begin{cases} \max(a, b), & \text{если } x = 0, \\ \min(a,b), & \text{если } x = 1, \\ |a + b|, & \text{в остальных случаях,} \end{cases}$$

где a, x, y — вещественные числа, вводимые с клавиатуры. Вывести результат с 5 знаками после запятой.

Задание 6

Пусть D — заштрихованная часть плоскости



и пусть u определяется по x и y следующим образом:

$$u = \begin{cases} -3, & \text{если } (x, y) \in D, \\ y^2, & \text{в противном случае,} \end{cases}$$

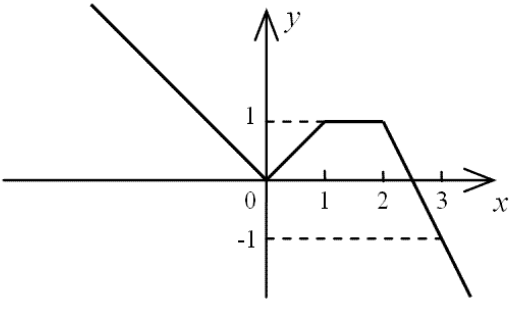
где x, y — вещественные числа, вводимые с клавиатуры.

Написать программу вычисления u, используя условный оператор.

Вывести результат с 5 знаками после запятой.

Задание 7

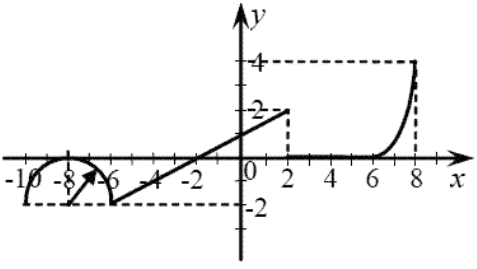
Дано действительное число a, вводимое с клавиатуры. Написать программу вычисления функции $f(x)$, график которой представлен на рисунке, используя условный оператор.



Вывести результат с 5 знаками после запятой.

Задание 8

Дано действительное число a, вводимое с клавиатуры. Написать программу, вычисления функции $f(x)$, график которой представлен на ðèñóí êå, èñï î ëüçóÿ óñëîâí ûé î ïåðàòîð ð. (Í à î òðåçêå $x \in [6, 8]$ используется квадратичная зависимость.)



Вывести результат с 5 знаками после запятой.

Лабораторная работа № 3

Организация итерационных циклических процессов

Задание 1

Написать программу вычисления с помощью цикла с параметром нижеследующих выражений, где целое число n ($1 \le n \le 100$) вводится с клавиатуры.

1. Сумма

$$\sum_{k=1}^{n}\left(2+\frac{1}{k!}\right).$$

2. Произведение

$$\prod_{k=1}^{n}\left(1+\frac{1}{k^k}\right).$$

3. Выражение, вычисляемое по формуле (n — корней):

$$\sqrt{2+\sqrt{2+\sqrt{2+...\sqrt{2}}}}\,.$$

4. Выражение, вычисляемое по формуле

$$\cfrac{1}{2^0+\cfrac{1}{2^1+\cfrac{1}{\vdots\;\;\cfrac{}{2^n+\cfrac{1}{2^{n+1}}}}}}$$

5. Используя вложенные циклы с параметром, написать программу вычисления суммы, где целое число n ($1 < n < 1000$) вводится с клавиатуры:

$$\sum_{i=0}^{n}\sum_{j=i}^{n}\frac{n+i}{2i+j+1}.$$

Задание 2

Написать программу вычисления с помощью *цикла с условием* нахождения с заданной точностью ε суммы бесконечного числового ряда. *Точность* вводится с клавиатуры.

Считать, что точность достигнута, если очередное слагаемое по модулю меньше ε. Вывести на печать значение суммы, число членов ряда, вошедших в сумму, и очередное слагаемое. Сравнить вычисленное значение суммы ряда с точным значением, указанным справа от ряда.

№ варианта	Ряд	Точное значение
1	$S=\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+\frac{1}{3\cdot 4}+\ldots$	1
2	$S=\frac{1}{1\cdot 3}+\frac{1}{3\cdot 5}+\frac{1}{5\cdot 7}+\ldots$	$\frac{1}{2}$
3	$S=1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\ldots$	ln 2

№ варианта	Ряд	Точное значение
4	$S = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$	$\frac{\pi}{4}$
5	$S = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$	e
6	$S = 1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots$	$\frac{1}{e}$
7	$S = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \ldots$	$\frac{\pi^4}{90}$
8	$S = \frac{1}{1^2} + \frac{1}{3^2} + \frac{1}{5^2} + \ldots$	$\frac{\pi^2}{8}$
9	$S = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots$	$\frac{\pi^2}{6}$
10	$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$	2
11	$S = 1 - \frac{1}{2^2} + \frac{1}{3^2} - \frac{1}{4^2} + \ldots$	$\frac{\pi^2}{12}$
12	$S = \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{2 \cdot 3 \cdot 4} + \frac{1}{3 \cdot 4 \cdot 5} + \ldots$	$\frac{1}{4}$
13	$S = \frac{1}{1^4} + \frac{1}{3^4} + \frac{1}{5^4} + \ldots$	$\frac{\pi^4}{96}$
14	$S = \frac{1}{1 \cdot 3} + \frac{1}{2 \cdot 4} + \frac{1}{3 \cdot 5} + \ldots$	$\frac{3}{4}$
15	$S = 1 - \frac{1}{2} + \frac{1}{4} - \frac{1}{8} + \ldots$	$\frac{2}{3}$

Задание 3

Написать программу табулирования функции $f(x)$ на отрезке $[x0; xk]$ с шагом $h > 0$.

Таблица функции $f(x)$ на консоли должна быть представлена в виде 3 столбцов под заголовками №, x, $f(x)$, где № — номер строки, x — значение аргумента, $f(x)$ — значение функции. Использовать форматирование данных согласно части I раздела 6.

Функцию $f(x)$ запрограммировать в виде статического метода в классе Program.

С помощью оператора switch реализовать три случая:

- использовать цикл с предусловием;
- использовать цикл с постусловием;
- использовать цикл с параметром.

Вариант 1

$$f(x)=\begin{cases} 2x\ln\left|\cos\sqrt[5]{4-x}\right|, & \text{если } x\ge 0 \\ 3{,}5\operatorname{ctg}^3 x-\dfrac{\lg x^2}{x-5}, & \text{если } x<0 \end{cases}; \quad x\in[-2;2], \quad h=0{,}25.$$

Вариант 2

$$f(x)=\begin{cases} \dfrac{\sin^4 x^2+x^2}{\sqrt{x+1}}, & \text{если } x\ge 0 \\ \dfrac{x^4+0{,}96x-1}{\cos x^3-2{,}37}, & \text{если } x<0 \end{cases}; \quad x\in[-\pi;\pi], \quad h=\frac{\pi}{6}.$$

Вариант 3

$$f(x)=\begin{cases} \dfrac{x-1}{x+\cos x^2}-e^{2-x}, & \text{если } x\le 0 \\ \sqrt{\ln\left|\operatorname{tg}^3\left(\pi-\dfrac{x}{6}\right)+x\right|+\dfrac{1}{x}}, & \text{если } x>0 \end{cases}; x\in[-2;2], h=0{,}25.$$

Вариант 4

$$f(x)=\begin{cases}3{,}56-\sqrt{x+\sqrt[3]{x+1}}, \text{ если } x\ge 0\\ \dfrac{7{,}2x\ln(\sin^2 x^{-3})}{x-1}, \text{ если } x<0\end{cases} \qquad x\in[-3;3],\ h=0{,}5.$$

Вариант 5

$$f(x)=\begin{cases}e^x\sqrt[3]{x}+1{,}7\left|x-5\right|^{1{,}4}, \text{ если } x\ge 1\\ \left|x+2{,}79\right|+\operatorname{tg}\left(\dfrac{\pi}{2}x\right), \text{ если } x<1\end{cases}; x\in[0;2], h=0{,}2.$$

Вариант 6

$$f(x)=\begin{cases}\left(\dfrac{x-2}{\cos^3\sqrt{x}}-1\right)\operatorname{arctg}x^4, \quad \text{если} \quad x\ge 0\\ \ln\left|\dfrac{\sin x}{x-1{,}5}\right|+2{,}306x^2, \quad \text{если} \quad x<0\end{cases}; x\in[-2\pi;2\pi],\ h=\frac{\pi}{3}.$$

Вариант 7

$$f(x)=\begin{cases}e^{-2\pi x\sqrt{x^2+1{,}2}}, \text{если x} \ge 0\\ 13x^4-\ln\left|\operatorname{tg}\dfrac{x}{2}\right|, \text{если } x<0\end{cases}; x\in[-2;2],\ h=0{,}25.$$

Вариант 8

$$f(x)=\begin{cases}e^{-(x^2+2{,}4)}+\ln\operatorname{tg}\dfrac{x}{6}, \text{если } x>0\\ \dfrac{\sqrt{\left|x\sqrt[3]{x+2}\right|}-3x}{x-1}, \text{если } x\le 0\end{cases}; x\in[-3;3],\ h=0{,}5.$$

Вариант 9

$$f(x)=\begin{cases}\dfrac{2,67-\ln\left|\sin x^2\right|}{\cos\sqrt{x+1}}, \text{ если } x>0\\ \operatorname{tg}\dfrac{1-\sqrt[3]{|x+1|}}{\pi-x}, \quad \text{если } x\le 0\end{cases}; \ x\in[-\pi;\pi], \ h=\frac{\pi}{6}.$$

Вариант 10

$$f(x)=\begin{cases}\dfrac{7,2\cdot 10^{1,3}\cdot\sqrt{|x-1|}}{\cos e^{\sin x}}, \text{ если } x<0\\ \sqrt[3]{x}-\operatorname{tg}\left(\dfrac{\pi}{30}-x\right), \text{если } x\ge 0\end{cases}; \ x\in[-3;2], \ h=0,5.$$

Вариант 11

$$f(x)=\begin{cases}\dfrac{\sqrt{x}\sin(x^{-2})}{x+e^{2x}}, \quad \text{если } x>0\\ \operatorname{tg}\sqrt[3]{\left|\dfrac{\pi}{3}+x\right|}, \quad \text{если } x\le 0\end{cases}; \ x\in[-2;3], \ h=0,5.$$

Вариант 12

$$f(x)=\begin{cases}\dfrac{\ln x^2}{\cos^3(x+1)}+\sqrt{2,5+x}, \quad \text{если } x<0\\ \dfrac{e^x-3,25x}{x+1}+\lg\left|tg\dfrac{x}{2}\right|, \qquad \text{если } x\ge 0\end{cases}; \ x\in[-1;1], \ h=0,2.$$

Вариант 13

$$f(x)=\begin{cases}\operatorname{tg}^3\left(\dfrac{x+1}{2}+\dfrac{\pi}{4}\right), \quad \text{если} \quad x\le 0\\ \dfrac{\operatorname{arctg}\left|x-\sqrt{x+1,5}\right|}{3x^4}, \quad \text{если } x>0\end{cases}; \ x\in[-\pi;\pi], \ h=\frac{\pi}{6}.$$

Вариант 14

$$f(x)=\begin{cases}\operatorname{arctg}\left(\dfrac{x}{\sqrt{1+x^{-2}}}+2\right), & \text{если } x>0\\ \ln\left|\dfrac{3{,}2}{5x\text{-}1}+x^3\right|-\sin(x+1), & \text{если } x\le 0\end{cases};\quad x\in[-3;2],\ h=0{,}5.$$

Вариант 15

$$f(x)=\begin{cases}x^4\left(1+\dfrac{x-2^x}{x+e^{x-1}}\right), & \text{если } x\le 0\\ \sqrt{x+\sqrt[3]{15x+\ln x}+1{,}25}, & \text{если } x>0\end{cases};\ x\in[-1;1],\ h=0{,}2.$$

2. ТИП КЛАСС В ЯЗЫКЕ C#

Поскольку программы с использованием пользовательских классов, согласно заданиям лабораторных работ части II раздела 2 разрабатываются в среде Visual Studio 2010, то необходимо создать консольный проект C# и добавить к проекту (решению) с помощью правой кнопки мыши, используя Solution Explorer (обозреватель решений) новый элемент (файл, который будет содержать программный код пользовательского класса), именуемый по умолчанию Class1.cs. Это процедура должна повторяться для каждого вводимого в проект нового пользовательского класса. Описание классов, в том числе и абстрактных — см. часть I, разделы 9–12.

Лабораторная работа № 4

Определение и тестирование классов

Создать проект Visual Studio 2010 для программируемого задания. Использовать метод Main класса Program или, если требуется, то дополнительно введенные (в частности, для большей модульности программного кода) в класс Program другие статические методы, чтобы:

- создать объекты класса, используя все имеющиеся конструкторы;
- запрограммировать ввод исходных данных с клавиатуры и вывод на экран результатов работы программы таким образом, чтобы имела место демонстрация работы всех доступных методов и свойств класса с необходимыми пояснениями;
- использовать модификатор private для полей;
- использовать свойства в конструкторе.

Задание 1

Определить класс Point, разработав следующие элементы класса.

Поля:

- int *x*, *y*.

Конструкторы, позволяющие создать экземпляр класса:

- с нулевыми координатами;
- с заданными координатами.

Методы, позволяющие:

- вывести координаты точки на экран;
- рассчитать расстояние от начала координат до точки;
- переместить точку на плоскости на вектор (*a, b*).

Свойства, позволяющие:

- получить-установить координаты точки (доступное для чтений и записи);
- умножить координаты точки на скаляр (доступное только для записи).

Задание 2

Определить класс Triangle, разработав следующие элементы класса.

Поля:

- double *a, b, c*;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- с заданными длинами сторон.

Методы, позволяющие:

- вывести длины сторон треугольника на экран;
- расчитать периметр треугольника;
- расчитать площадь треугольника.

Свойства, позволяющие:

- получить-установить длины сторон треугольника с ограничением — длины больше нуля (доступное для чтения и записи);
- установить, существует ли треугольник с данными длинами сторон (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 3

Определить класс Rectangle, разработав следующие элементы класса.

Поля:

- double *a, b*;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- с заданными длинами сторон.

Методы, позволяющие:

- вывести длины сторон прямоугольника на экран;
- расчитать периметр прямоугольника;
- расчитать площадь прямоугольника.

Свойства, позволяющие:

- получить-установить длины сторон прямоугольника с ограничением — длины больше нуля (доступное для чтения и записи);
- установить, является ли данный прямоугольник квадратом (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 4

Определить класс Money, разработав следующие элементы класса.

Поля:

- int first;//номинал купюры;
- int second; //количество купюр.

Конструктор, позволяющий:

- создать экземпляр класса с заданными значениями полей.

Методы, позволяющие:

- вывести номинал и количество купюр;
- определить, хватит ли денежных средств на покупку товара на сумму *N* рублей;
- определить, сколько шт. товара стоимости *n* рублей можно купить на имеющиеся денежные средства.

Свойства, позволяющие:

- получить-установить значение каждого поля с ограничением — номинал и количество купюр величины большие нуля (доступное для чтения и записи);
- расчитатать сумму денег (доступное только для чтения);
- установить значение поля, в соответствии с введенным значением строки с клавиатуры, а также получить значение данного поля (доступно для чтения и записи).

Лабораторная работа № 5

Определение и тестирование классов, использующих одномерные массивы

Создать проект Visual Studio 2010 для программируемого задания. Использовать метод Main класса Program или, если требуется, то дополнительно введенные (в частности, для большей модульности программного кода) в класс Program *другие* статические методы, чтобы:

- создать объекты класса, используя имеющиеся конструкторы;
- запрограммировать *ввод* исходных данных с клавиатуры и *вывод* на экран результатов работы программы таким образом, чтобы имела место демонстрация работы *всех* доступных методов и свойств класса с *необходимыми пояснениями*;

- использовать для инициализации массивов класс Random (см. часть I, раздел 8), если это оговорено в определении класса;
- использовать модификатор private для полей;
- использовать свойства в конструкторе.

Задание 1

Определить класс для работы с одномерным массивом типа double, разработав следующие элементы класса.

Поля:

- double [] dbArray;
- int n; // размерность массива;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность массива, выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа double в диапазоне [–50, 50].

Методы, позволяющие:

- вывести массив на экран;
- определить среднее арифметическое значение положительных элементов и среднее арифметическое значение отрицательных элементов;
- найти среднее геометрическое положительных элементов.

Среднее геометрическое вычисляется по формуле

$z = \sqrt[n]{a_1 a_2 ... a_n}$, где a_1, a_2, … , a_n — положительные числа.

Свойства, позволяющие:

- получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
- увеличить значение всех элементов массива на скаляр (доступное только для записи);
- получить имя объекта (доступное для чтения).

Задание 2

Определить класс для работы с одномерным массивом типа double, разработав следующие элементы класса.

Поля:

- double [] dbArray;
- int n; // размерность массива;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющийсоздать экземпляр класса:

- с заданным именем;
- устанавливающий размерность массива, выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа double в диапазоне [–70, 70].

Методы, позволяющие:

- вывести массив на экран;
- найти максимальное произведение двух соседних элементов;
- найти среднее арифметическое элементов массива и порядковый номер элемента, наиболее близкого к среднему арифметическому.

Свойства, позволяющие:

- получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
- получить количество положительных элементов массива (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 3

Определить класс для работы с одномерным массивом типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int n; // размерность массива;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность массива, выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа int в диапазоне [–50, 51].

Методы, позволяющие:

- вывести массив на экран;
- найти количество и произведение элементов массива, кратных 3;
- найти произведение положительных нечетных элементов и подсчитать их число.

Свойства, позволяющие:

- получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
- получить количество отрицательных элементов массива (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 4

Определить класс для работы с одномерным массивом типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int n; // размерность массива;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность массива, выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа int в диапазоне [–50, 51].

Методы, позволяющие:

- вывести массив на экран;

• определить количество и сумму элементов, кратных 5;
• найти сумму и произведение всех четных элементов.

Свойства, позволяющие:

• получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
• получить количество ненулевых элементов массива (доступное только для чтения).
• получить имя объекта (доступное для чтения).

Задание 5

Определить класс для работы с одномерным массивом типа double, разработав следующие элементы класса.

Поля:

• double [] dbArray;
• int n; // размерность массива;
• string name; // имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

• с заданным именем;
• устанавливающий размерность массива, выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа double в диапазоне [–100, 100].

Методы, позволяющие:

• вывести массив на экран;
• найти сумму элементов с нечетными индексами и произведение элементов с четными индексами;
• найти разность между максимальным и минимальным элементом.

Свойства, позволяющие:

• получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
• получить количество элементов массива, модуль которых меньше 10 (доступное только для чтения);
• получить имя объекта (доступное для чтения).

Задание 6

Определить класс для работы с одномерным массивом типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int n; // размерность массива;
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность массива и выделяющий память под заданное количество элементов массива и заполняющий массив случайными числами типа int в диапазоне [–80, 81].

Методы, позволяющие:

- вывести массив на экран;
- определить количество и сумму элементов, кратных 5;
- уменьшить все элементы с четными индексами на величину максимального элемента этого массива и вывести массив на экран.

Свойства, позволяющие:

- получить-установить значение размерности массива с ограничением — размерность больше нуля (доступное для чтения и записи);
- увеличить значение всех четных элементов массива на скаляр (доступное только для записи);
- получить имя объекта (доступное для чтения).

Лабораторная работа № 6

Определение и тестирование классов, использующих двумерные массивы (матрицы)

Создать проект Visual Studio 2010 для программируемого задания. Использовать метод Main класса Program или, если требуется, то дополнительно введенные (в частности, для большей модульно-

сти программного кода) в класс Program другие статические методы, чтобы:

- создать объекты класса, используя имеющиеся конструкторы;
- запрограммировать *ввод* исходных данных с клавиатуры и *вывод* на экран результатов работы программы таким образом, чтобы имела место демонстрация работы *всех* доступных методов и свойств класса с *необходимыми пояснениями*;
- использовать для инициализации массивов класс Random (см. часть I, раздел 8), если это оговорено в определении класса;
- использовать модификатор private для полей;
- использовать свойства в конструкторе.

Задание 1

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность матрицы, выделяющий память под заданное количество элементов матрицы и заполняющий матрицу случайными числами типа int в диапазоне [–50, 51].

Методы, позволяющие:

- вывести матрицу на экран;
- вычислить произведение и число элементов матрицы, находящихся под главной диагональю;
- вычислить сумму и число элементов матрицы, находящихся над главной диагональю.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении размерности больше нуля (доступны для чтения и записи);

- увеличить значение всех элементов матрицы на скаляр (доступное только для записи);
- получить имя объекта (доступное для чтения).

Задание 2

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр:

- с заданным именем;
- устанавливающий размерность матрицы, выделяющий память под заданное количество элементов матрицы и заполняющий матрицу случайными числами типа int в диапазоне [–70, 71].

Методы, позволяющие:

- вывести матрицу на экран;
- вычислить произведение и число положительных элементов матрицы, находящихся под главной диагональю;
- вычислить сумму и число отрицательных элементов матрицы, находящихся над главной диагональю.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении размерности больше нуля (доступны для чтения и записи);
- получить количество положительных элементов массива (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 3

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность матрицы (m×n) и выделяющий память под заданное количество элементов матрицы и заполняющий матрицу случайными числами типа int в диапазоне [–40, 41].

Методы, позволяющие:

- вывести матрицу на экран;
- вычислить количество строк, содержащих хотя бы один нулевой элемент;
- вычислить номер строки и номер столбца, на пересечении которых находится минимальный по модулю элемент матрицы.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении размерности больше нуля (доступны для чтения и записи);
- получить количество отрицательных элементов массива (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 4

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность матрицы (m×n) и выделяющий память под заданное количество элементов матрицы и заполняющий матрицу случайными числами типа int в диапазоне [–60, 61].

Методы, позволяющие:

- вывести матрицу на экран;
- вычислить номер строки с наибольшей суммой элементов;
- вычислить сумму наибольших значений элементов строк.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении размерности больше нуля (доступны для чтения и записи);
- получить количество ненулевых элементов массива (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 5

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность матрицы (m×n) и выделяющий память под заданное количество элементов матрицы и заполняющий матрицу случайными числами типа int в диапазоне [–50, 51].

Методы, позволяющие:

- вывести матрицу на экран;

- вычислить сумму элементов столбца, в котором расположен минимальный элемент матрицы;
- вычислить сумму наименьших значений элементов ее столбцов.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении — размерности больше нуля (доступны для чтения и записи);
- для получения количества элементов массива, модуль которых меньше 10; (доступное только для чтения);
- получить имя объекта (доступное для чтения).

Задание 6

Определить класс для работы с двумерным массивом (матрицей) типа int, разработав следующие элементы класса.

Поля:

- int[] intArray;
- int m, n; //размерность матрицы (m×n);
- string name; //имя объекта.

Конструктор, позволяющий создать экземпляр класса по умолчанию.

Конструктор, позволяющий создать экземпляр класса:

- с заданным именем;
- устанавливающий размерность матрицы (m×n) и выделяющий память под заданное количество элементов матрицы и заполняющий массив случайными числами типа int в диапазоне [–80, 81].

Методы, позволяющие:

- вывести матрицу на экран;
- найти в каждой строке матрицы наибольший элемент, поменять его местами с элементом главной диагонали, а на его место записать минимальный элемент этой строки, и вывести матрицу на экран;
- вычислить номер столбца с наименьшей суммой элементов.

Свойства, позволяющие:

- получить-установить значения размерности матрицы при ограничении размерности больше нуля (доступны для чтения и записи);
- увеличить значение всех четных элементов матрицы на скаляр (доступное только для записи);
- получить имя объекта (доступное для чтения).

Лабораторная работа № 7

Абстрактные классы, наследование и полиморфизм

Создать проект Visual Studio 2010 для программируемого задания. Использовать метод Main класса Program или, если требуется, то дополнительно введенные (в частности, для большей модульности программного кода) в класс Program другие статические методы, чтобы:

- запрограммировать *ввод* исходных данных с клавиатуры и *вывод* на экран результатов работы программы таким образом, чтобы имела место демонстрация работы *всех* доступных методов и свойств классов — наследников согласно заданиям;
- запрограммировать *вывод* на экран имен объектов классов — наследников, содержащихся в массиве класса Picture.

При создании классов — наследников и класса Picture:

- использовать модификатор private для полей;
- использовать свойства в конструкторе.

Задание 1

Определить абстрактный класс Shape с абстрактными свойствами и методами (см. часть I, раздел 12):

- абстрактные свойства класса Shape:

параметр (один), определяющий фигуру (доступное для чтения и записи);

подсчет числа вершин фигуры (доступное только для чтения);

позволяющее получить имя объекта (доступное для чтения);

• абстрактные методы класса Shape:

вычисления площади и периметра фигуры;

рисования закрашенной фигуры (для консоли — соответствующее текстовое сообщение).

Определить классы-наследники (в качестве примера подобных классов см. лабораторную работу № 5, задания 2, 3):

• Square (квадрат — задается длиной стороны);
• Circle (окружность — задается радиусом);
• Triangle (равносторонний треугольник — задается длиной стороны).

В классах Square, Circle и Triangle должны быть реализованы абстрактные свойства и методы класса Shape.

Определить класс Picture, содержащий массив объектов этих классов (см. ниже).

Класс Picture

Поля:

• Shape [] shpArray;
• int n; // размерность массива.

Конструктор, позволяющий создать экземпляр класса:

• устанавливающий размерность массива и выделяющий память под заданное количество элементов массива.

Методы, позволяющие:

• заполнить массив объектами классов наследников;
• вывести на экран имена объектов, заполняющих массив.

Свойства, позволяющие:

• получить-установить значение размерности массива с ограничением размерности больше нуля (доступное для чтения и записи).

Задание 2

Определить абстрактный класс Shape с абстрактными свойствами и методами (см. часть I, раздел 12):

• абстрактные свойства класса Shape:

параметры (два), определяющих фигуру (доступное для чтения и записи);

подсчет числа вершин фигуры (доступное только для чтения);

позволяющее получить имя объекта (доступное для чтения);

• абстрактные методы класса Shape:

вычисления площади и периметра фигуры;

рисование закрашенной фигуры (для консоли — соответствующее текстовое сообщение.)

Определить классы — наследники (в качестве примера подобных классов см. лабораторную работу № 5, задания 2, 3):

• Pentagon (правильный пятиугольник — задается длиной стороны и числом сторон);
• Ellipse (эллипс — задается полуосями);
• Triangle (прямоугольный треугольник — задается длиной катетов).

В классах Pentagon, Ellipse и Triangle должны быть реализованы абстрактные свойства и методы класса Shape.

Определить класс Picture, содержащий массив объектов этих классов (см. задание 1).

Задание 3

Определить абстрактный класс Triangle с абстрактными свойствами и методами (см. часть I, раздел 12):

• абстрактные свойства класса Triangle:

для длин двух сторон и угла между ними (доступное для чтения и записи);

позволяющее получить имя объекта (доступное для чтения);

• абстрактные методы класса Triangle:

вычисления площади и периметра фигуры;

рисование закрашенного треугольника (для консоли — соответствующее текстовое сообщение.)

Определить классы — наследники (в качестве примера подобных классов см. лабораторную работу № 5, задание 2):

• EqTriangle (равносторонний треугольник);
• IsTriangle (равнобедренный треугольник);
• RectTriangle (прямоугольный треугольник).

В классах EqTriangle, IsTriangle и RectTriangle должны быть реализованы абстрактные свойства и методы класса Triangle.

Определить класс Picture, содержащий массив объектов этих классов (см. задание 1).

Библиографический список

Нейгел Кристиан. C# 4.0 и платформа .NET 4 для профессионалов : пер. с англ. / *Нейгел Кристиан, Ивьен Билл, Глинн Джей, Уотсон Карли.* — М. : ООО «И.Д. Вильямс», 2011. — 1440 с. : ил.

Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд. / Дж. Рихтер. — СПб. : Питер, 2013. — 896 с.: ил. — (Серия «Мастер-класс»).

Троелсен Эндрю. Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. : пер. с англ. / Эндрю Троелсен. — М. : ООО «И.Д. Вильямс», 2011. — 1392 с. : ил.

Учебное издание

РАЗРАБОТКА АЛГОРИТМОВ С ИСПОЛЬЗОВАНИЕМ ПРИНЦИПОВ ООП НА ЯЗЫКЕ C#

Учебно-методическое пособие

Калинин Александр Александрович

Редактор *Е.Б. Казакова*
Компьютерная верстка *И.В. Бурлаковой*

Подписано в печать 25.02.14.
Формат 60×84/16. Бумага офсетная. Гарнитура «Times New Roman».
Печать на ризографе. Усл. печ. л. 6,16. Тираж 50 экз. Заказ № 32.

Московский государственный университет печати имени Ивана Федорова.
127550, Москва, ул. Прянишникова, 2А.
Отпечатано в Издательстве МГУП имени Ивана Федорова.