Министерство образования Республики Беларусь
УО «Полесский государственный университет»

Л. П. ВОЛОДЬКО, В. Л. НИКОЛАЕНКО, Д. В. НИКОЛАЕНКО

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие
для студентов технических специальностей различных форм обучения

Пинск
ПолесГУ
2020

УДК 004.42(075.8)
ББК 32.973я73
В68

Р е ц е н з е н т ы:
кандидат технических наук Б. А. Железко;
кандидат экономических наук О. А. Синявская

У т в е р ж д е н о
научно-методическим советом ПолесГУ

Володько, Л. П.
В68 Объектно-ориентированное программирование : учебное пособие / Л. П. Володько, В. Л. Николаенко, Д. В. Николаенко. – Пинск : ПолесГУ, 2020. – 164 с.

ISBN 978-985-516-649-9

Учебное пособие содержит материалы по основным принципам объектно-ориентированного программирования, его преимуществам и недостаткам, а также основным конструкциям языка C#: классы, объекты, конструкторы, модификаторы доступа, деструкторы и сборка мусора, структуры, перечисления, абстрактные классы, интерфейсы, массивы и коллекции, пользовательские коллекции, делегаты и события.

Издание предназначено для студентов технических специальностей различных форм обучения, изучающих программирование.

УДК 004.42(075.8)
ББК 32.973я73

ISBN 978-985-516-649-9

ОГЛАВЛЕНИЕ

ГЛАВА 1 ВВЕДЕНИЕ В ООП 6
1.1 Достоинства ООП 6
1.2 Недостатки ООП 7
1.3 Будущее ООП 9
1.4 Три кита ООП 9
Заключение 11
Вопросы для самоконтроля 12

ГЛАВА 2 КЛАССЫ И ОБЪЕКТЫ 13
2.1 Классы 13
2.2 Объявление классов 15
2.3 Экземпляры объектов 16
2.4 Статические классы 18
2.5 Статические члены класса 19
Вопросы для самоконтроля 20

ГЛАВА 3 КОНСТРУКТОРЫ, СБОРКА МУСОРА И ДЕСТРУКТОРЫ 21
Вопросы для самоконтроля 24

ГЛАВА 4 ДЕСТРУКТОРЫ 25
Вопросы для самоконтроля 27

ГЛАВА 5 КЛЮЧЕВОЕ СЛОВО «THIS» 28
5.1 Constructor chaining 28
5.2 Последовательность срабатывания конструкторов 29
Вопросы для самоконтроля 30

ГЛАВА 6 МОДИФИКАТОРЫ ДОСТУПА 31
Вопросы для самоконтроля 32

ГЛАВА 7 ПОЛЯ КЛАССА 33
Вопросы для самоконтроля 33

ГЛАВА 8 КОНСТАНТЫ 34
Вопросы для самоконтроля 34

ГЛАВА 9 СВОЙСТВА 35
9.1 Использование свойств 37
9.2 Автоматически реализуемые свойства 37
9.3 Свойства в интерфейсах 39
Вопросы для самоконтроля 39

ГЛАВА 10 НАСЛЕДОВАНИЕ: ЗАПЕЧАТАННЫЕ КЛАССЫ, ЧЛЕНЫ КЛАССОВ 40
Вопросы для самоконтроля 45

ГЛАВА 11 ПОЛИМОРФИЗМ 46
11.1 Операции с объектами 49

11.2 Наследование ... 51
11.3 Перегрузка методов ... 51
11.4 Роль ключевого слова «params» в полиморфизме ... 52
11.5 Приоритет языка в выборе методов ... 57
11.6 Ключевые слова «New» и «Override» в C# ... 58
11.6.1 Эксперимент ... 60
11.6.2 Эксперимент с тремя классами ... 62
11.6.3 Ключевое слово «base» ... 65
Заключение ... 67
Вопросы для самоконтроля ... 68

ГЛАВА 12 СТРУКТУРЫ. ... 69
Вопросы для самоконтроля ... 70

ГЛАВА 13 ПЕРЕЧИСЛЕНИЯ ... 71
Вопросы для самоконтроля ... 71

ГЛАВА 14 АБСТРАКТНЫЕ КЛАССЫ ... 72
14.1 Абстрактные классы в действии ... 72
14.2 Описание методов в абстрактном классе ... 72
14.3 Декларация методов в абстрактном классе ... 74
14.4 Реализация абстрактного метода в производном классе ... 75
14.5 Инициализация полей в абстрактных классах ... 77
14.6 Абстрактные методы в неабстрактных классах ... 78
14.7 Вызов абстрактного метода родителя ... 78
14.8 Абстрактный класс, который наследуется от другого абстрактного класса ... 79
14.9 Может ли абстрактный класс быть sealed ... 80
Заключение ... 80
Вопросы для самоконтроля ... 81

ГЛАВА 15 ИНТЕРФЕЙСЫ ... 82
15.1 Интерфейсные ссылки ... 87
15.2 Ключевое слово «as» ... 89
15.3 Ключевое слово «is» ... 89
15.4 Свойства в интерфейсах ... 89
15.5 Индексаторы в интерфейсах ... 90
15.6 Наследование интерфейсов ... 91
15.7 Явная реализация интерфейса ... 92
Вопросы для самоконтроля ... 94

ГЛАВА 16 МАССИВЫ И КОЛЛЕКЦИИ В C# ... 95
16.1 Одномерные массивы ... 95
16.2 Двумерные массивы ... 95
16.3 Массивы массивов ... 96
16.4 Массивы объектов ... 96
Вопросы для самоконтроля ... 97

ГЛАВА 17 КОЛЛЕКЦИИ ... 98
17.1 Коллекция List<T> ... 99
17.2 Коллекция Dictionary<T, V> ... 104

17.3 Коллекция Queue<T> 108
17.4 Сортированный список SortedList<TKey, TValue> 110
Вопросы для самоконтроля 114

ГЛАВА 18 ПОЛЬЗОВАТЕЛЬСКИЕ КОЛЛЕКЦИИ В C# 115
Вопросы для самоконтроля 121

ГЛАВА 19 ДЕЛЕГАТЫ И СОБЫТИЯ 122
Вопросы для самоконтроля 127

ГЛАВА 20 СОБЫТИЯ 128
20.1 Field-like события 129
20.2 Событийная модель 129
Вопросы для самоконтроля 133

ГЛАВА 21 СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ 134
Вопросы для самоконтроля 141

ГЛАВА 22 ДИАГРАММЫ UML 142
22.1 Диаграмма использования 142
22.2 Диаграмма классов 143
22.3 Диаграмма состояний 144
22.4 Диаграмма деятельности (активности) 145
22.5 Диаграмма последовательности 146
22.6 Диаграмма коммуникации 147
22.7 Диаграмма компонентов 148
22.8 Диаграмма размещения 148
Вопросы для самоконтроля 148

ГЛАВА 23 РЕФЛЕКСИЯ В C# 149
Вопросы для самоконтроля 151

ГЛАВА 24 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 152
24.1 Порождающие паттерны 152
24.2 Структурирующие паттерны 155
24.3 Паттерны поведения 157

ТЕРМИНЫ 160

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 162

ГЛАВА 1
ВВЕДЕНИЕ В ООП

1.1 Достоинства ООП

От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.

Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.

Данные и операции вместе образуют определенную сущность, и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.

Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.

Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП, именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа (type extension) и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях:

1. Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.
2. Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это может привести к изменению алгоритма, в котором используется данный объект.
3. Реализация родовых компонент. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более, чем с одним видом объектов.
4. Доведение полуфабрикатов. Нет надобности подстраивать компоненты под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять по мере необходимости до различных законченных продуктов.
5. Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Многоразового использования программного обеспечения на практике добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.

Сокращается время на разработку, которое с выгодой может быть отдано другим проектам.

Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.

Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество

работающих с ней программ. Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

1.2 Недостатки ООП

Объектно-ориентированное программирование требует знания четырех вещей:

1. Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменения мировоззрения и может отнять на изучение значительное количество времени.

2. Многоразовое использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотеки классов. А это тоже требует времени.

3. Проектирование классов – задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.

4. Очень трудно изучать классы, не имея возможности их «пощупать». Только с приобретением мало-мальского опыта можно уверенно себя почувствовать при работе с использованием ООП.

Итак, усилия на освоение базовых концепций невелики, но вот в случае библиотек классов и их использования они могут быть очень существенными.

Поскольку детали реализации классов обычно неизвестны, то программисту, если он хочет разобраться в том или ином классе, нужно опираться на документацию и на используемые имена. И время, которое было сэкономлено на том, что удалось обойтись без написания собственного класса, должно быть отчасти потрачено (особенно вначале освоения) на то, чтобы разобраться в существующем классе.

Документирование классов – задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда «размазан» по многим маленьким методам.

Абстракция данных ограничивает гибкость клиентов. Клиенты могут лишь выполнять те операции, которые предоставляет им тот или иной класс. Они уже лишены неограниченного доступа к данным. Причины здесь аналогичны тем, что вызвали к жизни использование высокоуровневых языков программирования, чтобы избежать непонятных программных структур. Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Часто можно слышать, что ООП является неэффективным. Как же дело обстоит в действительности? Мы должны четко проводить грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

1. *Неэффективность на этапе выполнения.* В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных С-программ [1].

В гибридных языках типа Oberon-2, Object Pascal и C++ посылка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако существует другой фактор (который затрагивает время выполнения) – это абстракция данных. Абстракция запрещает непосредственный доступ к полям класса и требует, чтобы каждая операция над данными выполнялась через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Однако когда абстракция используется только там, где она необходима (т. е. не из одной лишь прихоти), то замедление вполне приемлемое.

2. *Неэффективность в смысле распределения памяти*. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

3. *Излишняя универсальность*. Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода.

Одно из возможных решений – строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность.

Другой подход – дать возможность компоновщику удалять лишние методы. Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем.

Oberon избрал третий путь избавления от излишней универсальности. Программные части могут добавляться на этапе выполнения. Таким образом, нет надобности загружать всю программу целиком, а можно обойтись лишь теми ее частями, которые в данный момент необходимы. Как показала практика, это экономит гораздо больше кода, чем можно добиться при удалении лишних методов. Следовательно, нельзя утверждать, что ООП во-

все не эффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности и на этапе выполнения, и в смысле памяти сводится практически на нет.

1.3 Будущее ООП

Выживет ли объектно-ориентированное программирование, или оно лишь модное поветрие, которое скоро исчезнет?

Классы нашли свое место в большинстве современных языков программирования. Одно лишь это говорит о том, что им суждено остаться. Классы в самом ближайшем будущем войдут в стандартный набор концепций для каждого программиста точно так же, как многие сегодня применяют динамические структуры данных и рекурсию, которые двадцать лет назад были также в диковинку. В то же время классы – это просто еще одна новая конструкция наряду с остальными. Нам нужно узнать, для каких ситуаций они подходят, и только здесь мы и будем их использовать. Правильно выбрать инструмент для конкретной задачи – обязательно для каждого мастерового и в еще большей степени для каждого инженера.

ООП ввергает многих в состояние эйфории. Пестрящая тут и там реклама сулит нам невероятные вещи, и даже некоторые исследователи, похоже, склонны рассматривать ООП как панацею, способную решить все проблемы разработки программного обеспечения. Со временем эта эйфория постепенно уляжется. И после периода разочарования люди, быть может, перестанут уже говорить об ООП, точно так же как сегодня вряд ли от кого можно услышать о структурном программировании. Но классы будут использовать как нечто само собой разумеющееся, и мы сможем, наконец, понять, что они собой представляют: просто компоненты, которые помогают строить модульное и расширяемое программное обеспечение.

1.4 Три кита ООП

На заре компьютерной эры, когда польза от их использования обусловливалась прежде всего промышленными интересами, вдруг назрела потребность в «очеловечивании» и в то же время укреплении языка, с помощью которого человеку суждено было общаться с машиной. Языков программирования уже тогда было более чем предостаточно, однако такое же несметное количество не годилось для решения критических задач, в которых не было места грубым ошибкам и недоработкам. В ту пору уже доминировал Си. Вскоре на смену ему пришел Си++ – более изощренный и мощный, нежели первичное детище Карнигана и Ритчи. Наряду со многими нововведениями Си программисту в руки было вложено новое средство – Объект. Это резко упростило некоторые вещи. В тот период все перевернулось с ног на голову: программист, оперирующий аморфными понятиями, теоретическими «субъектами» и прочими персонажами из сказки о clsMyObj, решал те же задачи, что и программист, не решившийся бросить процедурный (последовательно-алгоритмический – устаревш. информ.) язык, выполнял свою работу качественней, быстрей, несомненно, с большим удовольствием и с меньшими усилиями. Один из попутных «козырей» ООП – использование готового кода.

Все так, однако бытует легенда, будто Объекты зародились еще задолго до появления первого языка программирования. Признаем тот факт, что постепенно все (даже самые молодые и «вторичные», детские (Basic) и студенческие (Pascal)) языки программирования стали обзаводиться своей объектной стратегией. И хотя некоторые только имитировали ООП, тенденция все же дошла до наших дней. До сих пор на пике популярности книги по объектно-ориентированному программированию, причем не применительно к какому-либо конкретному языку, а освещающие принципы, идеологию и образ «объект-

ного мышления». Сами ОО-специалисты называют это «ОО-философией». По своей природе и идее ООП не привязывается к конкретному языку – это принцип построения программы, ее структурная схема. Это поистине образ мышления!

ООП подразумевает в первую очередь наличие классов.

Классы – это как будто самый мелкий элемент той структуры, которая и формирует объектно-ориентированную концепцию в целом. Каждый класс имеет свои свойства, свои методы, свои события. Непременным свойством истинного ОО-языка является инкапсуляция, что означает «закупоренность» механизма того или иного явления в Объектах. Вообще, инкапсуляция основана на области видимости (Scope) внутренних переменных Класса. Таким образом, программист зачастую использует объекты, созданные другими программистами, и абсолютно не задумывается, как все это устроено, он просто доверяет «производителю объекта» и всецело занят лишь решением задачи, поставленной конкретно ему. Яркий пример такого подхода к созданию приложений при помощи инкапсулированных механизмов – элементы управления (ActiveX-компоненты), которые, кстати, можно использовать во всех поддерживающих ActiveX-технологию языках программирования. Одним словом, вы добавляете в проект текстовое поле, не задумываясь о том, как реализована, например, прорисовка его текста.

Говорят, ООП держится на трех «китах». В таком случае первым является инкапсуляция, поскольку оказалась бы нецелесообразной ОО-система вообще, не будь инкапсулированных методов, свойств, спрятанных за ненадобностью имитаций событий. Однако все это отнюдь не значит, что программист не в силах вникнуть в систему того или иного объекта. Если речь идет не о готовых коммерческих компонентах, скомпилированных и защищенных от декомпиляции, а об Объектах и Классах, созданных или включенных в проект в качестве равноправного кода, программист имеет к нему такой же доступ, как и к любому другому модулю.

Второй «кит» – наследование. Что это значит? Это значит, что Объекты (Классы, Коллекции) могут перенимать некоторые свойства у своих прародителей. Как? Это зависит от того языка, на котором пишется программа. Однако в любом случае картина та же: это приводит к повторному использованию уже написанного однажды кода. Наследование по определению заставляет что-то у кого-то наследовать, значит, можно создать свое текстовое поле на основе уже существующего класса TextBox (в Visual Basic), причем новый Класс (назовем его EnhancedBox) наделен всем тем, чем располагает его стандартный родитель, плюс новыми свойствами, определяемыми его создателем. Никуда не денутся свойства Font, Alignment, Multiline, если их специально не «ампутировать». На основе наследования, пусть даже искусственного, в Visual Basic выросла и техника Субклассинга (Subclassing), при которой компоненты наделяются новыми свойствами. Чаще термин употребляется применительно к элементам управления.

Третий «кит» ООП – полиморфизм. Вообще, это уже из области искусственного интеллекта. В данном случае речь идет о той роскоши, за которую стоит выдерживать и немногие издержки ООП (однако же выигрыш очевиден и бесспорен!). Объекты, располагающие одноименными методами или свойствами, могут с легкостью управляться в ходе программы независимо от того, что эти одноименные свойства и методы выполняют абсолютно разные действия в отношении абсолютно разных классов и устроены по-разному. Например, возьмем свойство Font, широко распространенное во многих компонентах:

```
Private Sub Command1_Click()
   Dim Ctl As Control
   For Each Ctl In Me.Controls
       Ctl.FontName = "Courier"
   Next
End Sub
```

Таким образом, мы избежали тщательного учета каждого элемента управления на нашей форме и упомянули сразу всех при помощи системы For Each:In : Next (Для Каждого: В_Коллекции: Стоп).

Конечно, ничто не мешает поступить по-иному. Это проще, но при условии, что на форму динамически не добавляются компоненты.

Впрочем, указанная конструкция – явный атрибут ООП, поскольку For Each работает лишь с коллекциями.

> Коллекции – это своеобразные классы, являющиеся учетными наборами других классов.

Чем удобны коллекции? Прежде всего, методами Add, Remove, свойствами Item. Используя наряду с другими стандартными для классов методами Add и Remove, программист имеет возможность пополнять коллекцию, удалять из нее определенные с помощью Item(Index) элементы. Любая коллекция имеет метод Clear.

Допустим, мы создали собственный интернет-загрузчик. Это повлечет за собой работу со списками гиперссылок, их хранением и управлением.

Для того чтобы программа оказалась эффективной и более-менее компактной, необходимо избавить ее от излишней конкретики там, где это возможно. Например, гиперссылка может вести как к графическому ресурсу, так и к обычной гипертекстовой странице. А может это запрос к удаленной базе данных? Поэтому целесообразно было бы создать одноименные методы для обоих случаев (DownLoad, Save и т. д.). Тогда великолепен код:

```
For Each cLink In cLinks
     cLink.DownLoad
     cLink.Save
Next
cLinks.Clear
```

Теперь можно забыть создание Классов cLink и cLinks как кошмарный сон. Впереди лишь удовольствие от работы с «запечатанными» методами / свойствами / событиями.Это и есть полиморфизм в примитиве. Почитатели ООП уже не мыслят себе процесса написания программы без Классов, свойств, методов и событий. Бытует также понятие «Объектно-ориентированный дизайн» (ООД).

Заключение

Определенный вклад в развитие ООП внес, конечно же, каждый из существующих и успешно применяемых сегодня языков; каждый по-своему популяризовал ОО-схему проектирования программного обеспечения, показал, насколько упрощаются некоторые задачи, а иногда – просто становятся выполнимыми. Однако техника ОО до сих пор на стадии изучения. Существуют целые институты, исследующие концепцию в целом, существуют и отдельные деятели движения. Кроме того, идея ООП и ООД порождает новые идеи и течения.

Вопросы для самоконтроля

1. Достоинства и недостатки объектно-ориентированного подхода.
2. Будущее объектно-ориентированного программирования.
3. В чем заключается принцип объектно-ориентированного подхода?
4. Назовите «три кита» объектно-ориентированного программирования.

ГЛАВА 2
КЛАССЫ И ОБЪЕКТЫ

2.1 Классы

> Классом называют логическую структуру, позволяющую создавать свои собственные пользовательские типы путем группирования переменных других типов, методов и событий.

Часто класс сравнивают с чертежом. Подобно чертежу в классе описывается конструкция объекта, его функционал, свойства. Объектно-ориентированный подход подразумевает то, что от классов можно создать любое количество объектов, если это не статические классы. Такой подход позволяет упростить создание программного обеспечения из готовых частей (классов).

Даже тот, кто специально не знакомился с объектно-ориентированным программированием, но пытался пробовать себя в программировании и создавал, например, простейшие оконные приложения, уже использовал понятия классы и объекты, когда добавлял на форму элементы управления (кнопки, метки и т. д.).

Объектно-ориентированный подход основан на том, что объективная реальность описана при помощи объектов (классов). Возьмем абстрактный пример. Есть корпус шариковой ручки и есть стержень. Согласно концепции объектно-ориентированного программирования корпус ручки и стержень – это объекты, которые взаимодействуют друг с другом, и в конечном итоге получается объект, умеющий выполнять определенные функции. Так, корпус ручки имеет свои свойства и методы (к свойствам можно отнести цвет, размер, материал и т. д.; к методам – возможность выдвигать / задвигать стержень, прятать стержень в колпачке и т. д.). Аналогично и стержень имеет свои свойства: объем пасты, цвет пасты, толщину шарика и прочее. По отдельности эти объекты можно назвать неработоспособными (конечно, можно писать стержнем без корпуса или выцарапывать корпусом ручки без стержня, но имеется ввиду нормальное функционирование).

Использование этих двух объектов вместе позволяет получить новый инструмент – шариковую ручку. Получается, для создания своего объекта были использованы готовые объекты, созданные когда-то и кем-то (они были придуманы и начерчены на бумаге (чертежи), а потом переданы в производство, и по этим чертежам были изготовлены объекты, соединены вместе и предоставлены нам для работы). Т. е. классом в этом случае выступает чертеж изделия, а объектом – готовая продукция. В программировании аналогично.

Программный код, написанный программистом, – это класс. В классе программист может предусмотреть поля класса (до эпохи ООП назывались переменные), методы (раньше их называли функциями и процедурами), свойства, события, делегаты и т. д.

Запуск программы на выполнение означает, что она попадает в ОЗУ, оттуда процессор ее считывает и выполняет команды. Выполняемая программа, размещенная в ОЗУ, – это и есть экземпляр объекта. Таких программ можно запустить одновременно несколько, т. е. от одного класса (чертежа) создать множество экземпляров.

Такой подход очень удобен и практичен при повторном использовании кода, позволяющем сократить время на написание ПО. Так, разработчики FrameWork предусмотрели и разработали класс «Button» (кнопка), наделили его всевозможными полями, свойствами, методами событиями, которые присущи всем кнопкам, а все остальные разработчики просто используют этот класс в своих программах. Для этого достаточно добавить новый экземпляр класса «Button», наделить его определенными свойствами, запрограммировать необходимые методы, и кнопка будет работать аналогично своим копиям.

В фантастических фильмах о будущем и о клонировании людей тоже можно усмотреть пример использования объектно-ориентированного подхода. Например, в фильме

«Шестой день» создавались копии людей следующим образом: была некая «заготовка» человека без пола, роста и прочих отличительных признаков. Это был некий каркас человека с заложенными в него методами (руки и ноги были и могли двигаться), свойствами (цвет волос, рост, вес, пол и т. д.), но все они были неопределенны (не заданы). Для того чтобы создать клон, необходимо было проинициализировать специальным конструктором эту «заготовку», т. е. задать поля (ДНК, хранящие все свойства человека), методы (умения говорить, ходить и выполнять определенный функционал) и т. д. И это очень напоминает создание экземпляров от классов. Ведь кнопка, пока ей не задать свойства (размер, положение, надпись и т. д.), методы, пока им не написать операторы, которые будут выполняться по нажатию на эту кнопку (наведение мышью, получение фокуса и прочее), – просто бесполезный прямоугольник на форме приложения.

Если взглянуть на рабочий стол ОС Windows, то тоже можно рассмотреть объекты со своими методами и свойствами. Рабочий стол – объект, обладающий свойствами (размер в пикселях, картинка и т. д.). На объекте «Рабочий стол» можно вызвать контекстное меню и посмотреть основные его свойства и методы. Этот объект содержит панель задач со своими свойствами и методами, ярлыки и т. д.

Итак, отличие между классом и объектом установлено, класс – это чертеж, а объект – выполняемая программа. Но часто говорят: «Экземпляр объекта». Рассмотрим разницу в понятиях «Объект» и «Экземпляр».

> Объект – это конкретная сущность, основанная на классе и иногда называемая экземпляром класса.

В любом объекте можно выделить статическую и динамическую составляющие. Если рассматривать объект «стержень шариковой ручки», то динамическим в нем окажется цвет пасты, а все остальные его члены – статическими (неизменными), так и у объекта «клон человека» есть изменяемые поля (цвет волос, рост, вес и т. д.), но есть и статические (умение ходить, говорить, писать и прочее). Аналогично и в программировании. Класс «Button» имеет изменяемые от объекта к объекту поля – надпись, размер и т. д. Представим ситуацию, что необходимо в приложении разместить сто таких копий. Каждая из них займет определенный объем ОЗУ (в куче) при выполнении программы, при этом в памяти будут храниться все статические члены класса, которые будут одинаковы для каждого экземпляра. Исходя из этих соображений решено было разделять классы на статические и динамические составляющие и хранить в памяти одну копию статической части (объект) и необходимое количество динамических частей (экземпляров).

На рисунке 2.1 схематически изображены класс, объект и экземпляры.

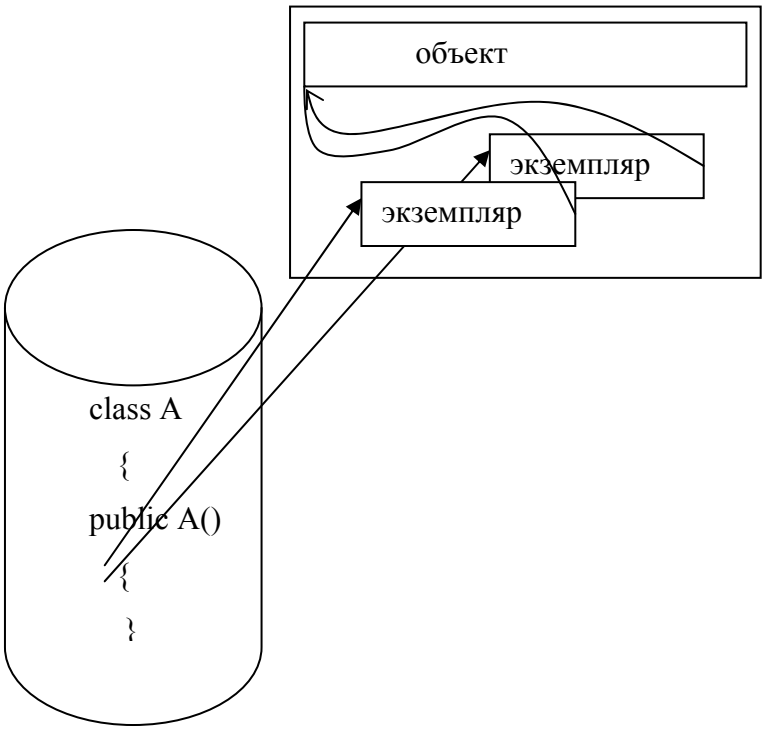


Рисунок 2.1 – Схематическое изображение класса, объектов и экземпляров

Как видно из рисунка, в программном коде происходит создание двух экземпляров класса A. Конструктор класса вернет ссылку на экземпляр объекта, хранящий динамические члены класса, а экземпляр, в свою очередь, хранит ссылку на сам объект – статическую часть класса. Таким образом происходит экономия ресурсов памяти.

Для нестатических классов клиентский код может его использовать, создав экземпляры этого класса. Если класс статический, то в памяти остается и хранится только одна его копия, и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра объекта.

2.2 Объявление классов

Классы объявляются с помощью ключевого слова «class», общий вид класса представлен ниже:

```
[область видимости] class ИмяКласса : [Родительский класс, интерфейсы]
{
    // Поля
    // Константы
    // Конструкторы
    // Свойства
    // Методы
    // Методы завершения
    // Индексаторы
    // Операторы
    // События
    // Делегаты
    // Классы
    // Интерфейсы
    // Структуры
    // Перечисления
    // Деструкторы
}
```

C# не допускает множественное наследование классов, но разрешает множественную реализацию интерфейсов.

Для классов можно задать уровень доступа. Уровни доступа будут рассмотрены в следующих разделах.

> Члены класса: поля, свойства, конструкторы, методы, события, делегаты, деструкторы.

2.3 Экземпляры объектов

Экземпляры объектов создаются при помощи ключевого слова «new». Создание экземпляра схоже с объявлением переменной. Рассмотрим пример создания экземпляра и сравним с объявлением переменной. Общий вид создания экземпляра класса имеет вид:

```
ИмяКласса ИмяЭкземпляра = new ИмяКонструктора([параметры]);
```

Частный случай: создадим экземпляр oA от класса A, вызвав конструктор по умолчанию (без параметров):

```
A oA = new A();
```

В этом примере будет создана переменная oA, которая будет хранить ссылку на экземпляр класса A, поля класса будут проинициализированы конструктором по умолчанию (без параметров). Задача конструктора состоит в выделении памяти для объекта, возвращении ссылки на этот объект и инициализации полей класса начальными значениями. Подробнее конструкторы рассмотрим позже. Вспомним синтаксис объявления переменных:

```
string str = «строка»;
```

В этом случае компилятор выделит в памяти место для хранения значения строкового типа, поименует его (в нашем примере str) и проинициализирует начальным значением «строка». В результате в str будет храниться ссылка на объект типа «string».

Объявление переменных и экземпляров очень схоже, более того, тип «string» в C# является ссылочным типом аналогично классам. В этом легко убедиться, объявив переменную «string» и вызвав IntelliSense. В появившемся окне будут все методы класса «string» (рисунок 2.2).

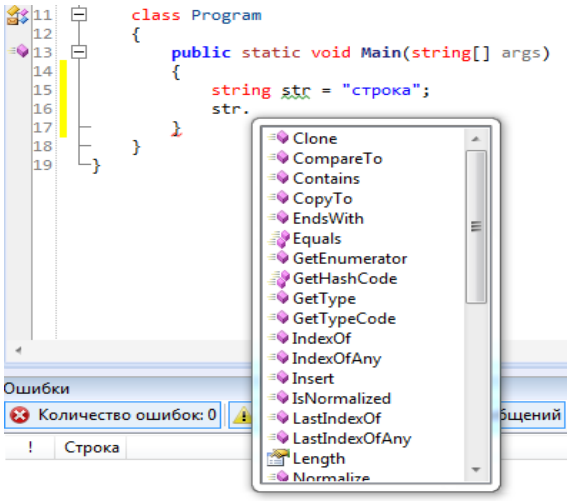


Рисунок 2.2 – IntelliSense строковой переменной

> IntelliSense – технология автодополнения Microsoft, наиболее известная в Microsoft Visual Studio. Дописывает название функции при вводе начальных букв. Кроме прямого назначения, IntelliSense используется для доступа к документации и для устранения неоднозначности в именах переменных, функций и методов, используя рефлексию [1].

При создании экземпляра класса ссылка на этот экземпляр передается программисту. В предыдущем примере oA является ссылкой на экземпляр объекта, основанный на типе «A». C# позволяет создать ссылку на объект без создания самого объекта:

```
ИмяКласса ИмяЭкземпляраОбъекта;
```

Для нашего случая это будет выглядеть так:

```
class A
{
}
class Program
{
  public static void Main(string[] args)
  {
    A oA1 = new A();
    A oA2;  // ссылка на объект без создания самого объекта
  }
}
```

Использование таких ссылок, не указывающих на объект, не рекомендовано и потенциально может привести к возникновению ошибки при попытке доступа к экземпляру объекта (рисунок 2.3).

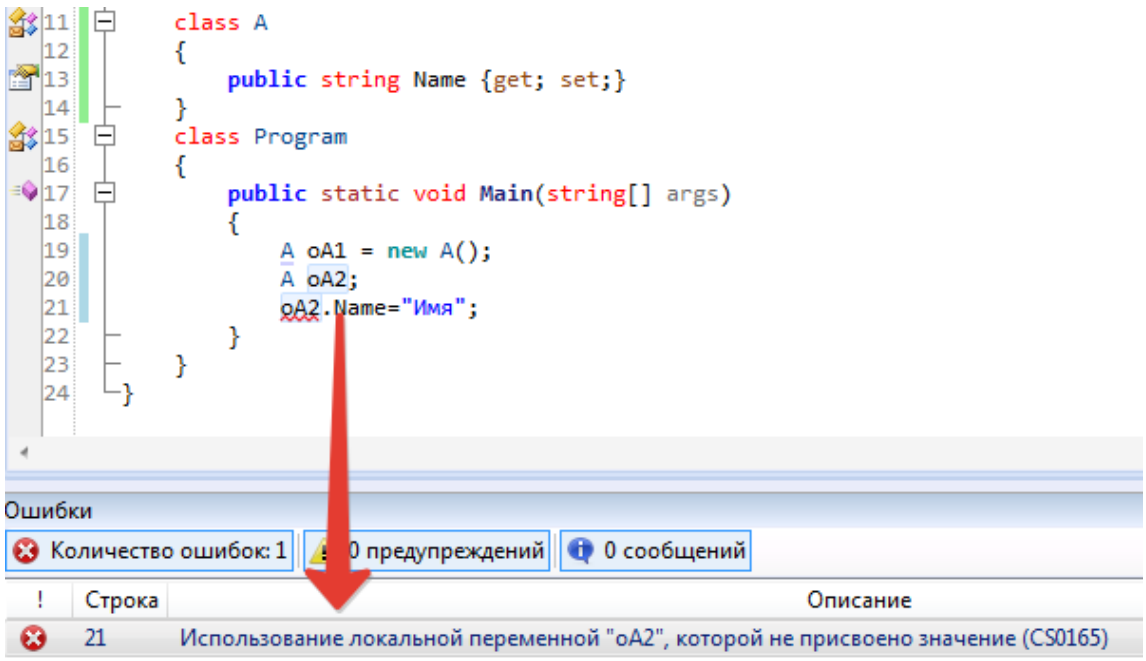


Рисунок 2.3 – Ошибка при использовании неинициализированного экземпляра объекта

Такие ссылки можно использовать после того, как ей будет назначена ссылка на существующий объект:

```
    class A
    {
        public string Name {get; set;}
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A oA1 = new A();
            A oA2;
            oA2 = oA1;
            oA2.Name="Имя";
        }
    }
```

В примере создаются две ссылки на экземпляры, которые указывают на один объект. Любые изменения, выполненные посредством oA1, будут видны при последующем использовании oA2. Это происходит потому, что классы – это ссылочные типы.

Пример:

```
    class A
    {
        public string Name {get; set;}
        public A()
        {
            this.Name="строка";
        }
    }
    class Program
    {
        public static void Main(string[] args)
        {
            // Создадим экземпляр объекта от класса A, вызвав конструктор без
параметров
            A oA1 = new A();
            // Конструктор без параметров инициализирует поле «Name» значени-
ем «строка»
            // Отобразим это значение на экране.
            Console.WriteLine(oA1.Name);

            // Создадим ссылку типа A
            A oA2;
            // Присвоем этой ссылке ссылку на конкретный экземпляр объекта
            oA2 = oA1;
            // Изменим значение поля «Name» по ссылке из oA2
            oA2.Name="новая строка";
            // Убедимся, что изменения произошли, используя ссылку oA1
            Console.WriteLine(oA1.Name);
        }
    }
```

2.4 Статические классы

Классы могут быть объявлены как статические при помощи ключевого слова «static». Статические классы практически ни чем не отличаются от обычных, за исключением того, что от статических классов нельзя создавать экземпляры, т. е. использование ключевого слова «new» не допустимо.

Так как нет переменной экземпляра класса, то доступ к членам класса возможен через использование самого имени класса. И конструктор перестает иметь смысл. Пример:

```
static class A
    {
        static public string Name {get; set;}
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A.Name = "строка";
            Console.WriteLine(A.Name);
            Console.ReadKey();
        }
    }
```

Чаще всего статические классы используются как контейнеры для методов, работающих на входных параметрах. Ярким примером служит встроенный в библиотеку .NET FrameWork статический класс Math, выполняющий математические операции.

Для статических классов характерны следующие признаки:

- внутри класса только статические члены,
- статические классы являются запечатанными (sealed),
- отсутствие конструкторов, следовательно, невозможность создания экземпляра класса.

Утверждение, что статический класс не имеет конструкторов, не совсем точное, т. к. в нем есть один закрытый конструктор, который инициализирует поля и не допускает создание экземпляров классов.

2.5 Статические члены класса

Обычные нестатические классы могут содержать статические члены, которые могут быть вызваны даже тогда, когда не создан экземпляр объекта. Доступ к таким членам осуществляется по указанию имени класса. При создании нескольких экземпляров класса статический член будет присутствовать только в одном экземпляре. Статические методы, свойства не могут обращаться к переменной экземпляра, если он не был передан явно в параметре метода; также статические свойства не могут обращаться к нестатическим полям.

Объявление статических членов осуществляется при помощи ключевого слова «static», которое ставится перед указанием возвращаемого типа. Инициализация статических полей выполняется перед первым доступом к статическому члену, а если имеется статический конструктор, то перед его вызовом.

```
public class A
{
    private static string _name;
    static A()
    {
        _name = "строка";
    }
    public static string Name
    {
        get
        { return _name; }
        set
        {
            _name = value;
        }
    }
}
class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine(A.Name);
        A.Name = "новая строка";
        Console.WriteLine(A.Name);
        Console.ReadKey();
    }
}
```

Использование статических полей оправдано для организации счетчика количества созданных экземпляров или для хранения значений, которые должны использоваться совместно всеми экземплярами.

Вопросы для самоконтроля

1. Понятие класса и объекта в ООП.
2. Отличие класса и объекта. Создание объектов.
3. Что такое объект и экземпляр объекта?
4. Как происходит размещение объектов/экземпляров в памяти при выполнении программы?
5. Как происходит объявление классов?
6. Особенности статических классов.
7. Объявление и использование статических членов в классе.

ГЛАВА 3
КОНСТРУКТОРЫ, СБОРКА МУСОРА И ДЕСТРУКТОРЫ

> Конструкторы – это специальные методы, выполняющие инициализацию полей класса и выделяющие память для экземпляра объекта в куче.

Синтаксис C# подразумевает, что имя конструктора совпадает с именем класса. «Специальность» конструктора заключается в том, что по сути это метод, но он явно не имеет тип возвращаемого параметра, однако это не так. Конструкторы возвращают ссылку на экземпляр объекта, следовательно, и возвращаемый параметр есть. Общий вид конструктора:

```
[доступ] имя_класса([список_параметров])
{
  // тело конструктора
}
```

Помимо выполнения инициализации полей класса, конструктор может выполнять определенные процедуры по формированию объекта. Чаще всего конструкторы имеют модификатор доступа «public», однако для реализации правила инкапсуляции можно инкапсулировать конструкторы и обеспечивать доступ к ним через специальные открытые конструкторы.

> Конструкторы по умолчанию не имеют параметров.

Даже если в классе явно не указан конструктор, он там все равно есть. Т. е. следующий код является верным:

```
namespace SubNewDefault
{
    class A
    {
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A oA = new A();
        }
    }
}
```

Как видим из примера, в классе A конструктора нет, но его можно вызвать, создав экземпляр класса в Main() методе.

Теперь добавим в класс конструктор по умолчанию:

```
namespace SubNewDefault
{
    class A
    {
        public A()
        {
            Console.WriteLine("Конструктор без параметров класса A");
        }
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A oA = new A();
        }
    }
}
```

Как видим, никаких замечаний и ошибок компилятор не генерирует. Добавим к классу конструктор с параметром – перегрузим конструктор.

```
    class A
    {
        public A()
        {
            Console.WriteLine("Конструктор без параметров класса A");
        }
        public A(string a)
        {
            Console.WriteLine("Конструктор с параметром (параметр={0}) класса
A",a);
        }
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A oA1 = new A();
            A oA2 = new A("строка");

            Console.ReadKey();
        }
    }
```

В Main() методе создадим еще один экземпляр класса A, но конструктор вызовем с параметром. В результате получим:

```
Конструктор без параметров класса A;
Конструктор с параметром (параметр=строка) класса A
```

Теперь закомментируем конструктор без параметров. По идее, ничего не должно произойти, код по-прежнему должен компилироваться, т. к. ранее было сказано, что у класса есть конструктор по умолчанию, даже если он не написан в коде. Однако такое действие приведет к ошибке компилятора «Класс не содержит конструктор, принимающий 0 аргументов». Получается, что теперь конструктора по умолчанию нет в классе. Все дело в том, что в данном случае работает сокрытие и любой объявленный конструктор класса скрывает конструктор по умолчанию.

Итак, компилятор понимает, какой из конструкторов необходимо вызвать по количеству аргументов в нем, а если количество аргументов в конструкторе класса совпадает, то выбор конструктора будет произведен по типу аргумента. Следовательно, в классе может быть сколько угодно конструкторов, главное условие, чтобы их сигнатуры не совпадали.

Сборка мусора и деструкторы

Рассмотрев понятия класс, объект и экземпляр объекта, стало понятно, что ключевое слово «new» при объявлении экземпляра класса приводит к тому, что объект размещается в управляемой куче (там, где размещаются все ссылочные типы), а в результате возвращается ссылка на этот объект. Освобождением памяти в среде CLR занимается автоматический диспетчер памяти, так называемый сборщик мусора. Использование сборщика мусора приводит к следующим преимуществам:

- программисту не нужно заботиться об освобождении памяти;
- эффективно используется память при размещении экземпляров в управляемой куче;
- обеспечивается безопасность памяти.

Сбор мусора выполняется по определенным алгоритмам. Например, сбор мусора происходит не сразу после завершения работы объекта, а в случае, когда недостаточно физической памяти в системе, превышен порог допустимого использования памяти в куче, принудительно вызван метод GC.Collect. В реальности процесс сбора мусора происходит в непрерывном режиме.

Все экземпляры объектов в куче разделены на несколько поколений. Поколение 0 – недавно созданный объект, который не отмечался как подлежащий удалению. Поколение 1 – объекты, помеченные на удаление, но не удаленные из-за достаточного объема памяти в куче для хранения объектов и «пережившие» хотя бы одну итерацию сборки мусора. Поколение 2 – объекты, которые не были удалены более чем за одну итерацию работы сборщика мусора.

Объект считается ненужным в том случае, когда ссылки на него отсутствуют.

|| Эфемерные поколения – поколения 0 и 1.

Первым делом сборщик мусора анализирует те объекты, которые принадлежат поколению 0, в результате их анализа он удаляет те, на которые больше нет ссылок. Если после их удаления освободилось достаточно памяти, то все оставшиеся объекты поколения 0 переходят в поколение 1. В случае, если памяти оказалось мало, то сборщик анализирует все объекты, принадлежащие поколению 1, и аналогично удаляет устаревшие. Объекты, которые не были удалены, после такого прохода помечаются как объекты принадлежащие поколению 2. Для второго поколения аналогично, но выжившие объекты остаются принадлежащими поколению 2. Таким образом, объекты поколения 0 удаляются быстрее объектов поколения 2.

Среда CLR оперирует следующими понятиями памяти:

1. Все процессы имеют свое собственное виртуальное адресное пространство.
2. Для 32-битных ОС для каждого процесса отводится по 2 ГБ.
3. Программисту доступна работа только с виртуальным адресным пространством.
4. Виртуальная память имеет три состояния:

4.1. Свободная – доступна для выделения.

4.2. Зарезервирована – доступна для определенного объекта и не выделяется под использование другими объектами.

4.3. Выделена – назначена какому-либо физическому хранилищу.

5. Виртуальная память подвержена фрагментированию.

Вопросы для самоконтроля

1. Инициализация полей класса. Инициализация с помощью конструкторов.
2. Конструкторы по умолчанию и особенности их использования.
3. В чем смысл деструктора и сборщика мусора?
4. Раскройте понятие «эфемерные поколения».

ГЛАВА 4
ДЕСТРУКТОРЫ

В языке C# предусмотрен специальный метод, который вызывается сборщиком мусора перед уничтожением объекта.

> Метод завершения (деструктор) – специальный метод, используемый сборщиком мусора, для окончательной очистки памяти.

Деструкторы присутствуют только в классах, структуры не могут содержать деструкторы. В классе может быть объявлен только один деструктор. Деструкторы не перегружаются и не наследуются, не принимают параметров и модификаторов. Деструкторы аналогично конструкторам имеют имя, совпадающее с именем класса, перед которым ставится знак ~.

```
~имя_класса ()
{
  // команды, выполняемые при завершении
}
```

Пример деструктора для класса A:

```
class A
{
  ~A()
  {
    // команды, выполняемые при завершении
  }
}
```

Деструктор неявно вызывает метод Finalize(), поэтому деструктор из предыдущего примера будет преобразован в следующий код:

```
protected override void Finalize()
{
  try
  {
    // команды, выполняемые при завершении
  }
  finaly
  {
    base.Finalize();
  }
}
```

Следует понимать, что деструктор не освобождает память, это лишь метод, вызываемый сборщиком мусора перед тем, как освободить память от объекта. Сборщик мусора можно вызвать принудительно при помощи метода Collect.

> GC.Collect – метод принудительного вызова сборщика мусора.

Вызов метода Finalize() происходит рекурсивно для всех экземпляров в иерархии наследования от низшего к высшему (от потомков к предкам).

```
using System;
using System.Diagnostics;

namespace DestructorExample
{
  class A
  {
    ~A()
    {
      Debug.WriteLine("Класс A завершен");
    }
  }
  class B:A
  {
    ~B()
    {
      Debug.WriteLine("Класс B завершен");
    }
  }
  class C:B
  {
    ~C()
    {
      Debug.WriteLine("Класс C завершен");
    }
  }



  class Program
  {
    public static void Main(string[] args)
    {
      C oB = new C();
    }
  }
}
Результат:
Класс C завершен
Класс B завершен
Класс A завершен
```

Вызов деструктора приводит к вызову сборщика мусора, если блок finaly отказался пустым, то сборщик мусора будет вызван лишний раз, что приведет к снижению производительности. Как было сказано выше, сборщик автоматически вызывается системой с определенным интервалом, любой дополнительный его вызов лишь замедлит работу системы в целом. Стоит избегать принудительного вызова сборщика, если это не требуется условиями работы алгоритма.

Рассмотрим пример использования деструкторов:

```
using System;
using System.Diagnostics;

namespace DestructorExample
{
  class A
  {
    private int i;
    public A(int arg)
    {
      i = arg;
    }
    ~A()
    {
      Debug.WriteLine("Объект {0} уничтожен", i);
    }
  }
  class Program
  {
    public static void Main(string[] args)
    {
        int i = 1;

        for (i=0;i<10;i++)
        {
            A oB = new A(i);
          }
    }
  }
}
Результат:
Объект 9 уничтожен Объект 8 уничтожен Объект 7 уничтожен Объект 6 уничтожен
Объект 5 уничтожен Объект 4 уничтожен Объект 3 уничтожен Объект 2 уничтожен
Объект 1 уничтожен Объект 0 уничтожен
```

Порядок вызова деструкторов не определен точно, отсюда непоследовательное их завершение.

Вопросы для самоконтроля

1. Работа метода GC.Collect.
2. Рекурсивный вызов метода Finalize().
3. Особенности наследования деструкторов.
4. Принудительный сбор мусора.

ГЛАВА 5
КЛЮЧЕВОЕ СЛОВО «THIS»

Ключевое слово «this» обеспечивает доступ к текущему экземпляру класса.

Ключевое слово «this» применяется в разных контекстах (например, для разрешения неоднозначности контекста). Часто в методах имя входящего параметра совпадает с названием поля.

```
        private string Name;
        public void M(string Name)
        {
            Name=Name;
        }
```

Компилятор успешно выполнит компиляцию проекта хотя и отобразит замечание о том, что переменная присваивается сама себе. Но с точки зрения читаемости кода, для программиста это неудобно, такая неоднозначность может затруднять понимание кода. Решить эту проблему можно при помощи ключевого слова «this.»

```
        private string Name;
        public void M(string Name)
        {
            this.Name=Name;
        }
```

Однако лучше избегать таких неоднозначностей и использовать правила именования переменных.

Еще одним «сахаром» при использовании this является использование IntelliSense.

5.1 Constructor chaining

Полезным является использование ключевого слова «this» при конструировании классов с использованием шаблона проектирования «сцепление конструкторов» или «цепочка конструкторов». Такой шаблон полезен, когда в классе много конструкторов и каждый из них имеет некую общую часть или, например, один конструктор дополняет другой.

```
    class A
    {
        public int Age{get;set;}
        public string Name {get;set;}
        public A(string _name) : this (_name,0)
        {
            this.Name = _name;
        }
        public A(int _age) : this ("",_age)
        {
            this.Age = _age;
        }
        public A(string _name, int _age)
        {
            this.Name = _name;
            this.Age = _age;
        }
    }
```

Шаблон программирования с использованием цепочки конструкторов работает с любой платформой .NET.

В .NET 4.0 можно использовать необязательные аргументы как альтернативу цепочкам конструкторов.

5.2 Последовательность срабатывания конструкторов

В иерархии наследования классов выполнение конструкторов начинается от базового класса к производному – вниз по иерархии. Это логично и объясняется следующим: для того, чтобы производный класс мог наследовать базовый, необходимо, чтобы экземпляр базового уже существовал. Деструкторы срабатывают в обратном порядке. Рассмотрим пример, иллюстрирующий последовательность срабатывания конструкторов при наследовании классов.

```
class A
{
    public A()
    {
        Console.WriteLine("Конструктор класса A");
    }
}
class B:A
{
    public B()
    {
        Console.WriteLine("Конструктор класса B");
    }
}
class C:B
{
    public C()
    {
        Console.WriteLine("Конструктор класса C");
    }
}
class Program
{
    public static void Main(string[] args)
    {
        C oC = new C();
        Console.ReadKey(true);
    }
}
```

В результате мы получим:

```
Конструктор класса A
Конструктор класса B
Конструктор класса C
```

То есть в примере необходимо было создать экземпляр класса C, который наследовал B, а тот, в свою очередь, наследовал A. В результате вначале был создан экземпляр объекта A, выполнен его конструктор, затем аналогично для класса B. И в конце был создан необходимый нам класс C.

Вопросы для самоконтроля

1. В чем особенность и преимущество использования ключевого слова «this»?
2. Контексты использования ключевого слова «this».
3. Взаимодействие ключевого слова «this» и IntelliSense.
4. Последовательность срабатывания конструкторов.

ГЛАВА 6
МОДИФИКАТОРЫ ДОСТУПА

Объектно-ориентированный подход подразумевает два типа членов класса – открытые и закрытые. В каждом языке программирования есть свои тонкости и дополнительные модификаторы доступа. Более того, существует возможность получить доступ к закрытым членам класса, используя рефлексию.

Идея разделения доступа на члены класса реализует понятие объектно-ориентированного подхода – инкапсуляцию.

> Инкапсуляция – это управляемый доступ к закрытым членам класса.

Ярким примером инкапсуляции служат поля и свойства. Принято, соблюдая правило инкапсулирования, поля класса объявлять как закрытые – private, а доступ к этим полям обеспечивать посредством публичных (public) свойств.

> Свойства – это специальные методы, обеспечивающие доступ к закрытым полям класса.

Предположим, в классе объявлено public поле типа float для хранения числителя рассчитываемой в методе M() дроби. Если поле объявлено как public, то оно доступно из другого класса и ему можно неконтролируемо присвоить любое значение, в том числе и ноль. Тип float допускает хранение значения 0 в переменной, поэтому никаких ошибок и исключений не возникнет, но при расчете дроби будет сгенерировано исключение – деление на ноль. Правильным решением будет объявление этого поля ключевым словом «private», а также реализовать контролируемый доступ к нему через свойство.

Контроль доступа к членам класса – это соблюдение правила инкапсуляции. Контролируя доступ к полям класса с помощью специальный методов, позволяет предупредить присваивание неверных значений этим данным.

> Доступ к закрытым членам класса извне невозможен.

Класс, реализованный с соблюдением понятия инкапсуляции, похож на «черный ящик», внутренний механизм которого закрыт для вмешательства извне, но пользоваться этим классом возможно.

Следует знать, что инкапсулируются не только поля, но и методы, иногда конструкторы класса. Также нужно помнить, что инкапсуляция – это не полное сокрытие данных от воздействия извне класса, а контроль доступа к закрытым полям.

Модификаторы доступа C#

В языке C# существует шесть модификаторов доступа, однако не все они могут использоваться всеми типами и членами. В некоторых случаях доступность члена ограничивается доступностью типа, в котором он содержится. Рассмотрим модификаторы доступа, описанные на официальном сайте Microsoft https://docs.microsoft.com/ru-ru/dotnet/csharp/ (таблица 6.1).

Таблица 6.1 – Модификаторы доступа языка C#

Модификатор доступа	**Описание**
1. Public	Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него
2. Private	Доступ к типу или члену возможен только из кода в том же классе или структуре
3. Protected	Доступ к типу или члену возможен только из кода в том же классе либо в классе, производном от этого класса
4. Internal	Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки
5. Protected internal	Доступ к типу или члену возможен из любого кода в той сборке, где он был объявлен, или из производного класса в другой сборке
6. Private protected	Доступ к типу или члену возможен только из его объявляющей сборки из кода в том же классе либо в типе, производном от этого класса

Вопросы для самоконтроля

1. Модификаторы доступа и их назначение.
2. В чем преимущества инкапсуляции?
3. Какова взаимосвязь инкапсуляции, свойств и модификаторов доступа?
4. Доступ к закрытым членам класса извне.

ГЛАВА 7
ПОЛЯ КЛАССА

> Поле класса – это переменная или константа любого типа, объявленная непосредственно в классе или структуре.

Поля являются членами классов. Поля в классах используются для хранения значений в классе или структуре как обычные переменные. Принято, но не обязательно, поля помечать как private для соблюдения концепции инкапсуляции.

Поля объявляются в классе следующим образом:

```
[модификатор доступа] [static] ТипПоля ИмяПоля [= начальное значение];
```

Пример:

```
public class A
{
    string Name;
    private DateTime date;
    private int Age = 25;
    private static int Kurs;
}
```

Поля инициализируются перед вызовом конструктора. Если конструктор тоже инициализирует эти поля, то значения из конструктора заменят значения, присвоенные при объявлении поля.

```
public class A
{
    string Name;
    private DateTime date;
    private int Age = 25;
    private static int Kurs;
    public A()
      {
        Name = "строка";
        Age = 18;
}
```

Поля могут быть отмечены одним из модификаторов: public, private, protected, internal или protected internal, static, readonly (значения могут быть присвоены только при инициализации или в конструкторе).

Поле «static readonly» (статическое, доступное только для чтения) есть не что иное, как константа.

Вопросы для самоконтроля

1. Что такое поля класса?
2. Пример объявления полей класса.
3. Статистические поля класса.

ГЛАВА 8
КОНСТАНТЫ

> Константы представляют собой неизменные значения, известные во время компиляции и неизменяемые на протяжении времени существования программы.

Для объявления констант используется модификатор const, константами могут быть встроенные типы, кроме Object. Методы, события, свойства быть константами не могут. Константы обязаны быть инициализированы сразу после их объявления.

```
class A
{
    public const string Name = "студент";
}
```

Возможно объявление нескольких констант одного типа одновременно:

```
class A
{
    const int Age = 19, Kurs = 2;
}
```

Вопросы для самоконтроля

1. Какие типы могут быть объявлены как const?
2. Модификаторы доступа констант.
3. Одновременное объявление нескольких констант.

ГЛАВА 9
СВОЙСТВА

> Свойство – это специальный метод (член класса), предоставляющий гибкий механизм для чтения, записи или вычисления значения закрытого (private) поля.

Свойства представляют из себя специальные методы доступа к закрытым полям класса (назовем их блоки) и обеспечивают понятие инкапсуляции. Свойства состоят из двух блоков get и set. Блок get (геттер) предназначен для возврата значения, а блок set – для его установки. Как правило, для каждого закрытого поля создается свой свойство.

Схематически принцип работы и назначение свойств можно изобразить следующим образом. Пусть имеется два класса A и B. В классе A есть два закрытых поля name и age. Предположим, из класса B необходимо получить доступ к значению поля age в классе A (например, записать туда новое значение). Однако, выполняя требование инкапсуляции, поле age скрыто (объявлено private), т. е. напрямую доступ к нему получить невозможно (рисунок 9.1).

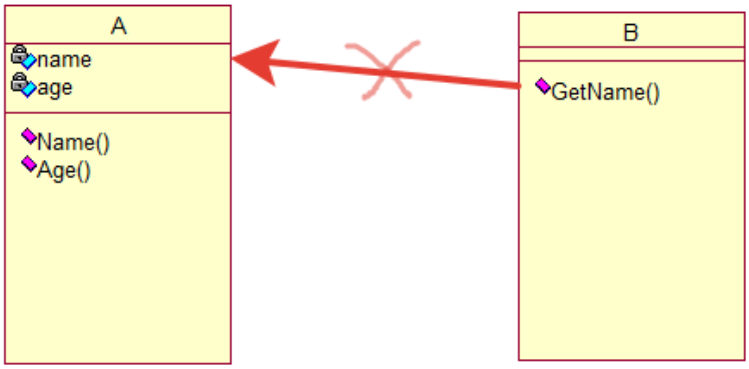


Рисунок 9.1 – Принцип работы и назначение свойств (неправильно)

Для этого реализовано свойство Age того же типа, что и поле age. В свойстве есть два блока get и set, позволяющие производить доступ к полю. Т. е. обращение из класса B идет не на прямую к полю, а через свойство. Таким образом, в блоках get и set можно реализовать проверку на область допустимых значений (для возраста, например, можно выполнять проверку на *«меньше нуля»* и на *«18+»*), опять же, реализовав понятие инкапсуляции.

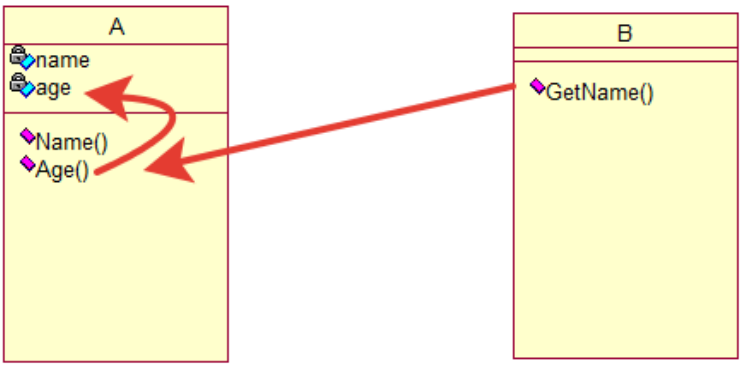


Рисунок 9.2 – Принцип работы и назначение свойств (правильно)

В коде это будет выглядеть следующим образом:

```
public class A
{
    private string name;
    private int age;

    public int Age
    {
        get
        {
            return this.age;
        }
        set
        {
            if (value>=0)
              this.age = value;
        }
    }
}
public class B
{
    A oA = new A();
    oA.Age = 33;
}
```

В примере выше отсутствует какая-либо проверка, т. е. в поле age можно положить любое значение. Модернизируем пример с учетом такой проверки.

Теперь стоит разобраться с переменной value. Она в коде нигде не объявлена, но мы можем ей оперировать внутри свойства. Все дело в том, что свойства – это специальные методы и одна из их «специальностей» заключается в том, что блоки get и set на самом деле компилятором преобразуются в два отдельных метода.

```
public int getAge()
{
    return this.age;
}
public void setAge(int value)
{
    this.age = value;
}
```

Из примера видно, что аргумент value имеет тот же тип, что и само свойство, а соответственно, и поле.

> Ключевое слово «value» используется для определения значения, присваиваемого методом доступа set.

Таким образом, свойства предоставляют общий способ чтения и задания значений, скрывая при этом код реализации или проверки.

> Блок get используется для возврата значения свойства.
> Блок set используется для присвоения нового значения.

Блоки get и set могут иметь различные уровни доступа.

Свойства могут не реализовывать один из блоков (либо get, либо set).

Свойство, которое не имеет блока set, может только считывать значение и называется свойством только для чтения (ReadOnly).
Свойство, которое не имеет блока get, может только записывать / устанавливать значение и называется свойством только для записи (WriteOnly).

Рассмотрим пример организации таких свойств.

```
public class A
{
    private string name;
    private int age;

    public int Age
    {
        set
        {
            if (value>=0)
                this.age = value;
        }
    }
    public string Name
    {
        get
        {
            return this.name;
        }
    }
}
```

9.1 Использование свойств

Свойства объединяют функции полей и методов. Для объекта, использующего какой-либо объект, свойство является полем, поэтому для доступа к свойству требуется тот же синтаксис, что и для поля.

В отличии от полей свойства не являются переменными, следовательно, свойство невозможно передать в качестве параметра ref.

Свойства используются для проверки данных перед их изменением, могут производить преобразование к нужному формату при записи в поле или считывании из него, могут быть использованы при программировании баз данных для вызова события или изменении значения в других полях.

9.2 Автоматически реализуемые свойства

Описанный ранее подход, при котором принято поля класса делать скрытыми, а для обеспечения доступа к ним предусматривать свойства, имеет лишь один недостаток – каждый раз программисту приходится прописывать в коде вначале само поле, а затем свойство для него. Особенно это выглядит бессмысленным, когда необходимо просто объявить «перемену» в классе без какого-либо контроля доступа, приходится делать «лишние телодвижения» и перегружать код. В C# предусмотрены автоматически реализуемые свойства, которые позволяют решить эту проблему. Синтаксис такого свойства очень напоминает объявление обычного поля.

```
модификатор_доступа тип_свойства Имя_свойства {[get;][set;]}
```

Например, для объявления ReadWrite свойства Name код будет выглядеть следующим образом:

```
public string Name {get;set;}
```

Если необходимо сделать только Read либо только Write, достаточно убрать один из блоков:

```
public string Name {set;} // только для записи
public int Age {get;} // только для чтения
```

Тут же можно изменять область видимости для отдельного блока, например:

```
public string Name {get; protected set;}
```

Для автогенерируемых свойств можно устанавливать значения по умолчанию, например:

```
public string Name {get;set;} = "Любая строка";
```

На самом деле компилятор, встретив синтаксис автогенерируемого свойства, преобразует его в обычное свойство, сделав для него закрытое поле. Имя этого поля будет выглядеть как набор случайных символов, но по факту программисту это не нужно, т. к. он работает с ним через имя свойства.

Выделим несколько правил и ограничений при использовании свойств:

- использование имени неявного параметра value для объявления локальной переменной в методе доступа set является ошибкой;
- свойства могут быть объявлены как public, private, protected, internal или protected internal;
- методы доступа get и set могут иметь различные модификаторы доступа для одного свойства.

Метод доступа get может иметь модификатор public, а set – модификатор private или protected.

> Свойство можно объявить как статическое свойство при помощи ключевого слова «static».

Таким образом, свойство становится доступным для вызова в любое время, даже если экземпляр класса отсутствует.

> Свойство можно пометить как виртуальное свойство при помощи ключевого слова «virtual».

Это позволяет производным классам переопределять поведение свойства при помощи ключевого слова «override».

> Свойство, переопределяющее виртуальное свойство, может также быть sealed.

Это указывает на то, что для производных классов оно более не является виртуальным.

|| Свойство может быть абстрактным (abstract).

Это означает, что в классе отсутствует реализация, поэтому производные классы должны создавать свою собственную реализацию.

9.3 Свойства в интерфейсах

В интерфейсах можно объявлять свойства. По требованиям к интерфейсам у объявляемых членов интерфейса не должно быть реализации. Рассмотрим пример интерфейса с объявлением свойства:

```
public interface IMyInterface
{
    string Name
    {
        get;
        set;
    }
}
```

По сути методы доступа в интерфейсах необходимы, чтобы указать, каким будет свойство (RW, RO, WO).

Вопросы для самоконтроля

1. Что такое свойство в классе?
2. Реализация принципа инкапсуляции через специальные методы-свойства.
3. Переменная value в теле свойства. В чем особенности ее объявления и использования?
4. Блоки свойств: Блок GET.
5. Блоки свойств: Блок SET.
6. Типы свойств RW RO WO.
7. Автоматически реализуемые свойства, преимущества и недостатки их использования.
8. Статические свойства.
9. Виртуальные свойства.
10. Абстрактные свойства.
11. Использование свойств при объявлении интерфейсов.

ГЛАВА 10
НАСЛЕДОВАНИЕ: ЗАПЕЧАТАННЫЕ КЛАССЫ, ЧЛЕНЫ КЛАССОВ

> Наследование – механизм создания нового класса на основе уже существующего старого.

При наследовании все члены родительского класса «переходят» и становятся доступными в дочернем классе, кроме конструкторов и деструкторов, а также случаев, когда члены помечены ключевым словом «sealed», запрещающим наследование; класс помеченный как sealed не наследуется подобно абстрактному классу.

> Старый класс называется «родительским», «предком» («super class»).
> Новый класс называется «дочерним», «наследником» («sub class»).

Наследование реализует принцип повторного использования кода, что способствует следование принципу DRY (Don't Repeat Yourself – не повторяйся).

> В C# множественное наследование не поддерживается.

У класса может быть только один родительский класс. Дочерний класс, как говорилось ранее, содержит методы и переменные родительского. Синтаксис наследования выглядит следующим образом:

```
class A : B
{
}
```

Символ «:» (двоеточие) обозначает наследование.
Наследование является транзитивным, т. е. если структура наследования имеет вид:

```
Class A
     |__class B
              |__class C,
```

и у класса A есть члены, то они будут доступны в классе C.

Рассмотрим наследование на примерах. Пусть имеется класс A с определенными членами, например конструктор, свойство, метод. Класс B будет наследовать класс A, причем в классе B не будет объявлено ни один из членов.

```
    /// <summary>
    /// Класс A будет родительским классом для класса B
    /// </summary>
    class A
    {
        /// <summary>
        /// конструктор класса A по умолчанию.
        /// </summary>
        public A()
        {

        }

        /// <summary>
        /// RW свойство Name
        /// </summary>
        public string Name {get;set;}

        /// <summary>
        /// Метод класса, принимающий аргумент и выводящий его в консоль
        /// </summary>
        /// <param name="arg">аргумент строкового типа</param>
        public void MyMethod(string arg)
        {
            Console.WriteLine(arg);
        }

    }
    /// <summary>
    /// Класс B наследует класс A,
    /// т.е. является дочерним для класса A
    /// </summary>
    class B:A
    {

    }
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Наследование классов\n");
            // Создадим экземпляры oA и oB для классов A и B соответственно:
            A oA = new A();
            B oB = new B();

            // Вызовем метод MyMethod у экземпляра класса oB:
            oB.MyMethod("тест");

            Console.ReadKey(true);
        }
    }
```

Как видно из примера, в классе B не объявлено ни одного члена, но тем не менее в примере можно вызвать его метод MyMethod(), который был унаследован от родительского класса A. Результатом работы программы будет:

```
Наследование классов

тест
```

Обратим внимание, что в Main() методе мы создали экземпляр класса A, хотя этого делать было необязательно. Если закомментировать строку кода:

```
            A oA = new A();
```

Результат работы останется прежним, т. к. компилятор, встретив команду наследования, создаст экземпляры объектов всей цепочки. Более подробно, почему такое возможно, можно почитать в разделе о конструкторах.

Интересно заметить, что если в родительском классе A объявить закрытое private-поле, то использовать его в дочернем поле не получится в силу его уровня защиты, так, компиляция следующего кода невозможна.

```
    /// <summary>
    /// Класс A будет родительским классом для класса B
    /// </summary>
    class A
    {
        /// <summary>
        /// Закрытое целочисленное поле
        /// </summary>
        private int Age1 = 10;
        public int Age2 = 20;
        /// <summary>
        /// конструктор класса A по умолчанию.
        /// </summary>
        public A()
        {

        }

        /// <summary>
        /// RW свойство Name
        /// </summary>
        public string Name {get;set;}

        /// <summary>
        /// Метод класса, принимающий аргумент и выводящий его в консоль
        /// </summary>
        /// <param name="arg">аргумент строкового типа</param>
        public void MyMethod(int arg)
        {
            this.Age1 = arg;
            Console.WriteLine(this.Age1);
        }

    }
    /// <summary>
    /// Класс B наследует класс A,
    /// т.е. является дочерним для класса A
    /// </summary>
    class B:A
    {
        public int M()
        {
            this.Age1 = 10;
            return this.Age1;
        }
    }
    class Program
```

```
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Наследование классов\n");
            // Создадим экземпляры oA и oB для классов A и B соответственно
            //A oA = new A();
            B oB = new B();

            // Вызовем метод MyMethod у экземпляра класса oB
            oB.MyMethod(25);

            Console.ReadKey(true);
        }
    }
```

Наследование предоставляет возможность повторно воспользоваться уже написаным и отлаженым ранее кодом (наследовать какой-либо класс) и дополнить этот код новым функционалом. То сеть все члены, которые будут добавлены в дочерний класс B, будут функционировать, расширяя тем самым возможности базового класса A. В качестве примера рассмотрим следующий код, в котором базовый класс A будет иметь метод M1(), а дочерний класс B будет расширять функционал, добавляя новый метод M2().

```
    /// <summary>
    /// Класс A будет родительским классом для класса B
    /// </summary>
    class A
    {
        /// <summary>
        /// Метод класса.
        /// </summary>
        public void M1()
        {
            Console.WriteLine("Метод M1() работает ...");
        }

    }
    /// <summary>
    /// Класс B наследует класс A,
    /// т.е. является дочерним для класса A
    /// </summary>
    class B:A
    {
        /// <summary>
        /// Метод класса.
        /// </summary>
        public void M2()
        {
            Console.WriteLine("Метод M2() работает ...\n");
        }

    }
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Наследование классов\n");
            // Создадим экземпляры oA и oB для классов A и B соответственно
            //A oA = new A();
            B oB = new B();
```

```
            // Вызовем метод M1 у экземпляра класса oB:
            oB.M1();

            // Вызовем метод M2 у экземпляра класса oB:
            oB.M2();

            Console.ReadKey(true);
        }
    }
```

Кроме добавления методов существует возможность изменения их функционала, о котором будет рассказано ниже.

Запечатанные классы, члены классов

Как говорилось ранее, можно запретить наследовать отдельные члены класса или весь класс. Для этого необходимо запечатать член класса или класс ключевым словом «sealed».

> Применение sealed для методов должно быть обязательно с использованием ключевого слова «override».

О переопределении методов будет рассказано ниже.

Рассмотрим запечатывание классов на следующем примере: создадим запечатанный класс A, выполним наследование класса A классом B и попытаемся вызвать метод M1() из экземпляра класса B.

```
    /// <summary>
    /// Класс A будет родительским классом для класса B
    /// </summary>
    sealed class A
    {    }
    /// <summary>
    /// Класс B наследует класс A,
    /// т.е. является дочерним для класса A
    /// </summary>
    class B:A
    {    }
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Наследование классов\n");
            // Создадим экземпляры oA и oB для классов A и B соответственно
            //A oA = new A();
            B oB = new B();
            Console.ReadKey(true);
        }
    }
```

При компиляции строки кода B oB = new B(); возникнет ошибка, т. к. наследовать запечатанный класс невозможно.

> Структуры запечатаны неявно, поэтому применять к ним sealed невозможно.

Для предотвращения переопределения производных классов при определении новых методов или свойств не стоит помечать их как виртуальные (virtual).

Абстрактные классы предназначены для наследования их другими классами, создавать экземпляры абстрактных классов нельзя, поэтому использовать модификатор abstract для запечатанных классов недопустимо.

Вопросы для самоконтроля

1. Особенности наследования классов.
2. Что такое родительский и дочерний классы? Их отличия.
3. Множественное наследование.
4. Что предоставляет принцип наследования разработчику?
5. Суть запечатывания классов. Какие классы называются запечатанными?
6. Запечатывание структур.

ГЛАВА 11
ПОЛИМОРФИЗМ

Полиморфизм – один интерфейс и много методов.

Что нужно запомнить: ничто не может помешать создать в дочернем классе такой же метод, как и в родительском.

Когда вызывается a.Display1(), C# сначала ищет Display1() в ClassA, а только потом в ClassB. Поскольку в A такой метод есть, вызывается именно он.

Что нужно запомнить: методы дочерних классов имеют приоритет при выполнении.

Такая возможность даётся для того, чтобы можно было изменить поведение методов предка, если оно не устраивает. Однако всё равно есть возможность вызывать методы родительского класса следующим образом:

```
ClassA:

  class ClassA:ClassB
    {
        public void Display1()
        {
            Console.WriteLine("ClassA Display1");
            base.Display1();
        }
    }
```

Результат:

- ClassA Display1;
- ClassB Display1.

Что нужно запомнить: ключевое слово «base» может быть использовано для обращения к методам класса-предка.

Вверх по иерархии обращаться есть возможность. Попробуем сделать наоборот:

```
/// <summary>
/// ClassB: выступает в роли класса-предка
/// </summary>
class ClassB
 {
     public int x = 100;
     public void Display1()
     {
         Console.WriteLine("ClassB Display1");
     }
 }

 /// <summary>
 /// ClassA: выступает в роли класса-наследника
 /// </summary>
 class ClassA : ClassB
 {
     public void Display2()
     {
         Console.WriteLine("ClassA Display2");
     }
 }

 /// <summary>
 /// Program: используется для выполнения кода.
```

```
 /// Contains Main method.
 /// </summary>
 class Program
 {
     static void Main(string[] args)
     {
         ClassB b = new ClassB();
         b.Display2();
         Console.ReadKey();
     }
 }
```

Error: ‘InheritanceAndPolymorphism.ClassB’ does not contain a definition for ‘Display2’ and no extension method ‘Display2’ accepting a first argument of type ‘InheritanceAndPolymorphism.ClassB’ could be found (are you missing a using directive or an assembly reference?)

Что нужно запомнить: наследование не работает в обратном направлении.

Когда класс A наследуется от B, он получает все его методы и может их использовать. Однако методы, которые были добавлены в A, не загружаются наверх в B, наследование не имеет обратной совместимости. Если попытаться вызвать из класса-родителя метод, который создан в классе-наследнике, получится ошибка.

Что нужно запомнить: кроме конструкторов и деструкторов, дочерний класс получает от родителя абсолютно всё.

Если класс C будет унаследован от класса B, который, в свою очередь, будет унаследован от класса A, то класс C унаследует члены как от класса B, так и от класса A. Это транзитивное свойство наследования. Потомок перенимает все члены родителей и не может исключить какие-либо. Он может «спрятать» их, создав свой метод с тем же именем. Конечно, это никак не повлияет на родительский класс, просто в дочернем метод не будет виден.

Члены класса могут быть двух типов: статический, который принадлежит именно классу, или обычный, который доступен только из реализаций класса (его объектов). Чтобы сделать член статическим, нужно использовать ключевое слово «static.

Если класс не наследуется ни от какого другого, подразумевается, что он наследуется от класса object. Это родитель всех классов, он единственный не унаследован ни от чего. Таким образом, такой код:

```
public class ClassB
 {
 }

 public class ClassA : ClassB
 {
 }
```

Автоматически воспринимается C# так:

```
public class ClassB:object
{
}
public class ClassA : ClassB
{
}
```

По свойству транзитивности ClassA также является наследником object.

Теперь ещё один момент. Если нужно сделать так:

```
public class ClassW : System.ValueType
  {
  }

  public class ClassX : System.Enum
  {
  }

  public class ClassY : System.Delegate
  {
  }

  public class ClassZ : System.Array
  {
  }
```

То это не получится, так как компилятор языка C# выдаст следующие ошибки:

– 'InheritanceAndPolymorphism.ClassW' cannot derive from special class 'System.ValueType';

– 'InheritanceAndPolymorphism.ClassX' cannot derive from special class 'System.Enum';

– 'InheritanceAndPolymorphism.ClassY' cannot derive from special class 'System.Delegate';

– 'InheritanceAndPolymorphism.ClassZ' cannot derive from special class 'System.Array'.

Словосочетание «special class»!!! Такие классы нельзя расширять.

Что нужно запомнить: классы не могут быть унаследованы от встроенных классов вроде System.ValueType, System.Enum, System.Delegate, System.Array и т. д. И ещё кое-что:

```
public class ClassW
  {
  }

  public class ClassX
  {
  }

  public class ClassY : ClassW, ClassX
  {
  }
```

Выше описаны три класса: ClassW, ClassX и ClassY, который наследуется от первых двух. Теперь попробуем это скомпилировать, после этого появится ошибка:

–Compile time Error: Class 'InheritanceAndPolymorphism.ClassY' cannot have multiple base classes: 'InheritanceAndPolymorphism.ClassW' and 'ClassX'.

Что ещё нужно запомнить: класс может иметь только одного родителя, множественное наследование в C# не поддерживается (оно поддерживается у интерфейсов).

При попытке обойти это правило таким способом:

```
public class ClassW:ClassY
 {
 }

 public class ClassX:ClassW
 {
 }

 public class ClassY :  ClassX
 {
 }
```

То это не пройдёт, опять компилятор выдаст ошибку:

Error: Circular base class dependency involving ‘InheritanceAndPolymorphism.ClassX’ and ‘InheritanceAndPolymorphism.ClassW’.

Что нужно запомнить: классы не могут наследоваться циклически (1-й от 2-го, 2-й от 3-го 3-й от 1-го), что, в общем-то, логично.

11.1 Операции с объектами

```
ClassB:
public class ClassB
    {
        public int b = 100;
    }

ClassA:

    public class ClassA
    {
        public int a = 100;
    }
/// <summary>
/// Program: используется для запуска кода.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        ClassA classA = new ClassA();
        classA = classB;
        classB = classA;
    }
}
```

Здесь идет попытка приравнять объект от разных классов друг к другу. Опять две ошибки:

–Cannot implicitly convert type ‘InheritanceAndPolymorphism.ClassB’ to ‘InheritanceAndPolymorphism.ClassA’;

–Cannot implicitly convert type ‘InheritanceAndPolymorphism.ClassA’ to ‘InheritanceAndPolymorphism.ClassB’.

Даже несмотря на то, что они имеют одинаковые поля с одинаковыми значениями, даже если бы эти поля имели одинаковые названия, C# работает с типами очень чётко – нельзя приравнять два объекта от двух независимых классов. Однако если бы класс A наследовался от B:

```
public class ClassA:ClassB
 {
     public int a = 100;
 }
```

Error: Cannot implicitly convert type ‘InheritanceAndPolymorphism.ClassB’ to ‘InheritanceAndPolymorphism.ClassA’. An explicit conversion exists (are you missing a cast?)

C# подходит к вопросам типов очень дотошно. Класс A унаследован от B, значит, имеет все его поля и методы – при назначении переменной типа B объекта типа A про-

блем не возникает. Однако в обратную сторону это не работает, в классе B нет полей и методов, которые могут быть в A.

Что нужно запомнить: можно назначить переменной родительского типа объект дочернего, но не наоборот.

Есть шанс обмануть правило:

```
public class ClassB
    {
        public int b = 100;
    }

    public class ClassA:ClassB
    {
        public int a = 100;
    }

    /// <summary>
    /// Program: используется для запуска кода
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassB classB = new ClassB();
            ClassA classA = new ClassA();
            classB=classA;
            classA = (ClassA)classB;
        }
    }
```

Приведение типа здесь сработает, но только потому, что эти классы находятся в наследственных отношениях. Два обособленных непримитивных типа привести друг к другу нельзя.

```
/// <summary>
/// Program: используется для запуска программы.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        int integerA = 10;
        char characterB = 'A';
        integerA = characterB;
        characterB = integerA;
    }
}
```

Error: Cannot implicitly convert type ‘int’ to ‘char’. An explicit conversion exists (are you missing a cast?).

Что нужно запомнить: можно конвертировать char в int. Нельзя конвертировать int в char (причина в том, что диапазон целого числа больше, чем символа).

11.2 Наследование

Наследование делает возможным повторное использование кода – если какой-то класс уже имеет какую-то логику и функции, нам не нужно переписывать всё это заново для создания нового класса, мы можем просто включить старый класс в новый целиком.

> Инкапсуляция – включение в класс объектов другого класса, вопросы доступа к ним, их видимости.
> Полиморфизм: «поли» означает «много», а «морфизм» – «изменение» или «вариативность», таким образом, «полиморфизм» – это свойство одних и тех же объектов и методов принимать разные формы.

Обмен сообщениями – способность одних объектов вызывать методы других объектов, передавая им управление.

11.3 Перегрузка методов

Создадим консольное приложение и класс Overload.cs с тремя методами DisplayOverload, с параметрами, как ниже:

```
public class Overload
  {
      public void DisplayOverload(int a){
          System.Console.WriteLine("DisplayOverload " + a);
      }
      public void DisplayOverload(string a){
          System.Console.WriteLine("DisplayOverload " + a);
      }
      public void DisplayOverload(string a, int b){
          System.Console.WriteLine("DisplayOverload " + a + b);
      }
  }
```

В главном методе Program.cs теперь напишем следующее:

```
class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.DisplayOverload(100);
            overload.DisplayOverload("method overloading");
            overload.DisplayOverload("method overloading", 100);
            Console.ReadKey();
        }
    }
```

Результат будет следующим:

- DisplayOverload 100;
- DisplayOverload method overloading;
- DisplayOverload method overloading100.

Класс Overload содержит три метода, и все они называются DisplayOverload, они различаются только типами параметров. В C# (как и в большинстве других языков) можно создавать методы с одинаковыми именами, но разными параметрами, это и называется

«перегрузка методов». Это значит, что нет нужды запоминать кучу имён методов, которые совершают одинаковые действия с разными типами данных.

Что нужно запомнить: метод идентифицируется не только по имени, но и по его параметрам.

Если запустить следующий код:

- public void DisplayOverload() { };
- public int DisplayOverload() { };

получим ошибку компиляции:

Error: Type ‘InheritanceAndPolymorphism.Overload’ already defines a member called ‘DisplayOverload’ with the same parameter types.

Здесь две функции, которые различаются только по возвращаемому типу, и скомпилировать это нельзя.

Что нужно запомнить: метод не идентифицируется по возвращаемому типу, это не полиморфизм. Если скомпилировать, то это не получится:

```
static void DisplayOverload(int a)  {   }
public void DisplayOverload(int a) {   }
public void DisplayOverload(string a){  }
```

Error: Type ‘InheritanceAndPolymorphism.Overload’ already defines a member called ‘DisplayOverload’ with the same parameter types

Здесь присутствуют два метода, принимающих целое число в качестве аргумента, с той лишь разницей, что один из них помечен как статический.

Что нужно запомнить: модификаторы вроде static также не являются свойствами, идентифицирующими метод.

Если запустить нижеследующий код в надежде, что теперь-то идентификаторы у методов будут разными, то тоже будет ошибка:

```
private void DisplayOverload(int a) {   }

private void DisplayOverload(out int a)
        {
            a = 100;
        }
private void DisplayOverload(ref int a) {   }
```

Error: Cannot define overloaded method ‘DisplayOverload’ because it differs from another method only on ref and out.

Что нужно запомнить: на идентификатор метода оказывают влияние только его имя и параметры (их тип, количество). Модификаторы доступа не влияют. Двух методов с одинаковыми идентификаторами существовать не может.

11.4 Роль ключевого слова «params» в полиморфизме

Параметры могут быть четырёх разных видов:

- переданное значение;
- преданная ссылка;
- параметр для вывода;
- массив параметров.

Подробнее взглянем на четвёртый. Если запустить следующий код, то получим две ошибки:

```
public void DisplayOverload(int a, string a)  {   }

        public void Display(int a)
        {
            string a;
        }
```

Error1: The parameter name ‘a’ is a duplicate;

Error2: A local variable named ‘a’ cannot be declared in this scope because it would give a different meaning to ‘a’, which is already used in a ‘parent or current’ scope to denote something else.

Отсюда следуют вывод: имена параметров должны быть уникальны. Также не могут быть одинаковыми имя параметра метода и имя переменной, созданной в этом же методе.

Теперь запустим следующий код:

```
Overload.cs
public class Overload
    {
        private string name = "Василий";

        public void Display()
        {
            Display2(ref name, ref name);
            System.Console.WriteLine(name);
        }

        private void Display2(ref string x, ref string y)
        {
            System.Console.WriteLine(name);
            x = "Василий 1";
            System.Console.WriteLine(name);
            y = "Василий 2";
            System.Console.WriteLine(name);
            name = "Василий 3";
        }
    }
Program.cs
class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.Display();
            Console.ReadKey();
        }
    }
```

Результат:

- Василий;
- Василий 1;
- Василий 2;
- Василий 3.

Можно передавать одинаковые ссылочные параметры столько раз, сколько захотим. В методе Display строка name имеет значение «Василий». Когда меняется значение «x» на значение «Василий», на самом деле мы меняем значение name, т. к. через параметр «x» передана ссылка именно на него. То же и с y – все эти три переменные ссылаются на одно место в памяти.

Теперь самое интересное:

```
Overload.cs
public class Overload
    {
        public void Display()
        {
            DisplayOverload(100, "Василий", "Петро", "ООР");
            DisplayOverload(200, "Василий");
            DisplayOverload(300);
        }

        private void DisplayOverload(int a, params string[] parameterArray)
        {
            foreach (string str in parameterArray)
               Console.WriteLine(str + " " + a);
        }
    }
Program.cs
class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.Display();
            Console.ReadKey();
        }
    }
```

Результат:

- Василий 100;
- Петро 100;
- ООР 100;
- Василий 200.

Часто может потребоваться передать методу n параметров. В C# такую возможность предоставляет ключевое слово «params».

Важно: это ключевое слово может быть применено только к последнему аргументу метода, так что метод ниже работать не будет, появится ошибка:

private void DisplayOverload(int a, params string[] parameterArray, int b) {}.

В случае DisplayOverload первый аргумент должен быть целым числом, а остальные – сколько угодно много строк или, наоборот, ни одной.

```
public class Overload
{
    public void Display()
    {
        DisplayOverload(100, 200, 300);
        DisplayOverload(200, 100);
        DisplayOverload(200);
    }

    private void DisplayOverload(int a, params int[] parameterArray)
    {
        foreach (var i in parameterArray)
            Console.WriteLine(i + " " + a);
    }

}

//Program.cs тот же, что и в предыдущем примере
```

Вывод:

- 200 100;
- 300 100;
- 100 200.

Важно запомнить: C# достаточно умён, чтобы разделить обычные параметры и массив параметров, даже если они одного типа. Рассмотрим следующие два метода:

```
private void DisplayOverload(int a, params string[][] parameterArray) {   }
private void DisplayOverload(int a, params string[,] parameterArray)  {   }
```

Разница между ними в том, что первый запустится, и такая синтаксическая конструкция будет подразумевать, что в метод будет передаваться n массивов строк. Вторая же выдаст ошибку:

Error: The parameter array must be a single dimensional array.

Запомните: массив параметров должен быть одномерным. Следует упомянуть, что последний аргумент не обязательно заполнять отдельными объектами, можно его использовать, будто это обычный аргумент, принимающий массив, то есть:

```
Overload.cs
public class Overload
    {
        public void Display()
        {
            string[] names = {"Вера", "Надежда", "Любовь"};
            DisplayOverload(3, names);
        }

        private void DisplayOverload(int a, params string[] parameterArray)
        {
            foreach (var s in parameterArray)
                Console.WriteLine(s + " " + a);
        }

    }
Program.cs
class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.Display();
            Console.ReadKey();
        }
    }
```

Результат будет следующим:

- Вера 3;
- Надежда 3;
- Любовь 3.

Однако такой код

```
public class Overload
    {
        public void Display()
        {
           string [] names = {"Вера","Надежда"};
           DisplayOverload(2, names, "Любовь");
        }

        private void DisplayOverload(int a, params string[] parameterArray)
        {
            foreach (var str in parameterArray)
                Console.WriteLine(str + " " + a);
        }

    }
```

вызовет ошибки:

Error: The best overloaded method match for ‘InheritanceAndPolymorphism.Overload.DisplayOverload(int, params string[])’ has some invalid arguments;

Error: Argument 2: cannot convert from ‘string[]’ to ‘string’.

Смешивать передачу отдельными параметрами и одним массивом нельзя.

Теперь рассмотрим поведение следующей программы:

```
Overload.cs
public class Overload
    {
        public void Display()
        {
            int[] numbers = {10, 20, 30};
            DisplayOverload(40, numbers);
            Console.WriteLine(numbers[1]);
        }

        private void DisplayOverload(int a, params int[] parameterArray)
        {
            parameterArray[1] = 1000;
        }

    }
Program.cs
class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.Display();
            Console.ReadKey();
        }
    }
```

После её выполнения получим в консоли 1000. Это происходит из-за того, что при подобном синтаксисе массив передаётся по ссылке. Однако стоит отметить следующую особенность:

```
public class Overload
    {
        public void Display()
        {
```

```
            int number = 102;
            DisplayOverload(200, 1000, number, 200);
            Console.WriteLine(number);
        }

        private void DisplayOverload(int a, params int[] parameterArray)
        {
            parameterArray[1] = 3000;
        }

    }
```

Результатом выполнения такого кода будет 102. Ведь из переданных параметров C# автоматически формирует новый, временный массив.

11.5 Приоритет языка в выборе методов

Предположим, есть такой код:

```
public class Overload
    {
        public void Display()
        {
            DisplayOverload(200);
            DisplayOverload(200, 300);
            DisplayOverload(200, 300, 500, 600);
        }

        private void DisplayOverload(int x, int y)
        {
            Console.WriteLine("The two integers " + x + " " + y);
        }

        private void DisplayOverload(params int[] parameterArray)
        {
            Console.WriteLine("parameterArray");
        }

    }
///Program.cs всё тот же
```

C# рассматривает методы с массивом параметров последними, так что во втором случае будет вызван метод, принимающий два целых числа. В первом и третьем случае будет вызван метод с params, так как ничего кроме него запустить невозможно. Таким образом, на выходе мы получим:

- parameterArray;
- The two integers 200 300;
- parameterArray.

Теперь кое-что интересное. Как вы думаете, каким будет результат выполнения следующей программы?

```
Overload.cs
public class Overload
    {
        public static void Display(params object[] objectParamArray)
        {
            foreach (object obj in objectParamArray)
            {
                Console.Write(obj.GetType().FullName + " ");
            }
            Console.WriteLine();

        }
    }
Program.cs
class Program
    {
        static void Main(string[] args)
        {
            object[] objArray = { 100, "Akhil", 200.300 }; //Массив
            object obj = objArray; //Массив как объект
            Overload.Display(objArray);
            Overload.Display((object)objArray); //Массив, приведённый к объекту
            Overload.Display(obj);
///Почему бы не пойти глубже? :D
            Overload.Display((object[])obj); //Массив как объект, приведённый
к массиву
            Console.ReadKey();

        }
    }
```

Результат:

- System.Int32 System.String System.Double;
- System.Object[] System.Object[] System.Int32 System.String System.Double.

В первом и в четвёртом случаях массив передаётся именно как массив, заменяя собой objectParamArray, а во втором и третьем случаях массив передаётся как единичный объект, из которого создаётся новый массив из одного элемента.

11.6 Ключевые слова «New» и «Override» в C#

Создадим консольное приложение и два класса в нём:

```
public class ClassA
{
    public void AAA()
    {
        Console.WriteLine("ClassA AAA");
    }

    public void BBB()
    {
        Console.WriteLine("ClassA BBB");
    }

    public void CCC()
    {
        Console.WriteLine("ClassA CCC");
    }
```

```
}

public class ClassB : ClassA
{
    public void AAA()
    {
        Console.WriteLine("ClassB AAA");
    }

    public void BBB()
    {
        Console.WriteLine("ClassB BBB");
    }

    public void CCC()
    {
        Console.WriteLine("ClassB CCC");
    }
}
```

Эти классы содержат три метода с попарно одинаковыми именами. Выполним следующий код из Program.cs:

```
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassA x = new ClassA();
        ClassB y=new ClassB();
        ClassA z=new ClassB();

        x.AAA(); x.BBB(); x.CCC();
                Console.WriteLine("");
        y.AAA(); y.BBB();y.CCC();
                Console.WriteLine("");
        z.AAA(); z.BBB(); z.CCC();
    }
}
```

Результат:

- ClassA AAA;
- ClassA BBB;
- ClassA CCC;
- ClassB AAA;
- ClassB BBB;
- ClassB CCC;
- ClassA AAA;
- ClassA BBB;
- ClassA CCC.

Но кроме вывода, есть ещё и три предупреждения от компилятора:

- ‘InheritanceAndPolymorphism.ClassB.AAA()’ hides inherited member;

- ‘InheritanceAndPolymorphism.ClassA.AAA()’. Use the new keyword if hiding was intended;
- ‘InheritanceAndPolymorphism.ClassB.BBB()’ hides inherited member;
- ‘InheritanceAndPolymorphism.ClassA.BBB()’. Use the new keyword if hiding was intended;
- ‘InheritanceAndPolymorphism.ClassB.CCC()’ hides inherited member;
- ‘InheritanceAndPolymorphism.ClassA.CCC()’. Use the new keyword if hiding was intended.

Что нужно запомнить: можно записать в переменную класса-родителя объект наследника, но не наоборот.

ClassA – родитель ClassB. То есть ClassB содержит то, что находится в ClassA, и ещё что-то своё. В этом причина правила, записанного выше: класс-родитель не содержит описания всех необходимых полей и методов класса-наследника, поэтому нельзя использовать ClassA как ClassB.

Что происходит в коде: x и y объявлены и инициализированы одним и тем же типом. Рассмотрим подробнее z. Эта переменная типа ClassB, а её значение – объект типа ClassA, хотя в данном контексте нет никакой разницы, какого типа её значение, вывод всегда будет аналогичен выводу от y. Выбор метода по типу ссылки, а не по типу объекта – это стандартное поведение, когда явно не указан приоритет методов, о чём свидетельствуют warning’и. Как же описать требуемое поведение? Здесь помогут ключевые слова «new» и «override».

11.6.1 Эксперимент

Добавим к двум методам из ClassB ключевые слова «new» и «override» следующим образом:

```
public class ClassB : ClassA
{
    public override void AAA()
    {
        Console.WriteLine("ClassB AAA");
    }

    public new void BBB()
    {
        Console.WriteLine("ClassB BBB");
    }

    public void CCC()
    {
        Console.WriteLine("ClassB CCC");
    }
}
```

Если выполнить Program.cs, то на выходе:

Error: ‘InheritanceAndPolymorphism.ClassB.AAA()’: cannot override inherited member;

‘InheritanceAndPolymorphism.ClassA.AAA()’ because it is not marked virtual, abstract, or override.

Ошибка возникает из-за того, что поля родителя не помечены ключевым словом «virtual.»

Virtual – это модификатор, который обозначает, что мы имеем право вызывать метод из дочернего класса или перезаписывать его.

Добавим virtual ко всем методам ClassA:

```
public class ClassA
    {
        public virtual void AAA()
        {
            Console.WriteLine("ClassA AAA");
        }

        public virtual void BBB()
        {
            Console.WriteLine("ClassA BBB");
        }

        public virtual void CCC()
        {
            Console.WriteLine("ClassA CCC");
        }
    }
```

Результат выполнения Program.cs:

- ClassB AAA;
- ClassB BBB;
- ClassB CCC;
- ClassA AAA;
- ClassA BBB;
- ClassA CCC;
- ClassB AAA;
- ClassA BBB;
- ClassA CCC.

Очевидно, что метод дочернего класса вызвался только там, где стоял модификатор override. В связи с этим делаем вывод:

> override значит, что помеченный метод – новая версия родительского, и должен использоваться вместо него.

И, напротив, new обозначает, что метод, хоть и случайно имеет такое же имя, является абсолютно другим.

Значит, в примере должен выполняться метод родительского класса. Если не написать никакого модификатора, подразумевается именно new.

Разберём подробнее логику C#. Когда вызывается метод какого-то объекта по ссылке, то в первую очередь он смотрит на тип ссылки. Если в этом классе обнаружен модификатор virtual, он начинает искать среди дочерних классов тип объекта, и, если встречает new, запускает последний override метод, который встретил (либо метод типа ссылки). Возможно, это не слишком понятно, обратимся к более сложному примеру.

11.6.2 Эксперимент с тремя классами

```
/// <summary>
/// ClassA, acting as a base class
/// </summary>
public class ClassA
{
    public  void AAA()
    {
        Console.WriteLine("ClassA AAA");
    }

    public virtual void BBB()
    {
        Console.WriteLine("ClassA BBB");
    }

    public virtual void CCC()
    {
        Console.WriteLine("ClassA CCC");
    }
}

/// <summary>
/// Class B, acting as a derived class
/// </summary>
public class ClassB : ClassA
{
    public virtual void AAA()
    {
        Console.WriteLine("ClassB AAA");
    }

    public new void BBB()
    {
        Console.WriteLine("ClassB BBB");
    }

    public override void CCC()
    {
        Console.WriteLine("ClassB CCC");
    }
}

/// <summary>
/// Class C, acting as a derived class
/// </summary>
public class ClassC : ClassB
{
    public override void AAA()
    {
        Console.WriteLine("ClassC AAA");
    }

    public void CCC()
    {
        Console.WriteLine("ClassC CCC");
    }
}

public class Program
```

```
{
    private static void Main(string[] args)
    {
        ClassA y = new ClassB();
        ClassA x = new ClassC();
        ClassB z = new ClassC();

        y.AAA(); y.BBB(); y.CCC();
        Console.WriteLine("");
        x.AAA(); x.BBB(); x.CCC();
        Console.WriteLine("");
        z.AAA(); z.BBB(); z.CCC();

        Console.ReadKey();
    }
}
```

Результатом такого эксперимента станет:

- ClassA AAA;
- ClassA BBB;
- ClassB CCC;
- ClassA AAA;
- ClassA BBB;
- ClassB CCC;
- ClassC AAA;
- ClassB BBB;
- ClassB CCC.

В первом случае имеем дело с типом ссылки ClassA и типом объекта ClassB. Компилятор действует вполне очевидно (рисунок 11.1):

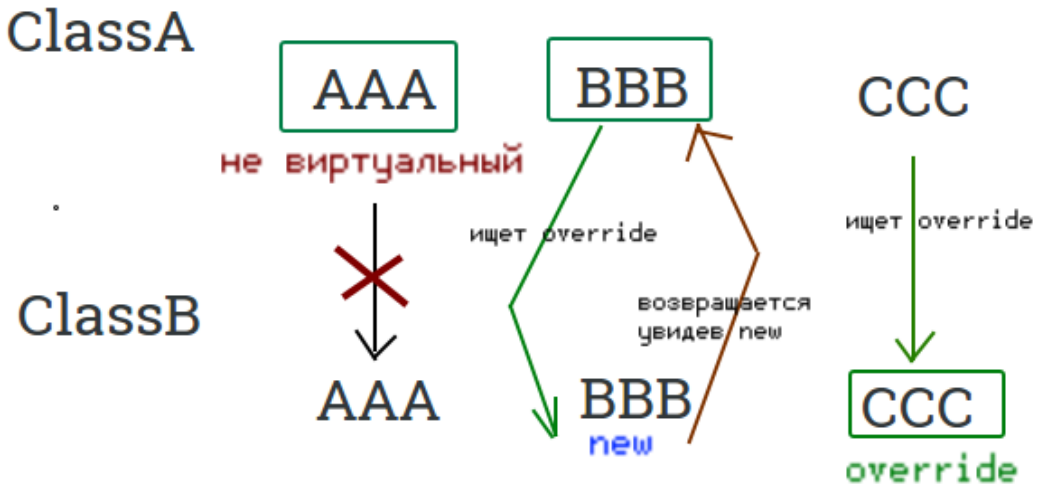


Рисунок 11.1 – Схема использования ключевых new и override

Во втором случае тип объекта уже ClassC. Поскольку он наследуется от ClassA не напрямую, а через ClassB, диаграмма будет уже несколько сложнее (рисунок 11.2).

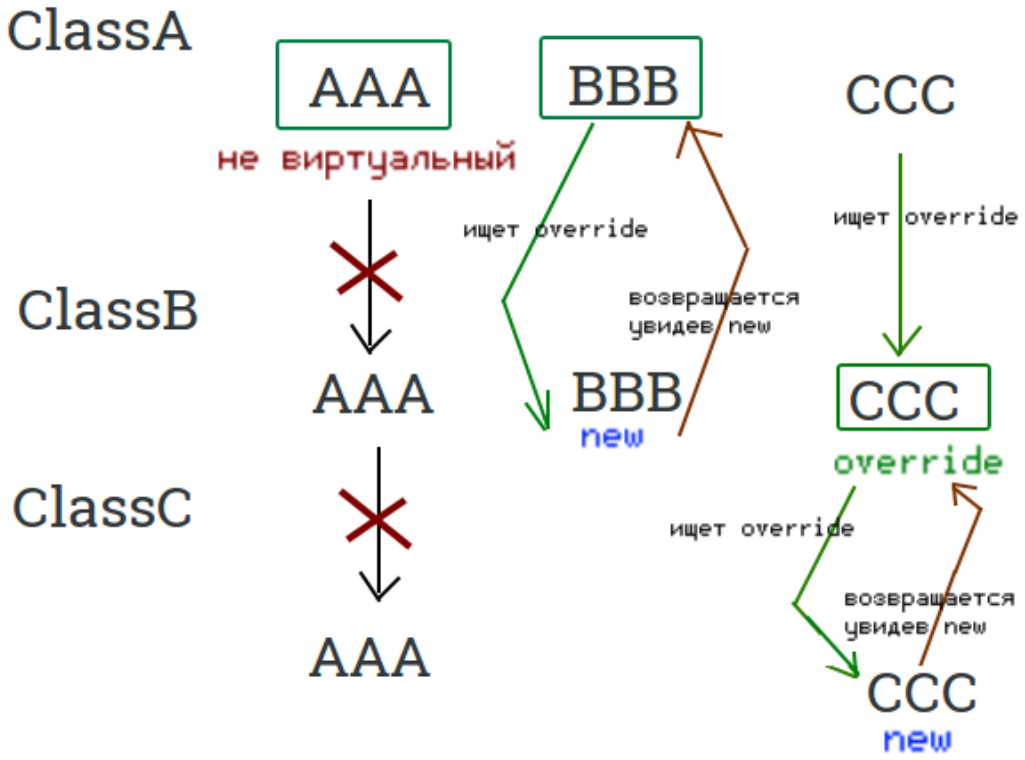


Рисунок 11.2 – Схема использования ключевых new и override при сложной иерархии наследования

В третьем случае имеем дело снова с двумя классами, ClassA просто игнорируем. Если учесть это, то вывод будет очевиден (рисунок 11.3).

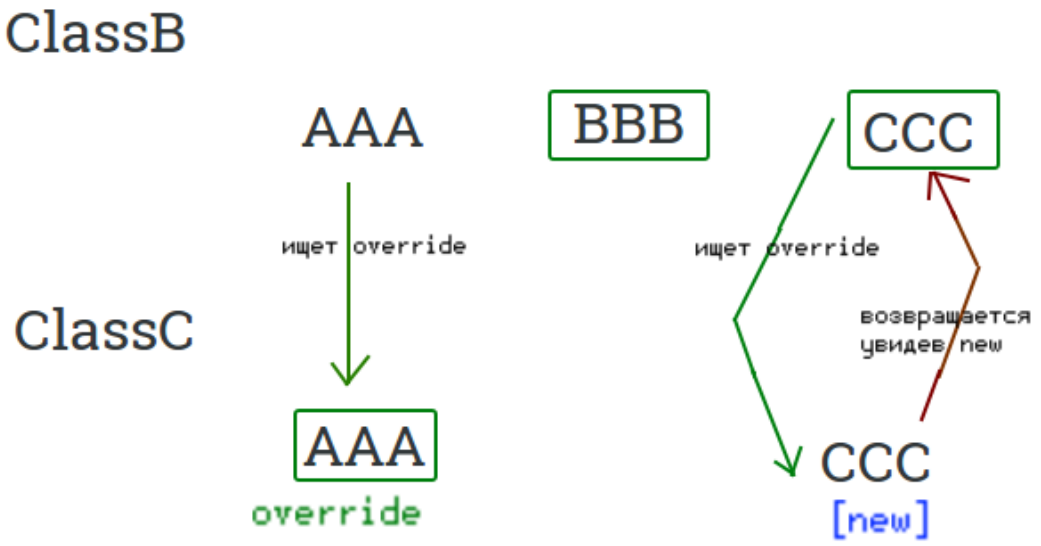


Рисунок 11.3 – Схема использования ключевых new и override без базового класса A

Важным замечанием будет, что следующий код:

```
internal class A
{
    public virtual void X()
    {
    }
}

internal class B : A
{
    public new void X()
    {
    }
}

internal class C : B
{
    public override void X()
    {
    }
}
```

Выдаст ошибку:

Error: ‘InheritanceAndPolymorphism.C.X()’: cannot override inherited member ‘InheritanceAndPolymorphism.B.X()’ because it is not marked virtual, abstract, or override.

Ведь из-за того, что в B метод помечен как new, он не наследует свойство virtual, а значит не может быть перезаписан с помощью override в C. Правильным вариантом было бы добавить к описанию метода в B ключевое слово «virtual» или изменить в C override на new, в зависимости от требуемого поведения.

11.6.3 Ключевое слово «base»

Рассмотрим возможность вызова методов родительского класса из дочернего. Простой пример:

```
/// <summary>
    /// Class A
    /// </summary>
    public class ClassA
    {
        public virtual void XXX()
        {
            Console.WriteLine("ClassA XXX");
        }
    }

    /// <summary>
    /// ClassB
    /// </summary>
    public class ClassB:ClassA
    {
        public override void XXX()
        {
            base.XXX();
            Console.WriteLine("ClassB XXX");
        }
    }
```

```
    /// <summary>
    /// Class C
    /// </summary>
    public class ClassC : ClassB
    {
        public override void XXX()
        {
            base.XXX();
            Console.WriteLine("ClassC XXX");
        }
    }

   /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassA a = new ClassB();
            a.XXX();
            Console.WriteLine("");
            ClassB b = new ClassC();
            b.XXX();
            Console.ReadKey();
        }
    }
```

Результат:

- ClassA XXX;
- ClassB XXX;
- ClassA XXX;
- ClassB XXX;
- ClassC XXX.

В первом случае выполняется метод ClassB, который через base вызывает метод из ClassA. Во втором – XXX() из ClassC, который обращается к ClassB, а тот, в свою очередь, к ClassA.

Немного рекурсии:

```
/// <summary>
    /// Class A
    /// </summary>
    public class ClassA
    {
        public virtual void XXX()
        {
            Console.WriteLine("ClassA XXX");
        }
    }

    /// <summary>
    /// ClassB
    /// </summary>
    public class ClassB:ClassA
    {
        public override void XXX()
        {
```

```
            ((ClassA)this).XXX();
            Console.WriteLine("ClassB XXX");
        }
    }


    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassA a = new ClassB();
            a.XXX();

        }
    }
```

В этом примере вызов ClassB.XXX() всегда будет приводить к созданию нового объекта типа ClassB в ссылке ClassA. Очевидно, что по такой ссылке снова будет вызван ClassB.XXX() и т. д. В данном случае выводом будет ошибка:

Error: {Cannot evaluate expression because the current thread is in a stack overflow state.}.

Заключение

В C# можно записать в переменную класса-родителя объект наследника, но не наоборот.

Модификатор override используется, чтобы указать на то, что должен вызваться метод именно дочернего класса. Чтобы использовать модификаторы override и new, метод родительского класса должен быть помечен ключевым словом «virtual».

Когда вызывается метод какого-то объекта по ссылке, то C# в первую очередь смотрит на тип ссылки. Если в этом классе обнаружен модификатор virtual, он начинает искать среди дочерних классов тип объекта и (если встречает new) запускает последний override метод, который встретил (либо метод типа ссылки). Как сын получается похожим на отца, наследует его черты, так и дочерний класс имеет параметры родительского.

Ничто не может помешать создать в дочернем классе такой же метод, как и в родительском.

Методы дочерних классов имеют приоритет при выполнении.

Ключевое слово «base» может быть использовано для обращения к методам класса-предка.

Наследование не работает в обратном направлении.

Кроме конструкторов и деструкторов, дочерний класс получает от родителя абсолютно всё.

Классы не могут быть унаследованы от встроенных классов вроде system.valuetype, system.enum, system.delegate, system.array и т. д. Класс может иметь только одного родителя, множественное наследование классов в C# не поддерживается.

Классы не могут наследоваться циклически (1-й от 2-го, 2-й от 3-го 3-й от 1-го), это невозможно чисто логически.

Можно назначить переменной родительского типа объект дочернего, но не наоборот.

Можно конвертировать char в int. Нельзя конвертировать int в char (причина в том, что диапазон целого числа больше, чем символа).

Рассмотрена перегрузка методов, особенности компиляции с ней связанные и буквально все возможные случаи использования ключевого слова «params».

Метод идентифицируется не только по имени, но и по его параметрам.

Метод не идентифицируется по возвращаемому типу.

Модификаторы вроде static также не являются свойствами, идентифицирующими метод.

На идентификатор метода оказывают влияние только его имя и параметры (их тип, количество). Модификаторы доступа не влияют. Двух методов с одинаковыми идентификаторами существовать не может.

Имена параметров должны быть уникальны. Также не могут быть одинаковыми имя параметра метода и имя переменной, созданной в этом же методе.

Ключевое слово «params» может быть применено только к последнему аргументу метода.

C# достаточно умён, чтобы разделить обычные параметры и массив параметров, даже если они одного типа.

Массив параметров должен быть одномерным.

Вопросы для самоконтроля

1. Что называется полиморфизмом в ООП?
2. Перегрузка наследованных методов.
3. Роль ключевого слова «params» в полиморфизме.
4. Как работают и что позволяют реализовать ключевые слова «new» и «override»?
5. Использование ключевого слова «base» в дочерних классах.
6. Приоритет выбора методов при перегрузке.
7. Передача массива параметров в методы.

ГЛАВА 12
СТРУКТУРЫ

Структуры похожи на классы, их отличие от классов заключается в том, что структуры располагаются не в куче, а в стеке и являются, в отличие от классов, не ссылочными, а значениями типа. Использование структур повышает эффективность работы программы, но вносит ряд ограничений.

> Структура – это пользовательский тип, размещенный в стеке и доступный по значению.

К ограничениям следует отнести то, что структуры не реализуют наследование. Объявление структуры и класса схоже, но при объявлении структуры используется ключевое слово «struct» вместо «class».

```
struct имя_структуры
{
// конструкторы
// поля
// свойства
// методы
// события
// индексаторы
// и т.д.
}
```

В структурах отсутствуют деструкторы и конструкторы без параметров. Конструктор без параметров существует у всех структур по умолчанию и перекрытию не подлежит.

У структур отсутствуют абстрактные (abstract), виртуальные (virtual) и защищенные (protected) члены, они бесполезны из-за того, что структуры не поддерживают наследование.

Объекты структур создаются (так же как и экземпляры классов) при помощи ключевого слова «new», но использование ключевого слова «new» не обязательно, т. к. new вызывает конструктор по умолчанию, которого может и не быть.

Общий вид структуры:

```
struct S
{
  public string Name;
  public int Age;

  public S(string aName, byte aAge)
  {
    this.Name = aName;
    this.Age = aAge;
  }

  public void PrintStudent()
  {
    Console.WriteLine("Имя: {0}, возраст: {1}",Name,Age);
  }
}

static void Main()
{
  S oS1 = new S("Решетов", 19);
  oS1.PrintStudent();
```

```
  S oS2 = new S("Тоичкин",20);
  oS2.PrintStudent();

  // отличие структур от классов
  oS1 = oS2;
  oS2.Name = "Николаенко";
  oS2.Age = 38;
  oS1.PrintStudent();
  oS2.PrintStudent();

Console.ReadLine();
}
```

Разница между структурой и классом в том, что во время выполнения операции присваивания oS1 = oS2 для структуры создается копия структуры, а будь S не структурой, а классом, произошло бы присвоение ссылки, оба экземпляра oS1 и oS2 ссылались бы на один и тот же экземпляр.

Использование структур связано исключительно с повышением производительности программ. Функционально структуры уступают классам, но за счет того, что структурами можно оперировать непосредственно, а не по ссылке, не требуется переменная ссылочного типа, а это приводит к существенной экономии оперативной памяти.

Чаще всего структуры используются для хранения данных, не требующих наследования и обращения по ссылке.

Вопросы для самоконтроля

1. В чем различие структур и классов?
2. Как объявляются структуры?
3. Использование структур и повышение производительности программ.

ГЛАВА 13
ПЕРЕЧИСЛЕНИЯ

> Перечисление – это определяемый пользователем логически связанный целочисленный набор констант.

Перечисления характерны тем, что при их объявлении указывается определенный набор значений (имеющих понятные пользователю имена), которые может принимать экземпляр перечисления. При попытке присвоить экземпляру перечисления значение, которое не входит в область допустимых значений, компилятор сгенерирует ошибку.

Использование перечислений приводит к тому, что код становится читаемым, а это экономит время разработки и сопровождения программ. Выделяют несколько преимуществ от использования перечислений вместо простых чисел:

– гарантируют использование только легитимных значений из списка допустимых значений;

– упрощают кодирование благодаря возможностям IntelliSense;

– обеспечивают упрощение понимание кода.

Для объявления данных используется ключевое слово «enum».

```
enum имя {список_перечисления};
```

Где имя – это имя типа перечисления, а список_перечисления – список идентификаторов, разделяемый запятыми.

```
enum Kurs
    {first, second=2, third, fouth, fifth}
namespace Myenum
{
    class Program
    {
        public static void Main(string[] args)
        {
            Kurs k = Kurs.first;
            Console.WriteLine(k);
            Console.ReadKey(true);
        }
    }
}
```

Вопросы для самоконтроля

1. Дайте определение понятию перечисление.
2. Чем перечисления отличаются от обычных переменных или массивов?
3. В чем преимущества использования перечислений?

ГЛАВА 14
АБСТРАКТНЫЕ КЛАССЫ

В прошлых лекциях был рассмотрен полиморфизм (а также его нюансы на практике) и наследование. В этой мы поговорим о самой захватывающей части ООП-парадигмы – об абстрактных классах. В целом концепция абстрактных классов в C# ничем не отличается от таковой в других языках, но в C# работать с ней приходится несколько иначе.

> Модификатор abstract указывает, что реализация сущности с данным модификатором является неполной или отсутствует.

Модификатор abstract может использоваться с классами, методами, свойствами, индексаторами и событиями. Модификатор abstract в объявлении класса указывает, что класс предназначен только для использования в качестве базового класса для других классов. Члены, помеченные как абстрактные или включенные в абстрактный класс, должны быть реализованы с помощью классов, производных от абстрактных классов.

14.1 Абстрактные классы в действии

Попробуем создать абстрактный класс:

```
using System;

namespace InheritanceAndPolymorphism
{
    public abstract class ClassA
    {
    }
    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassA classA = new ClassA();
            Console.ReadKey();
        }
    }
}
Compile time error: Cannot create an instance of the abstract class or
interface 'InheritanceAndPolymorphism.ClassA'
```

> Мы не можем создать экземпляр абстрактного класса с помощью ключевого слова «new».

14.2 Описание методов в абстрактном классе

Попробуем добавить в наш абстрактный класс немного кода:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassA classA = new ClassA();
        Console.ReadKey();
    }
}
```

С кодом класса никаких проблем нет, но скомпилировать снова не получается, потому что нельзя создавать экземпляры абстрактных классов.

Использование абстрактного класса в качестве базового

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA
/// </summary>
public class ClassB:ClassA
{
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

Теперь всё спокойно компилируется.

Мы можем унаследовать обычный класс от абстрактного.
Мы можем создать экземпляр обычного класса, унаследованного от абстрактного, с помощью ключевого слова «new».

14.3 Декларация методов в абстрактном классе

Попробуем сделать вот так:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
    public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

Получаем ошибку компиляции:

```
Compile time error: 'InheritanceAndPolymorphism.ClassA.YYY()'
must declare a body because it is not marked abstract, extern, or partial
```

Дело в том, что если мы объявляем метод в абстрактном классе и при этом хотим, чтобы его конкретное поведение было определено в производных классах, то к такому методу мы должны также добавить ключевое слово «abstract». Добавим его:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
```

```
   abstract public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

…и снова получим ошибку компиляции:

```
Compiler error: 'InheritanceAndPolymorphism.ClassB' does not implement
inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY()'
```

> Если мы хотим объявить метод в абстрактном классе, но не реализовывать его, к методу нужно добавить ключевое слово «abstract».
> Если мы объявляем абстрактный метод в абстрактном классе, то этот метод должен реализовываться в неабстрактных наследниках этого класса.

14.4 Реализация абстрактного метода в производном классе

Давайте тогда попробуем реализовать метод YYY() в классе ClassB:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
   abstract public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
    public void YYY()
    {
    }
}
/// <summary>
```

```
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

Получим сразу две ошибки компиляции:

```
Compile time error: 'InheritanceAndPolymorphism.ClassB' does not implement
inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY()'
Compile time warning: 'InheritanceAndPolymorphism.ClassB.YYY()' hides
inherited member 'InheritanceAndPolymorphism.ClassA.YYY()'.
```

Дело в том, что в C# нужно явно объявить, что мы реализуем абстрактный метод класса-родителя с помощью ключевого слова «override»:

```
/// <summary>
    /// Abstract class ClassA
    /// </summary>
    public abstract class ClassA
    {
        public int a;
        public void XXX()
        {

        }
       abstract public void YYY();
    }
    /// <summary>
    /// Derived class.
    /// Class derived from abstract class ClassA.
    /// </summary>
    public class ClassB:ClassA
    {
        public override void YYY()
        {

        }
    }
    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassB classB = new ClassB();
            Console.ReadKey();
        }
    }
```

Нет никаких ошибок!

> Абстрактный метод базового класса и метод с override класса-наследника должны быть одинаковы.

Это значит, что мы не можем менять тип возвращаемого значения или аргументы, которые передаются в метод. Например, если мы напишем такое:

```
/// <summary>
  /// Abstract class ClassA
  /// </summary>
  public abstract class ClassA
  {
      public int a;
      public void XXX()
      {
      }
     abstract public void YYY();
  }
  /// <summary>
  /// Derived class.
  /// Class derived from abstract class ClassA.
  /// </summary>
  public class ClassB:ClassA
  {
      public override int YYY()
      {
      }
  }
  /// <summary>
  /// Program: used to execute the method.
  /// Contains Main method.
  /// </summary>
  public class Program
  {
      private static void Main(string[] args)
      {
          ClassB classB = new ClassB();
          Console.ReadKey();
      }
  }
```

Тогда в консоли увидим следующую ошибку:

```
Compile time error: 'InheritanceAndPolymorphism.ClassB.YYY()': return type
must be 'void'
to match overridden member 'InheritanceAndPolymorphism.ClassA.YYY()'
```

14.5 Инициализация полей в абстрактных классах

В примерах выше переменная int a будет обладать значением по умолчанию (0). Мы можем изменить его на нужное нам прямо в абстрактном классе – с этим не связано никаких особенностей.

14.6 Абстрактные методы в неабстрактных классах

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public class ClassA
{
    public int a;
    public void XXX()
    {
    }
   abstract public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
    public override void YYY()
    {
    }
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

Такой код не скомпилируется:

```
Compiler error: 'InheritanceAndPolymorphism.ClassA.YYY()' is abstract
but it is contained in non-abstract class 'InheritanceAndPolymorphism.ClassA'
```

Абстрактные методы могут быть объявлены только в абстрактных классах.

14.7 Вызов абстрактного метода родителя

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
   abstract public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
```

```
/// </summary>
public class ClassB:ClassA
{
    public override void YYY()
    {
         base.YYY();
    }
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB classB = new ClassB();
        Console.ReadKey();
    }
}
```

Вывод:

```
Compile time error : Cannot call an abstract base member:
'InheritanceAndPolymorphism.ClassA.YYY()'
```

Разумеется, мы не можем исполнить код, которого не существует.

14.8 Абстрактный класс, который наследуется от другого абстрактного класса

```
/// <summary>
/// Base class ClassA
/// </summary>
public class ClassA
{
    public virtual void XXX()
    {
        Console.WriteLine("ClassA XXX");
    }
}

/// <summary>
/// Derived abstract class.
/// Class derived from base class ClassA.
/// </summary>
public abstract class ClassB:ClassA
{
    public new abstract void XXX();
}

public class ClassC:ClassB
{
    public override void XXX()
    {
        System.Console.WriteLine("ClassC XXX");
    }
}

/// <summary>
```

```
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassA classA = new ClassC();
        ClassB classB = new ClassC();
        classA.XXX(); classB.XXX();
    }
}
Вывод:
ClassA XXX
ClassC XXX
```

14.9 Может ли абстрактный класс быть sealed

Проверим:

```
/// <summary>
/// sealed abstract class ClassA
/// </summary>
public sealed abstract class ClassA
{
    public abstract void XXX()
    {
        Console.WriteLine("ClassA XXX");
    }
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
    }
}
```

И получим ошибку:

```
Compile time error: 'InheritanceAndPolymorphism.ClassA':
an abstract class cannot be sealed or static
```

Разумеется, абстрактный класс не может быть sealed, т. к. он для того и создан, чтобы от него создавались производные классы.

Заключение

Абстрактный класс не может иметь модификатор sealed;
Абстрактный класс не может иметь модификатор static;
Мы не можем создать экземпляр абстрактного класса с помощью ключевого слова «new»;
Мы можем унаследовать обычный класс от абстрактного;

Мы можем создать экземпляр обычного класса, унаследованного от абстрактного, с помощью ключевого слова «new»;
Если мы хотим объявить метод в абстрактном классе, но не реализовывать его, к методу нужно добавить ключевое слово «abstract»;
Если мы объявляем абстрактный метод в абстрактном классе, то этот метод должен реализовываться в неабстрактных наследниках этого класса;
Абстрактные методы могут быть объявлены только в абстрактных классах;
Абстрактный класс не может иметь модификатор sealed;
Абстрактный класс не может иметь модификатор static.

Вопросы для самоконтроля

1. Объявление абстрактных классов.
2. Как описываются методы в абстракных класах?
3. Как происходит реализация абстрактных методов в производных классах?
4. Особенности инициализации полей в абстрактных классах.
5. Возможно ли реализовывать абстрактные методы в неабстрактных классах?
6. Можно ли запечатывать абстрактный класс?

ГЛАВА 15
ИНТЕРФЕЙСЫ

Интерфейсы – это еще один инструмент реализации полиморфизма в C#. Книжные определения интерфейсов звучат следующим образом:

> Интерфейс представляет собой набор методов (свойств, событий, индексаторов), реализацию которых должен обеспечить класс, который реализует интерфейс.
> Интерфейс – это именованный набор сигнатур методов;
> Интерфейс – это именованный набор абстрактных членов.

Однако такое определение не полностью отражает его смысл. Вернёмся к определению позднее. Важно понимать, что интерфейс взаимодействия и интерфейс – это не одно и то же.

> Интерфейс взаимодействия – это набор публичных методов класса.

Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс – множество методов.

Интерфейс содержит только сигнатуры своих членов без их реализации, это одно из интерфейса от абстрактного класса. Также интерфейс не содержит конструкторы, поля, константы, статические члены. Создать экземпляр интерфейса невозможно.

Интерфейс представляет собой именованный набор абстрактных членов. Разница между абстрактными классами и интерфейсами малопонятна для начинающих программистов.

Члены объекта, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. В качестве примера можно привести коллекции. Как известно, коллекции – это не что иное, как «сахарные» массивы, т. е. в основе коллекции лежит массив, но он «обернут» методами, упрощающими использование этих массивов и наделяющие их определенным функционалом, превращая их из массивов в коллекции. Одним из таких «сахарных» методов является метод Sort, позволяющий, не реализуя самих алгоритмов сортировки, выполнить сортировку коллекции.

Давайте поставим себя на место разработчиков классов коллекций. Перед нами (разработчиками) стоит задача: написать метод Sort таким, чтобы он позволял сортировать любую коллекцию. И тут возникает трудность следующего характера: по какому из полей отсортировать коллекцию элементов? Ведь, как известно, элементами коллекции могут быть экземпляры объектов, а они, в свою очередь, могут состоять из большого количества полей. И перед нами (разработчиками класса коллекции) стоит неразрешимая задача: как организовать это сравнение?

Решением данной задачи может быть отложенная реализация метода сравнения двух элементов коллекции, и его реализацию следует возложить на тот класс, который будет использовать нашу коллекцию.

Стандартные коллекции, например ArrayList, для того, чтобы выполнить сортировку, обязуют при помощи интерфейсов реализовать метод CompareTo. Данный пример демонстрирует один из способов использования интерфейсов.

Интерфейсы объявляются с помощью ключевого слова «interface». Ниже приведена упрощенная форма объявления интерфейса:

```
interface имя{
    возвращаемый_тип имя_метода_1 (список_параметров);
    возвращаемый_тип имя_метода_2 (список_параметров);
    // …
    возвращаемый_тип имя_метода_N (список_параметров);
}
```

где имя – это конкретное имя интерфейса.

В объявлении методов интерфейса используются только их возвращаемый_тип и сигнатура, и тут они являются абстрактными методами. Так как интерфейс не содержит никакой реализации, следовательно, все методы интерфейса обязаны быть реализованы в каждом классе, который реализует этот интерфейс. В интерфейсе все методы неявно считаются открытыми, поэтому доступ к ним явно не указывается. В интерфейсах можно объявлять свойства, индексаторы и события. Интерфейсы не содержат члены данных; ни один из членов интерфейса не может быть объявлен как static. Логично, что в интерфейсах нельзя объявлять конструкторы, деструкторы или операторные методы.

Реализация интерфейса подобна наследованию классов – достаточно указать имя интерфейса после имени класса:

```
class имя_класса : имя_интерфейса {
    // тело класса
}
```

где имя_интерфейса – это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

Рассмотрим пример использования интерфейсов.

```
class A
{
   public string Name { get; set; }
   public void GetName()
      {
         WL(this.Name);
      }
   private int Age { get; set; }
}
Main()
{
   A oA = new A();
   oA.Name = "Имя";
   oA.GetName();
}
```

В классе A реализованы следующие члены: публичное строковое свойство Name, публичный метод GetName и закрытое свойство Age. В Main-методе происходит вызов и использование этих публичных членов. Так вот, всё, что можно вызвать у экземпляра класса, называется интерфейсом взаимодействия.

Интерфейсы помогают «заставить» класс иметь интерфейсы взаимодействия. Для этого необходимо определить сам интерфейс и заставить класс его реализовать (через «:» (двоеточие) после имени класса указать имя интерфейса, как в следующем коде).

```
class A : IMyInterface
{

}
```

Отметим, что наследование и реализация интерфейсов – это не одно и то же. Говорить: «Наследуем интерфейс», – неверно. Правильно говорить: «Реализация интерфейса». Но у этих двух процессов есть и общее, что и позволяет сказать: «Наследование интерфейсов».

Дело в том, что во время наследования одним классом другого дочерний класс получает не только методы, но и интерфейс взаимодействия.

Класс, реализующий интерфейс, обязан иметь все публичные члены, объявленные в интерфейсе, как в примере ниже, где класс A, реализующий интерфейс IMyInterface, обязан реализовать и реализует такие публичные члены, как свойство Name и метод WriteName.

```
interface ImyInterface
{
   string Name { get; set; }
   string WriteName(string a);
}
class A : ImyInterface
{
   public string Name { get; set; }
   public string WriteName(string a)
   {
      WL(this.name);
      return this.name;
   }
   public void NewMethod()
   {
      WL("Новый метод не из интерфейса");
   }
}
```

После того, как класс реализовал все члены интерфейса, появляется возможность выполнять приведение этого экземпляра к интерфейсу и использовать его (как показано в примере ниже):

```
Main()
{
   ImyInerface oA = new A() //Upcast
   oA.Name = "Имя";
   oA.WriteName();
   oA.NewMethod(); //Выполнить не получится
}
```

При приведении экземпляра к интерфейсу возможно использование только тех членов, которые определены и в интерфейсе, и в классе.

Все члены, определённые в классе, «исчезли». Под исчезновением следует понимать, что они существуют, но только как скрытые (private). И даже существует возможность вызвать их, например из тех методов, которые определены в интерфейсе, т. е. можно организовать вызов скрытого метода из открытого, тем самым реализовав принцип инкапсуляции метода. В рассмотренном выше примере метод NewMethod, реализованный в классе и не объявленный в интерфейсе, «исчезнет», но если добавить в метод WriteName строку вызова метода NewMethod, он по-прежнему может быть выполнен. Следующий пример демонстрирует такую возможность.

```
interface IA
{
   void Do();
```

```
}
class A : IA
{
   public void Do()
   {
      M();
   }
   public void M()
   {
      WL("Метод");
   }
}
```

Метод M() не определён в интерфейсе. При приведении его к интерфейсу его вызов напрямую невозможен, но его можно вызвать через метод Do(), который определён в интерфейсе.

```
Main()
{
   IA oA = new A();
   oA.Do();
}
```

Данная ситуация – это одна из форм инкапсуляции. Публичные методы становятся private.

C# допускает множественную реализацию интерфейсов.

```
interface IA1
{
   void Do();
}
interface IA2
{
   void Do();
}
class A : IA1, IA2
{
   public void Do()
   {
      WL("Doing…");
   }
}
```

В примере выше метод Do() имеет одну и ту же реализацию для обоих интерфейсов.

Каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, следовательно, тем самым поддерживать множество поведений. Существует возможность реализовать метод Do() для каждого интерфейса отдельно, что называется явной и неявной реализацией интерфейса.

```
class A : IA1, IA2
{
   void IA1.Do()
   {
      WL("Реализация... IA1");
   }
   void IA2.Do()
```

```
    {
       WL("Реализация... IA2");
    }
}
Main()
{
    IA1 oA1 = new A();
    IA2 oA2 = new A();
    oA1.Do(); //выведет: Реализация…IA1
    oA2.Do(); //выведет: Реализация…IA 2
}
```

Если не приводить экземпляр класса к интерфейсу, то данный метод вызвать не удастся.

```
Main()
{
    A oA = new A();
    oA.Do(); // Ошибка
}
```

Из выше сказанного можно сделать вывод о том, что:

- интерфейсы предусмотрены для формирования интерфейса взаимодействия.

Следующий пример использования интерфейсов иллюстрирует возможности интерфейсов в случаях, когда необходимо реализовать взаимозаменяемость классов.

```
interface IWorker
{
    void Work(int a);
}
class WorkerStandart
{
    public IWorker Worker { get; set; }
    public void Work(int a)
    {
       a+=10;
       Worker.Work(a);
    }
}
class WorkerEconomist : IWorker
{
    public void Work(int a)
    {
       WL("ноль работы" + a);
    }
}
class WorkerMinistr : IWorker
{
    public void Work(int a)
    {
       WL("не работает, а деньги есть");
       a+=1000;
    }
}
class WorkerIT : IWorker
{
```

```
   public void Work(int a)
   {
      a+=10000;
      WL("работает за " + a);
   }
}
```

У класса WorkerStandart есть свойство, тип которого соответствует интерфейсу IWorker. Кроме того, интерфейс IWorker одновременно реализует несколько классов. Это позволяет непосредственно в коде присвоить этому свойству любой класс при условии, что он реализует этот интерфейс.

```
Main()
{
   WorkerStandart stWorker = new WorkerStandart();
   stWorker.Worker = new WorkerEconomist();
   stWorker.Work(100);
   stWorker. Worker = new WorkerMinistr();
   stWorker.Work(100);
   stWorker.Worker = new WorkerIT();
   stWorker.Work(100);
}
```

Этим примером демонстрируется возможность менять логику экземпляра, содержащегося в данном свойстве. Интерфейс в данном случае – корпус ручки, в которую можно вложить любого цвета стержень при условии, что размеры подходящие.

Кроме того, интерфейсы позволяют ограничить классы определённым кругом задач. Например, когда не все методы класса необходимы. Это достигается использованием апкаста (UpCast) экземпляра к интерфейсу.

Подведем итог и сформулируем окончательное понятие интерфейса исходя из рассмотренного выше:

> Интерфейс – это аспект объектно-ориентированного подхода, который предназначен для формирования у экземпляров интерфейса взаимодействия.

15.1 Интерфейсные ссылки

C# допускает объявление переменных ссылочного интерфейсного типа (переменные-ссылки на интерфейс). Эти переменные могут ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки выполняется его вариант, реализованный в классе данного объекта. Это аналогично применению ссылки на базовый класс для доступа к объекту производного класса.

Переменной ссылке на интерфейс доступны только методы, объявленные в ее интерфейсе. Поэтому интерфейсную ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

```
using System;

namespace ConsoleApplication1
{
    public interface IMyInterface
    {
        void PrintName();
```

```
        void PrintFamily();
        void PrintAge();
    }

    class A : IMyInterface
    {
        string Name, Family;
        int Age;

        public A(string aName, string aFamily, int aAge)
        {
            this.Name = aName;
            this.Family = aFamily;
            this.Age = aAge;
        }

        // Реализуем интерфейс
        public void PrintName()
        {
            Console.WriteLine("Имя: " + Name);
        }

        public void PrintFamily()
        {
            Console.WriteLine("Фамилия: " + Family);
        }

        public void PrintAge()
        {
            Console.WriteLine(«Возраст: « + Age);
        }

        // Собственный метод класса
        public void GetInfo()
        {
            Console.WriteLine(Name + " " + Family + " " + Age);
        }
    }

    class Program
    {
        static void Main()
        {
            A oA = new A(Name: "Ваня", Family: "Обамович", Age: 19);

            // Создадим ссылку на интерфейс
            IMyInterface obj;
            //Используем ссылку на объект oA
            obj = oA;
            obj.PrintName();
            obj.PrintFamily();
            obj.PrintAge();
            // Вызов собственного метода недоступен, т.к. он стал private:
            // obj.GetInfo();
            Console.ReadLine();
        }
    }
}
```

15.2 Ключевое слово «as»

Ключевое слово «as» позволяет проверить, поддерживает ли данный тип тот или иной интерфейс. В случае удачной интерпретации возвращается ссылка на интерфейс, в противном случае – ссылка null. Значит, в коде необходимо предусмотреть проверку на null.

```
В предыдущем примере, в методе Main() можно добавить следующую проверку:
IMyInterface obj = oA as IMyInterface;
if (obj != null)
   Console.WriteLine("A поддерживает интерфейс IMyInterface");
else
   Console.WriteLine("A не поддерживает интерфейс IMyInterface");
```

15.3 Ключевое слово «is»

Еще одним способом проверить, был ли реализован интерфейс, можно с помощью ключевого слова «is». Если запрашиваемый объект несовместим с указанным интерфейсом, будет возвращено значение false, иначе – возможно вызывать члены этого интерфейса без использования try/catch.

```
if (oA is IMyInterface)
  Console.WriteLine("Тип A поддерживает интерфейс IMyInterface");
else
  Console.WriteLine(«A не поддерживает интерфейс IMyInterface»);
```

15.4 Свойства в интерфейсах

Общий вид объявления свойств в интерфейсе:

```
// Интерфейсное свойство
тип имя{
  get;
  set;
}
```

Такое объявление напоминает объявление автоматически реализуемого свойства в классе, но при объявлении в интерфейсе свойство не становится автоматически реализуемым, его реализация предоставляется каждому реализующему классу. При объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для геттеров и сеттеров. Например, set не может быть указан в интерфейсе как private.

Пример:

```
using System;

namespace ConsoleApplication1
{
    interface IMyInterface
    {
        string Name
        {
            get;
            set;
        }
```

```
    }

    class A : IMyInterface
    {
        string name;

        public string Name
        {
            set
            {
                name = value;
            }

            get
            {
                return name;
            }
        }
    }

    class Program
    {
        static void Main()
        {
            A oA = new A();
            oA.Name = "Владимир";

            Console.ReadLine();
        }
    }
}
```

В данном примере в классе A реализуется свойство интерфейса IMyInterface.

15.5 Индексаторы в интерфейсах

Интерфейс может содержать индексаторы. Общая форма объявления интерфейсного индексатора выглядит следующим образом:

```
// Интерфейсный индексатор
тип_элемента this[int индекс]{
  get;
  set;
}
```

Реализуем индексатор:

```
using System;

interface IMyInterface
{
  string Name
    {
      get;
      set;
    }

  string this[int index]
```

```
  {
    get;
    set;
  }
}

class A : IMyInterface
{
  string name;

  public string Name
  {
    set {name = value;}
    get {return name;}
  }

  public string this[int index]
  {
    set {name = value; }
    get {return name; }
  }
}

class Program
{
  static void Main()
  {
    A oA = new A();
    oA.Name = "Фельдьман";
    oA[5] = "Лапко";
    oA[10] = "Башков";

    Console.ReadLine();
  }
}
```

15.6 Наследование интерфейсов

Интерфейсы могут наследовать друг друга подобно классам, синтаксически это не отличается от наследования классов. Если класс реализует интерфейс, который, в свою очередь, наследует еще один интерфейс, то необходимо реализовать все члены, определенные в цепочке наследования интерфейсов. В теме о пользовательских коллекциях таким примером будет интерфейс IEnumerator<T>.

Используя эту особенность, можно организовать интерфейсы в иерархию, когда какой-то интерфейс расширяет существующий, наследуя все абстрактные члены базового интерфейса. Естественно, дочерние интерфейсы не наследуют реализацию, а расширяют собственное определение за счет добавления дополнительных абстрактных членов.

Применение иерархии интерфейсов удобно, когда необходимо расширить функциональность интерфейса без нарушения уже существующих кодовых баз.

```
public interface A
{ int Sum(); }

public interface B : A
{ int Del(); }

class Calculator : B
{
```

```
  int x = 10, y = 5;
  public int Sum()
  { return x + y; }
  public int Del()
  { return x / y; }
}
```

Класс Calculator обязан реализовать методы обоих интерфейсов.

Благодаря множественному наследованию в интерфейсах (один интерфейс может расширять сразу несколько интерфейсов) можно проектировать мощные и гибкие абстракции.

15.7 Явная реализация интерфейса

Согласно концепции ООП в C# классы могут реализовать любое количество интерфейсов. Может возникнуть ситуация, когда в разных интерфейсах могут быть члены с одинаковыми сигнатурами. Это приведет к необходимости в устранении конфликтов на уровне имен.

> Для этого при реализации члена интерфейса необходимо указать его имя полностью вместе с именем самого интерфейса. ИмяИнтерфейса Реализуемый_член. Это называется явной реализацией члена интерфейса (или явная реализация).

Если интерфейс наследует другой интерфейс, то в дочернем интерфейсе может быть объявлен член, скрывающий член с аналогичной сигнатурой в базовом интерфейсе, если не указать в объявлении члена производного интерфейса ключевое слово «new», то компилятор сгенерирует соответствующее предупреждающее сообщение.

Необходимость в явной реализации членов интерфейса обусловлена несколькими причинами:

1) если интерфейсный метод реализуется с указанием его полного имени, то такой метод доступен не через экземпляры классов, реализующих данный интерфейс, а по интерфейсной ссылке. Явная реализация позволяет реализовать интерфейсный метод таким, как private, что соответствует парадигме инкапсулирования в ООП;

2) в одном классе могут быть реализованы два интерфейса с членами, объявленными с одинаковыми именами и сигнатурами. Неоднозначность устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.

Пример:

```
using System;

namespace ConsoleApplication1
{
    public interface IName
    {
        void WriteName();
    }

    public interface INameFamily
    {
        // Объявляем в данном интерфейсе такой же метод
        void WriteName();
        void WriteFamily();
    }
```

```
    public interface IUserInfo : INameFamily
    {
        // Обязательно нужно указать ключевое слово «new»,
        // чтобы не скрывались методы базового интерфейса
        new void WriteName();
        void WriteUserInfo();
    }

    // Класс, реализующий два интерфейса
    class UserInfo : IUserInfo,IName
    {
        string ShortName, Family, Name;

        public UserInfo(string Name, string Family, string ShortName)
        {
            this.Name = Name;
            this.Family = Family;
            this.ShortName = ShortName;
        }

        // Используем явную реализацию интерфейсов
        // для исключения неоднозначности
        void IName.WriteName()
        {
            Console.WriteLine("Короткое имя: " + ShortName);
        }

        void INameFamily.WriteFamily()
        {
            Console.WriteLine("Фамилия: " + Family);
        }

        void INameFamily.WriteName()
        {
            Console.WriteLine("Полное имя: " + Name);
        }

        void IUserInfo.WriteName() { }

        public void WriteUserInfo()
        {
            UserInfo obj = new UserInfo(Name, Family, ShortName);
            // Для использования закрытых методов необходимо
            // создать интерфейсную ссылку
            IName link1 = (IName)obj;
            link1.WriteName();
            INameFamily link2 = (INameFamily)obj;
            link2.WriteName();
            link2.WriteFamily();
            IUserInfo link3 = (IUserInfo)obj;
            link3.WriteName();
        }
    }

    class Program
    {
        static void Main()
        {
            UserInfo obj = new UserInfo(Name: "Alexandr", ShortName: "Alex",
Family: "Erohin");
```

```
            obj.WriteUserInfo();
            Console.ReadLine();
        }
    }
}
```

Вопросы для самоконтроля

1. Что представляет собой интерфейс?
2. Возможна ли множественная реализация интерфейса?
3. Почему правильно говорить «реализация интерфейса», а не «наследование интерфейса»?
4. Что такое переменные-ссылки на интерфейс?
5. Для чего используется ключевое слово «as»?
6. Для чего используется ключевое слово «is»?
7. Особенности использования свойств в интерфейсе.
8. Возможно ли наследование интерфесами интерфейсов?
9. Явная и неявная реализация интерфейсов.

ГЛАВА 16
МАССИВЫ И КОЛЛЕКЦИИ В C#

16.1 Одномерные массивы

> Массивы – это именованная область памяти для хранения изменяемых значений одного типа.

Фактически размер массивов ничем не ограничен. Массивы могут быть использованы для хранения очень большого объема различных объектов, однако размерность массива желательно указывать непосредственно при объявлении массива.

Преимущество использования хранения данных в массиве обусловлено несколькими факторами – это доступ к произвольному элементу массива по числовому индексу, указывающему на его позицию «место» в массиве. Массивы могут содержать в себе как ссылочные типы, так и типы значений.

> Можно сказать, что массив – это индексированная коллекция объектов.

Объявление массива выглядит:

```
Тип[] имя_массива;
```

Следующий пример иллюстрирует инициализацию числа элементов массива при его объявлении:

```
int[] array = new int[5];
```

В этом случае все элементы массива по умолчанию будут равны нулю, но существует возможность их определить непосредственно при объявлении массива:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

или

```
int[] array2 = {1, 3, 5, 7, 9};
```

В том случае размер массива автоматически станет равным количеству инициализируемых элементов.

> Индексация элементов в массиве начинается с 0 (нулевого).

16.2 Двумерные массивы

Двумерные массивы (матрицы) объявляются аналогично:

```
int[,] array2D = new int[2,3];
int[,] array2D2 = { {1, 2, 3}, {4, 5, 6} };
```

16.3 Массивы массивов

Частным случаем многомерного массива является массив массивов.

> Массив массивов – это одномерный массив, каждый элемент которого является массивом.

Элементы массива не обязаны иметь одинаковый размер.
Объявление массива массивов происходит следующим образом:

```
int[][] ArrayOfArray = new int[3][];
```

16.4 Массивы объектов

С приходом объектно-ориентированного подхода появилась возможность создания и использования массива объектов. Создание такого массива происходит в два этапа: создание массива, создание объектов для хранения в массиве.

В качестве примера рассмотрим класс, который определяет студента, а вторым этапом создадим массив для хранения нескольких таких классов.

```
namespace StudentsCollection
{
    // Определим класс СТУДЕНТ
    class Student
    {
        private string name;
        private string group;
        private int kurs;

        public string Name
        {
            get {return this.name;}
            set {this.name = value;}
        }
        public string Group
        {
            get {return this.group;}
            set {this.group = value;}
        }
        public int Kurs
        {
            get {return this.kurs;}
            set {this.kurs = value;}
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Создадим массив для хранения объектов
            Student[] Students = new Student[20];

            // Цикл заполнения массива
            for (int i=0; i<20; i++)
            {
                Students[i] = new Student();
```

```
            }

            // Обработка произвольного элемента массива
            Students[0].Name = «Ваня»;
            Students[0].Group = "КС-ХХа";
            Students[0].Kurs = 1;
        }
    }
}
```

Вопросы для самоконтроля

1. Дайте определение понятию массив.
2. Одномерные и двумерные массивы.
3. Может ли быть объявлен массив массивов?
4. Существует ли массив объектов?

ГЛАВА 17
КОЛЛЕКЦИИ

> По сути своей коллекции – это те же массивы, но с расширенным функционалом. Можно сказать, что массивы, дополненные полезным «сахаром», и есть коллекции.

Как позже мы увидим, то именно так все и есть. В основе любой коллекции лежит массив. Рассмотрим, какие преимущества есть у коллекций по сравнению с массивами. Одним из преимуществ называют то, что коллекции могут динамически изменять размер. Кроме того, в коллекциях уже реализованы такие полезные методы, как добавление или удаление элемента коллекции, причем это можно сделать как с последним элементом в коллекции, так и с любым элементом, находящимся внутри на любой позиции. Ниже будут рассмотрены все эти преимущества подробнее с примерами, иллюстрирующими их применение.

На данном этапе можно предварительно сформулировать следующее определение коллекции:

> Коллекции – это гибкий инструмент для работы с группами объектов.
> В отличие от массивов, группа объектов в классе может динамически возрастать и сокращаться в соответствии с потребностями приложения.

Коллекции бывают типизированными и нетипизированными (обобщенные и необобщенные). Типизированные коллекции могут хранить объекты одного типа (подобно массивам). В нетипизированные можно добавлять объекты любого типа, что заявляется как преимущество. Однако такое преимущество спорно. Проведем аналогию со следующим примером.

Пусть человек, идя по лесу, собирает в нетипизированную коллекцию (корзинку) грибы (элементы коллекции). Во время сбора грибов (добавления элементов в коллекцию) человек действительно испытывает преимущества нетипизированной коллекции. Он может добавить в корзинку абсолютно любой гриб, не заботясь о его типе (съедобности). Однако когда возникнет задача обработать эти элементы (перебрать грибы) и что-либо с ними сделать (сварить грибной суп), то задача достать из корзинки грибы, определить их тип может быть затруднена. Подобно и с элементами коллекции в программировании.

Начнем рассмотрение с типизированных системных коллекций. C# предлагает несколько таких коллекций: списки, очереди, двоичные массивы, хэш-таблицы и словари. Все они описаны в пространстве имен System.Collections, поэтому для использования таких коллекций необходимо подключить соответствующее пространство имен.

```
using System.Collections
```

В таблице 17.1 приведен перечень стандартных коллекций C#.

Таблица 17.1 – Стандантные коллекции C#

Наименование коллекции	Описание коллекции
1. Dictionary<TKey,TValue>	Предоставляет коллекцию пар «ключ-значение», которые упорядочены по ключу
2. List<T>	Представляет список объектов, доступных по индексу. Предоставляет методы для поиска по списку, его сортировки и изменения
3. Queue<T>	Представляет коллекцию объектов, которая обслуживается в порядке поступления (FIFO)
4. SortedList<TKey,TValue>	Представляет коллекцию пар «ключ-значение», упорядоченных по ключу на основе реализации IComparer<T>
5. Stack<T>	Представляет коллекцию объектов, которая обслуживается в обратном порядке (LIFO)

17.1 Коллекция List<T>

Представляет собой строго типизированную коллекцию объектов, доступных по индексу, поддерживает методы поиска, сортировки и т. д.

Рассмотрим пример использования коллекции. В примере будет создан класс Student, который будет использован как элемент типизированной коллекции List<T>. Для демонстрации возможностей коллекции организуем добавление нового элемента в коллекцию по указанию индекса его позиции и удалим элемент коллекции по указанию его индекса.

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

namespace StandartCollections
{
    public class Student
    {
        public string Name { get; set; }
        public string Group { get; set; }
        public int Kurs { get; set; }

        /// <summary>
        /// Переопределим стандартный метод ToString
        /// </summary>
        /// <returns>Результатом работы будет сформированная по нашим требо-
ваниям строка</returns>
        public override string ToString()
        {
            return «Имя: « + Name + «   группа: « + Group + «   курс: « +
Kurs;
        }
    }
public class Program
{
    public static void Main()
    {
```

```
        // Создадим коллекцию List<>
        List<Student> students = new List<Student>();

        // Добавляем лементы в коллекцию
        students.Add(new Student() { Name="Иван    ", Group="КС-ХХ",
Kurs=1});
        students.Add(new Student() { Name="Андрей   ", Group="КС-ХХ",
Kurs=2});
        students.Add(new Student() { Name="Денис   ", Group="КС-ХХ",
Kurs=2});
        students.Add(new Student() { Name="Дмитрий ", Group="КС-ХХ",
Kurs=3});
        students.Add(new Student() { Name="Вова    ", Group="КС-ХХ",
Kurs=1});
        students.Add(new Student() { Name="Виктория", Group="КС-ХХ",
Kurs=3});
        students.Add(new Student() { Name="Тарас   ", Group="КС-ХХ",
Kurs=1});
        students.Add(new Student() { Name="Петя    ", Group="КС-ХХ",
Kurs=2});

        Console.WriteLine();
        // Выведем на экран список элементов коллекции,
        // при этом будет использован переопределенный нами стандартный метод
ToString
        foreach (Student aStudent in students)
        {
            Console.WriteLine(aStudent);
        }
        Console.WriteLine();
        // Добавим в коллекцию на место с индексом 2 новый объект
        students.Insert(2,new Student() { Name="Новый   ", Group="КС-ХХ",
Kurs=1});
        // Выведем на экран список элементов коллекции после преобразования,
        // при этом будет использован переопределенный нами стандартный метод
ToString
        foreach (Student aStudent in students)
        {
            Console.WriteLine(aStudent);
        }
        Console.WriteLine();

        // Удалим элемент коллекции с индексом 0
        students.RemoveAt(0);


        // Выведем на экран список элементов коллекции после преобразования,
        // при этом будет использован переопределенный нами стандартный метод
ToString
        foreach (Student aStudent in students)
        {
            Console.WriteLine(aStudent);
        }
        Console.WriteLine();
        Console.Write("Для выхода нажмите любую клавишу . . . «);
        Console.ReadKey(true);
    }
}
}
```

Результат работы будет следующий:

```
Имя: Иван       группа: КС-ХХ   курс: 1
Имя: Андрей     группа: КС-ХХ   курс: 2
Имя: Денис      группа: КС-ХХ   курс: 2
Имя: Дмитрий    группа: КС-ХХ   курс: 3
Имя: Вова       группа: КС-ХХ   курс: 1
Имя: Виктория   группа: КС-ХХ   курс: 3
Имя: Тарас      группа: КС-ХХ   курс: 1
Имя: Петя       группа: КС-ХХ   курс: 2

Имя: Иван       группа: КС-ХХ   курс: 1
Имя: Андрей     группа: КС-ХХ   курс: 2
Имя: Новый      группа: КС-ХХ   курс: 1
Имя: Денис      группа: КС-ХХ   курс: 2
Имя: Дмитрий    группа: КС-ХХ   курс: 3
Имя: Вова       группа: КС-ХХ   курс: 1
Имя: Виктория   группа: КС-ХХ   курс: 3
Имя: Тарас      группа: КС-ХХ   курс: 1
Имя: Петя       группа: КС-ХХ   курс: 2

Имя: Андрей     группа: КС-ХХ   курс: 2
Имя: Новый      группа: КС-ХХ   курс: 1
Имя: Денис      группа: КС-ХХ   курс: 2
Имя: Дмитрий    группа: КС-ХХ   курс: 3
Имя: Вова       группа: КС-ХХ   курс: 1
Имя: Виктория   группа: КС-ХХ   курс: 3
Имя: Тарас      группа: КС-ХХ   курс: 1
Имя: Петя       группа: КС-ХХ   курс: 2
```

В примере демонстрируются стандартные свойства методов List<T>.

List<T> является универсальным эквивалентом класса ArrayList, размер которого может динамически увеличиваться по мере добавления в него новых элементов. Индексация элементов начинается с нуля. Есть возможность увеличивать емкость коллекции, задав для атрибута enebled значение true.

Класс ArrayList имеет аналогичные методы, но List<T>, по заявлению разработчиков, работает быстрее.

Ниже приведена таблица членов класса List<T> с кратким их описанием (таблица 17.2).

Таблица 17.2 – Члены класса List<T>

Наименование члена класса	Описание
1	2
1. Конструкторы	
1.1. Public List()	Инициализирует новый экземпляр класса System.Collections.Generic.List <T>, коллекция пуста и имеет начальную емкость по умолчанию
1.2. Public List(IEnumerable<T> collection)	Инициализирует новый экземпляр класса System.Collections.Generic.List <T>, коллекция содержит элементы, скопированные из указанной коллекции, и имеет достаточную емкость для размещения количества копируемых элементов
1.3. Public List(int capacity)	Инициализирует новый экземпляр класса System.Collections.Generic.List <T>, коллекция пуста и имеет указанную начальную емкость
2. Свойства	
2.1. Public int Capacity { get; set; }	Получает или задает общее количество элементов, которые может хранить внутренняя структура данных без изменения размера
2.2. Public int Count { get; }	Возвращает количество элементов, содержащихся в коллекции List <T>

Продолжение табл. 17.2

1	2
2.3. Public T this[int index] { get; set; }	Получает или задает элемент по указанному индексу
3. Методы	
3.1. Public void Add(T item)	Добавляет объект в конец коллекции List <T>
3.2. Public void AddRange(IEnumerable<T> collection)	Добавляет элементы указанной коллекции в конец коллекции List <T>
3.3. Public ReadOnlyCollection<T> AsReadOnly()	Возвращает оболочку коллекции IList <T> для текущей коллекции
3.4. Public int BinarySearch(T item)	Ищет весь отсортированный System.Collections.Generic.List <T> для элемента с использованием сравнения по умолчанию и возвращает нулевой индекс элемента
3.5. Public int BinarySearch(T item, IComparer<T> comparer)	Ищет весь отсортированный System.Collections.Generic.List <T> для элемента с использованием указанного компаратора и возвращает нулевой индекс элемента
3.6. Public int BinarySearch(int index, int count, T item, IComparer<T> comparer)	Ищет ряд элементов в отсортированном System.Collections.Generic.List <T> для элемента с использованием указанного компаратора и возвращает нулевой индекс элемента
3.7. Public void Clear()	Удаляет все элементы из коллекции List <T>
3.8. Public bool Contains(T item)	Определяет, находится ли элемент в коллекции List <T>
3.9. Public List<TOutput> ConvertAll<TOutput>(Converter<T, TOutput> converter)	Преобразует элементы коллекции List <T> в другой тип и возвращает список, содержащий преобразованные элементы
3.10. Public void CopyTo(T[] array)	Копирует всю коллекцию List <T> в совместимый одномерный массив, начиная с начала целевого массива
3.11. Public void CopyTo(T[] array, int arrayIndex)	Копирует всю коллекцию List <T> в совместимый одномерный массив, начиная с указанного индекса целевого массива
3.12. Public void CopyTo(int index, T[] array, int arrayIndex, int count)	Копирует ряд элементов из коллекции List <T> в совместимый одномерный массив, начиная с указанного индекса целевого массива
3.13. Public bool Exists(Predicate<T> match)	Определяет, содержит ли коллекция List <T> элементы, которые соответствуют условиям, определенным указанным предикатом
3.14. Public T Find(Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает первое вхождение в коллекцию List <T>
3.15. Public List<T> FindAll(Predicate<T> match)	Получает все элементы, соответствующие условиям, определенным указанным предикатом
3.16. Public int FindIndex(Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс первого вхождения во всей коллекции List <T>
3.17. Public int FindIndex(int startIndex, Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс первого вхождения в пределах диапазона элементов в System.Collections.Generic.List <T>, который простирается от указанного индекса до последнего элемента
3.18. Public int FindIndex(int startIndex, int count, Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс первого вхождения в пределах диапазона элементов в System.Collections.Generic.List <T>, который начинается с указанного индекса и содержит указанное количество элементов
3.19. Public T FindLast(Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает последнее вхождение во всей коллекции List <T>
3.20. Public int FindLastIndex(Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс последнего вхождения во всей коллекции List <T>
3.21. Public int FindLastIndex(int startIndex, Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс последнего вхождения в пределах диапазона элементов в коллекции List <T>, который простирается от первого элемента до указанного индекса
3.22. Public int FindLastIndex(int startIndex, int count, Predicate<T> match)	Ищет элемент, который соответствует условиям, определенным указанным предикатом, и возвращает нулевой индекс последнего вхождения в диапазоне элементов в коллекции List <T>, который содержит указанное количество элементов и заканчивается по указанному индексу

Окончание табл. 17.2

1	2
3.23. Public List<T>.Enumerator GetEnumerator()	Возвращает перечислитель, который выполняет итерацию через System.Collections.Generic.List <T>
3.24. Public List<T> GetRange(int index, int count)	Создает копию ряда элементов в источнике System.Collections.Generic.List <T>
3.25. Public int IndexOf(T item)	Ищет указанный объект и возвращает индекс первого вхождения во всей System.Collections.Generic.List <T>
3.26. Public int IndexOf(T item, int index)	Ищет указанный объект и возвращает индекс первого вхождения в пределах диапазона элементов в System.Collections.Generic.List <T>, который продолжается от указанного индекса до последнего элемента
3.27. Public int IndexOf(T item, int index, int count)	Ищет указанный объект и возвращает индекс первого вхождения в пределах диапазона элементов в System.Collections.Generic.List <T>, который начинается с указанного индекса и содержит указанное количество элементов
3.28. Public void Insert(int index, T item)	Вставляет элемент в System.Collections.Generic.List <T> по указанному индексу
3.29. Public void InsertRange(int index, IEnumerable<T> collection)	Вставляет элементы коллекции в System.Collections.Generic.List <T> по указанному индексу
3.30. Public int LastIndexOf(T item)	Ищет указанный объект и возвращает нулевой индекс последнего вхождения во всей коллекции List <T>
3.31. Public int LastIndexOf(T item, int index)	Ищет указанный объект и возвращает индекс последнего вхождения в диапазоне элементов в System.Collections.Generic.List <T>, который простирается от первого элемента до указанного индекса
3.32. Public int LastIndexOf(T item, int index, int count)	Ищет указанный объект и возвращает индекс последнего вхождения в диапазоне элементов в System.Collections.Generic.List <T>, который содержит указанное количество элементов и заканчивается по указанному индексу
3.33. Public bool Remove(T item)	Удаляет первое вхождение определенного объекта
3.34. Public int RemoveAll(Predicate<T> match)	Удаляет все элементы, соответствующие условиям, определенным указанным предикатом
3.35. Public void RemoveAt(int index)	Удаляет элемент по указанному индексу
3.36. Public void RemoveRange(int index, int count)	Удаляет ряд элементов из коллекции List <T>
3.37. Public void Reverse()	Изменяет порядок элементов во всей коллекции List <T>
3.38. Public void Reverse(int index, int count)	Изменяет порядок элементов в указанном диапазоне
3.39. Public void Sort()	Сортирует элементы во всем System.Collections.Generic.List <T> с помощью сравнения по умолчанию
3.40. Public void Sort(Comparison<T> comparison)	Сортирует элементы во всем System.Collections.Generic.List <T>, используя указанный System.Comparison <T>
3.41. Public void Sort(IComparer<T> comparer)	Сортирует элементы во всем System.Collections.Generic.List <T> с использованием указанного компаратора
3.42. Public void Sort(int index, int count, IComparer<T> comparer)	Сортирует элементы в диапазоне элементов в System.Collections.Generic.List <T> с использованием указанного компаратора
3.43. Public T[] ToArray()	Копирует элементы System.Collections.Generic.List <T> в новый массив
3.44. Public void TrimExcess()	Устанавливает емкость фактического количества элементов в System.Collections.Generic.List <T>, если это число меньше порогового значения
3.45. Public bool TrueForAll(Predicate<T> match)	Определяет, соответствует ли каждый элемент в System.Collections.Generic.List <T> условиям, определенным указанным предикатом
3.46. Public T Current { get; }	Получает элемент в текущей позиции счетчика
3.47. Public void Dispose()	Освобождает все ресурсы, используемые System.Collections.Generic.List <T> .Enumerator
3.48. Public bool MoveNext()	Активирует перечислитель для следующего элемента System.Collections.Generic.List <T>

17.2 Коллекция Dictionary<T, V>

Коллекция представляет собой словарь, состоящий из пары ключ-значение.

Коллекция содержит объекты, представляющие пару ключ-значение. Каждый элемент коллекции является объектом структуры KeyValuePair<TKey, TValue>. Используя свойства Key и Value, можно обеспечить доступ к элементу коллекции.

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

namespace StandartCollectionsDictionary
{
class Program
{
    public static void Main()
    {
        // Создадим словарь со строковыми ключ-значение.
        //
        Dictionary<string, string> students = new Dictionary<string,
string>();

        // Добавим в коллекцию словарь элементы.
        // Ключ должен быть уникален, значение нет.
        students.Add("0001", "Иванов");
        students.Add("0002", "Петров");
        students.Add("0003", "Сидоров");
        students.Add("0004", "Трамп");

        // Попытаемся вызвать исключение, добавив элемент с тем же ключом
        try
        {
            students.Add("0002", "Сидоров");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("Элемент с таким ключом уже есть в коллек-
ции.");
        }

        // Отобразим значение для определенного ключа
        Console.WriteLine("Для ключа \"0002\", value = {0}.",
students["0002"]);


        // Установим по ключу новое значение
        students["0001"] = "Иванов И.И.";

        // Вызовем исключение "не тот ключ"
        try
        {
            Console.WriteLine("Для ключа \"0007\", value = {0}.",
students["0007"]);
        }
        catch (KeyNotFoundException)
        {
            Console.WriteLine("Ключ \"0007\" не найден");
        }
```

```
        // Выберем только значения из коллекции
        Dictionary<string, string>.ValueCollection valueColl =
students.Values;

        // Получившиеся строго типизированные элементы выведем на экран
        Console.WriteLine();
        foreach( string s in valueColl )
        {
            Console.WriteLine("Зачение = {0}", s);
        }

        // Выберем только ключи
        Dictionary<string, string>.KeyCollection keyColl = students.Keys;

        // Получившиеся строго типизированные элементы выведем на экран
        Console.WriteLine();
        foreach( string s in keyColl )
        {
            Console.WriteLine("Ключ = {0}", s);
        }

        // Попытаемся удалить несуществующий элемент
        Console.WriteLine("\nУдалим несуществующий элемент (\"0007\")");
        students.Remove("0007");

        if (!students.ContainsKey("0007"))
        {
            Console.WriteLine("Ключ \"0007\" ошибочный");
        }
        Console.ReadKey();
    }
    }
}
```

В результате выполнения получим:

```
Элемент с таким ключому уже есть в коллекции.
Для ключа "0002", value = Петров.
Ключ "0007" не найден

Зачение = Иванов И.И.
Зачение = Петров
Зачение = Сидоров
Зачение = Трамп

Ключ = 0001
Ключ = 0002
Ключ = 0003
Ключ = 0004

Удалим несуществующий элемент ("0007")
Ключ "0007" ошибочный
```

Ниже приведены таблицы 17.3–17.4 членов класса Dictionary<T> с кратким их описанием.

Таблица 17.3 – Члены класса Dictionary<T>

Наименование членов класса	Описание
1. Конструкторы	
Dictionary<TValue TKey,>.ValueCollection(Dictionary<TValue TKey,>)	Инициализирует новый экземпляр Dictionary<TValue TKey,>.ValueCollection класс, который отражает значения в указанном Dictionary<TValue TKey,>
2. Свойства	
Count	Получает число элементов, содержащихся в интерфейсе Dictionary<TValue TKey,>.ValueCollection
3. Методы	
3.1. CopyToInt32) (TValue[],	Копирует элементы коллекции Dictionary<TValue TKey,>.ValueCollection в существующий одномерный массив Array, начиная с указанного значения индекса массива
3.2. Equals(Object)	Определяет, равен ли заданный объект текущему объекту
3.3. GetEnumerator()	Возвращает перечислитель, осуществляющий перебор элементов списка Dictionary<TValue TKey,>.ValueCollection
3.4. GetHashCode()	Служит хэш-функцией по умолчанию
3.5. GetType()	Возвращает объект Type для текущего экземпляра
3.6. ToString()	Возвращает строку, представляющую текущий объект
3.7. ICollection<TValue>.Add(TValue)	Добавляет элемент в коллекцию ICollection<T>. Данная реализация всегда выдает NotSupportedException
3.8. ICollection<TValue>.Clear()	Удаляет все элементы из коллекции ICollection<T>. Данная реализация всегда создает исключение NotSupportedException
3.9. ICollection<TValue>.Contains(TValue)	Определяет, содержит ли коллекция ICollection<T> указанное значение
3.10. ICollection<TValue>.Remove(TValue)	Удаляет первое вхождение указанного объекта из коллекции ICollection<T>. Данная реализация всегда выдает NotSupportedException
3.11. IEnumerable<TValue>.GetEnumerator()	Возвращает перечислитель, который осуществляет итерацию по коллекции
3.12. ICollection.CopyToInt32) (Array,	Копирует элементы коллекции ICollection в массив Array, начиная с указанного индекса массива Array
3.13. IEnumerable.GetEnumerator()	Возвращает перечислитель, который осуществляет итерацию по коллекции
3.14. ICollection<TValue>.IsReadOnly	Получает значение, указывающее, является ли объект ICollection<T> доступным только для чтения
3.15. ICollection.IsSynchronized	Возвращает значение, показывающее, является ли доступ к коллекции ICollectionсинхронизированным (потокобезопасным)
3.16. ICollection.SyncRoot	Получает объект, с помощью которого можно синхронизировать доступ к коллекции ICollection

Таблица 17.4 – Методы расширения класса Dictionary<T>

Наименование метода расширения	Описание
1	**2**
1. All<TValue>(Func<Boolean TValue,>)	Определяет, удовлетворяют ли условию все элементы последовательности
2. Any<TValue>()	Перегружен. Определяет, содержит ли последовательность какие-либо элементы
3. Any<TValue>(Func<Boolean TValue,>)	Перегружен. Определяет, удовлетворяет ли условие какой-либо элемент последовательности
4. AsEnumerable<TValue>()	Возвращает входные данные, типизированные как IEnumerable<T>
5. AsParallel()	Перегружен. Позволяет осуществлять параллельный запрос
6. AsParallel<TValue>()	Перегружен. Позволяет осуществлять параллельный запрос
7. AsQueryable()	Перегружен. Преобразует IEnumerable для IQueryable
8. AsQueryable<TValue>()	Перегружен. Преобразует универсальный IEnumerable<T> к универсальному IQueryable<T>

Продолжение табл. 17.4

1	2
9. Average<TValue>(Func<Decimal TValue,>)	Перегружен. Вычисляет среднее для последовательности Decimal значений, получаемой в результате применения функции преобразования к каждому элементу входной последовательности
10. Average<TValue>(Func<Single TValue,>)	Перегружен. Вычисляет среднее для последовательности значений типа Single, получаемой в результате применения функции преобразования к каждому элементу входной последовательности
11. Cast<TResult>()	Приводит элементы IEnumerable для указанного типа
12. Concat<TValue>(IEnumerable<TValue>)	Объединяет две последовательности
13. Contains<TValue>(TValue)	Перегружен. Определяет, содержит ли последовательность указанный элемент, используя компаратор проверки на равенство по умолчанию
14. Contains<TValue>IEqualityComparer (TValue,<TValue>)	Перегружен. Определяет, содержит ли последовательность указанный элемент, используя указанный IEqualityComparer<T>
15. Count<TValue>()	Перегружен. Возвращает количество элементов в последовательности
16. Count<TValue>(Func<Boolean TValue,>)	Перегружен. Возвращает число, представляющее количество элементов в указанной последовательности, удовлетворяющих условию
17. DefaultIfEmpty<TValue>()	Перегружен. Возвращает элементы указанной последовательности или одноэлементную коллекцию, содержащую указанное значение, если последовательность пуста
18. DefaultIfEmpty<TValue>(TValue)	Перегружен. Возвращает элементы указанной последовательности или одноэлементную коллекцию, содержащую указанное значение, если последовательность пуста
19. Distinct<TValue>()	Перегружен. Возвращает различающиеся элементы последовательности, используя для сравнения значений компаратор проверки на равенство по умолчанию
20. Distinct<TValue>(IEqualityComparer <TValue>)	Перегружен. Возвращает различающиеся элементы последовательности, используя указанную IEqualityComparer<T> для сравнения значений
21. ElementAt<TValue>(Int32)	Возвращает элемент по указанному индексу в последовательности
22. ElementAtOrDefault<TValue>(Int32)	Возвращает элемент последовательности по указанному индексу или значение по умолчанию, если индекс вне допустимого диапазона
23. Except<TValue>(IEnumerable<TValue>)	Перегружен. Находит разность двух последовательностей, используя для сравнения значений компаратор проверки на равенство по умолчанию
24. Except<TValue>(IEnumerable<TValue>IEqualityComparer ,<TValue>)	Перегружен. Находит разность двух последовательностей с помощью заданного IEqualityComparer<T> для сравнения значений
25. First<TValue>()	Перегружен. Возвращает первый элемент последовательности
26. First<TValue>(Func<Boolean TValue,>)	Перегружен. Возвращает первый элемент последовательности, удовлетворяющий указанному условию
27. GroupBy<TKey TValue,>(Func<TKey TValue,>)	Перегружен. Группирует элементы последовательности в соответствии с заданной функцией селектора ключа
28. GroupJoin<TResult TKey, TInner, TValue,>(IEnumerable<TInner>Func ,<TKey TValue,>Func ,<TKey TInner,>Func ,<IEnumerable TValue,<TInner>TResult ,>)	Перегружен. Устанавливает корреляцию между элементами двух последовательностей на основе равенства ключей и группирует результаты. Для сравнения ключей используется компаратор проверки на равенство по умолчанию
29. Intersect<TValue>(IEnumerable<TValue>)	Перегружен. Находит пересечение двух последовательностей, используя для сравнения значений компаратор проверки на равенство по умолчанию
30. Join<TResult TKey, TInner, TValue,>(IEnumerable<TInner>Func ,<TKey TValue,>Func ,<TKey TInner,>Func ,<TResult TInner, TValue,>)	Перегружен. Устанавливает корреляцию между элементами двух последовательностей на основе сопоставления ключей. Для сравнения ключей используется компаратор проверки на равенство по умолчанию
31. Last<TValue>()	Перегружен. Возвращает последний элемент последовательности
32. LastOrDefault<TValue>()	Перегружен. Возвращает последний элемент последовательности или значение по умолчанию, если последовательность не содержит элементов

Окончание табл. 17.4

1	2
33. LastOrDefault<TValue>(Func<Boolean TValue,>)	Перегружен. Возвращает последний элемент последовательности, удовлетворяющий указанному условию, или значение по умолчанию, если ни одного такого элемента не найдено.
34. LongCount<TValue>()	Перегружен. Возвращает Int64, представляет собой общее число элементов в последовательности
35. Max<TValue>()	Перегружен. Возвращает максимальное значение, содержащееся в универсальной последовательности
36. Max<TValue>(Func<Decimal TValue,>)	Перегружен. Вызывает функцию преобразования для каждого элемента последовательности и возвращает максимальное Decimal значение
37. Min<TValue>()	Перегружен. Возвращает минимальное значение, содержащееся в универсальной последовательности
38. OfType<TResult>()	Фильтрует элементы IEnumerable на основе указанного типа
39. OrderBy<TKey TValue,>(Func<TKey TValue,>)	Перегружен. Сортирует элементы последовательности в возрастающем порядке по ключу
40. Reverse<TValue>()	Изменяет порядок элементов в последовательности
41. Select<TResult TValue,>(Func<TResult TValue,>)	Перегружен. Проецирует каждый элемент последовательности в новую форму
42. Select<TResult TValue,>(Func<TResult Int32, TValue,>)	Перегружен. Проецирует каждый элемент последовательности в новую форму, добавляя индекс элемента
43. SelectMany<TResult TValue,>(Func<IEnumerable TValue,<TResult>>)	Перегружен. Проецирует каждый элемент последовательности в IEnumerable<T> и объединяет результирующие последовательности в одну последовательность
44. SequenceEqual<TValue>(IEnumerable<TValue>)	Перегружен. Определяет, совпадают ли две последовательности, используя для сравнения элементов компаратор проверки на равенство по умолчанию для их типа
45. Single<TValue>()	Перегружен. Возвращает единственный элемент последовательности и вызывает исключение, если число элементов последовательности отлично от одного
46. Skip<TValue>(Int32)	Пропускает заданное число элементов в последовательности и возвращает остальные элементы
47. SkipWhile<TValue>(Func<Boolean TValue,>)	Перегружен. Пропускает элементы в последовательности, пока заданное условие истинно и затем возвращает оставшиеся элементы
48. Sum<TValue>(Func<Decimal TValue,>)	Перегружен. Вычисляет сумму последовательности Decimal значений, получаемой в результате применения функции преобразования к каждому элементу входной последовательности
49. Take<TValue>(Int32)	Возвращает заданное число смежных элементов с начала последовательности
50. TakeWhile<TValue>(Func<Boolean TValue,>)	Перегружен. Возвращает элементы из последовательности, пока указанное условие истинно
51. ToArray<TValue>()	Создает массив из IEnumerable<T>
52. ToDictionary<TKey TValue,>(Func<TKey TValue,>)	Перегружен. Создает Dictionary<TValue TKey,> из IEnumerable<T> в соответствии с заданной функцией селектора ключа
53. Union<TValue>(IEnumerable<TValue>)	Перегружен. Находит объединения наборов двух последовательностей, используя компаратор проверки на равенство по умолчанию
54. Where<TValue>(Func<Boolean TValue,>)	Перегружен. Выполняет фильтрацию последовательности значений на основе заданного предиката

17.3 Коллекция Queue<T>

Коллекция Queue – это очередь, работающая по принципу стека FIFO (first in, first out).

Данная коллекция реализуется при помощи классов из пространства имен System.Collections и System.Collections.Generic в случае типизированной Queue<T>.

Схематически можно изобразить работу коллекции следующим образом:

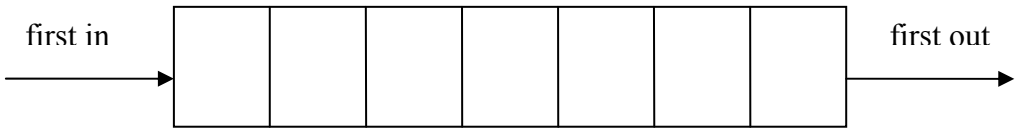


Рисунок 17.1 – Технология работы коллекции Queue<T>

Простой пример, иллюстрирующий работу очереди. В примере создаем экземпляр класса Queue<T>, заполняем очередь объектами типа int путем ввода с клавиатуры; выводим очередь на консоль.

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

namespace StandartCollectionsQueue
{
    class Program
    {
        public static void Main(string[] args)
        {
            // Создадим экземпляр класса Queue<T>
            Queue<int> queue = new Queue<int>();


            // Заполним очередь объектами
            for (int i = 0; i < 10; i++)
                queue.Enqueue(Convert.ToInt32(Console.ReadLine()));

            // Отобразим очередь в консоли
            Console.WriteLine("Очередь: \n");
            foreach (int i in queue)
                Console.Write(i);

            Console.Write("\nДля продолжения нажмите любую клавишу . . . ");
            Console.ReadKey(true);
        }
    }
}
```

Результатом работы будет:

```
1
5
3
7
8
3
9
0
2
4
Очередь:

1537839024
Для продолжения нажмите любую клавишу . . .
```

Для работы с очередью определены следующие члены класса Queue<T> (таблица 17.5):

Таблица 17.5 – Члены типизированного класса Queue<T>

Наименование члена класса	Описание
1. Конструкторы	
1.1. Public Queue()	Создается пустая очередь с выбираемой по умолчанию первоначальной емкостью
1.2 Public Queue(int capacity)	Создается пустая очередь, первоначальный размер которой определяет параметр capacity
1.3 Public Queue(IEnumerable<T> collection)	Создается очередь, содержащая элементы коллекции, определяемой параметром collection. Ее первоначальная емкость равна количеству указанных элементов
2. Свойства	
Count	Возвращает количество элементов в очереди
3. Методы	
3.1 Enqueue()	Добавляет элемент в конец очереди
3.2 Dequeue()	Читает и удаляет элемент из головы очереди. Если на момент вызова метода Dequeue() элементов в очереди больше нет, генерируется исключение InvalidOperationException
3.3 Peek()	Читает элемент из головы очереди, но не удаляет его
3.4 TrimExcess()	Изменяет емкость очереди. Метод Dequeue() удаляет элемент из очереди, но не изменяет ее емкости. TrimExcess() позволяет избавиться от пустых элементов в начале очереди

17.4 Сортированный список SortedList<TKey, TValue>

Особенность данной коллекции в том, что элементы сортируются по значению ключа. Размер такой коллекции изменяется в зависимости от количества объектов в ней. Сортированный список подобен коллекции SortedDictionary<TKey, TValue> сортированный словарь, но обладает следующими отличиями: сортированный список требует меньших затрат памяти, но большего времени на добавление в коллекцию неупорядоченных объектов.

Особенность динамически изменять размер коллекции делает нецелесообразным указание ее емкости в момент объявления, однако указывать размер рекомендуется, если заранее известен ее размер.

Ниже приведен пример использования коллекции:

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

namespace StandartCollectionsSortedList
{
    class Program
    {
        public static void Main(string[] args)
        {
           // Создадим коллекцию сортированного списка

            SortedList<string, string> students = new SortedList<string,
string>();

            // Добавим несколько элементов в коллекцию
            students.Add("Иванов", "КС-ХХ");
            students.Add("Петров", "КС-ХХ");
            students.Add("Сидоров", "ПО-ХХ");
            students.Add("Боднар", "ЭК-ХХ");

            // Коллекция ключей
            ICollection<string> keys = students.Keys;
```

```
            // Теперь используем ключи для получения значений
            foreach (string s in keys)
                Console.WriteLine("Студент: {0} \t Группа: {1}", s,
students[s]);

            Console.Write("\nДля продолжения нажмите любую клавишу . . . ");
            Console.ReadKey(true);
        }
    }
}
```

В результате имеем:

```
Студент: Боднар          Группа: ЭК-ХХ
Студент: Иванов          Группа: КС-ХХ
Студент: Петров          Группа: КС-ХХ
Студент: Сидоров         Группа: ПО-ХХ

Для продолжения нажмите любую клавишу . . .
```

Рассмотрим стандартные члены класса SortedList в таблице 17.6.

Таблица 17.6 – Стандартные члены класса SortedList

Наименование члена класса	Описание
1. Конструкторы	
1.1. Public SortedList()	Создается пустой список с выбираемой по умолчанию первоначальной емкостью
1.2. Public Sorted-List(IDictionary<TKey, TValue> dictionary)	Создается отсортированный список с указанным количеством элементов dictionary
1.3. Public SortedList(int capacity)	Создается список с указанием емкости коллекции
1.4. Public Sorted-List(IComparer<TK> comparer)	Создается список с указанием компаратора, описывающего правило сравнения объектов, содержащихся в списке
2. Методы	
2.1. Add()	Добавляет в список пару «ключ-значение». Если ключ уже находится в списке, то его значение не изменяется, и генерируется исключение Ar-gumentException
2.2. ContainsKey()	Возвращает логическое значение true, если вызывающий список содер-жит объект key в качестве ключа; а иначе – логическое значение false
2.3. ContainsValue()	Возвращает логическое значение true, если вызывающий список содер-жит значение value; в противном случае – логическое значение false
2.4. GetEnumerator()	Возвращает перечислитель для вызывающего словаря
2.5. IndexOfKey(), IndexOf-Value()	Возвращает индекс ключа или первого вхождения значения в вызываю-щем списке. Если искомый ключ или значение не обнаружены в списке, возвращается значение –1
2.6. Remove()	Удаляет из списка пару «ключ-значение» по указанному ключу key. При удачном исходе операции возвращается логическое значение true, а если ключ key отсутствует в списке – логическое значение false
2.7. TrimExcess()	Сокращает избыточную емкость вызывающей коллекции в виде отсорти-рованного списка
3. Свойства	
3.1. Capacity	Получает или устанавливает емкость вызывающей коллекции в виде от-сортированного списка
3.2. Comparer	Получает метод сравнения для вызывающего списка
3.3. Keys	Получает коллекцию ключей
3.4. Values	Получает коллекцию значений

Итак, подводя итог, можно отметить, что коллекции – это удобный инструмент для хранения и обработки набора объектов. Принцип работы коллекций напоминает массивы, по сути классы-коллекции внутри используют именно массивы, но они обрамлены удобными методами, которые значительно упрощают работу с набором объектов.

> Коллекция – это более гибкая организация данных, чем обычный массив. Но работа с массивом всегда ведётся быстрее, чем с коллекцией.

В последних версиях C# постоянно появляются «сахарные» конструкции для работы с коллекциями. Рассмотрим некоторые из них.

> Если содержимое коллекции известно заранее, можно использовать инициализатор коллекции для инициализации коллекции.

Пример:

```
// Создадим список слов.
var words = new List<string> {”мама”,”мыла”,”раму”};

foreach (var word in words)
{
    Console.Write(word + " ");
}
// Результат: мама мыла раму
```

Можно использовать оператор for вместо оператора ForEach для прохода по коллекции. Это выполняется путем доступа к элементам коллекции по позиции индекса.

> Индекс элементов начинается с 0 и заканчивается на числе, равном количеству элементов за –1.

Пример:

```
// Создадим список слов.
var words = new List<string> {”мама”,”мыла”,”раму”};

for (var index = 0; index < words.Count; index++)
{
    Console.Write(word + " ");
}
// Результат: мама мыла раму
```

В завершение рассмотрим пример сортировки коллекций, в котором выполняется сортировка вначале по полю Name в порядке возростания, а затем по полю Age в порядке убывания.

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

namespace CollectionSort
{
      class Student: IComparable<Student>
      {
            public string Name {get; set;}
```

```
            public int Age {get; set;}

            public Student(string name, int age)
            {
                  this.Name = name;
                  this.Age = age;
            }

            public new string ToString()
            {
                  return Name + " " + Age;
            }
            public int CompareTo(Student obj)
            {
                  int compare;
                        compare = String.Compare(this.Name, obj.Name);

                  if (compare == 0)
                  {
                        compare = this.Age.CompareTo(obj.Age);
                        compare = -compare;
                  }
                  return compare;
            }
      }

      class Program
      {


      public static void Main(string[] args)
      {
            Console.WriteLine("Сортировка коллекций");
            List<Student> students = new List<Student>();
            students.Add(new Student("Андрей   ",19));
            students.Add(new Student("Андрей   ",20));
            students.Add(new Student("Вова    ",17));
            students.Add(new Student("Вова    ",19));
            students.Add(new Student("Вова    ",16));
            students.Add(new Student("Андрей   ",18));
            students.Add(new Student("Денис   ",18));
            students.Add(new Student("Вова    ",17));
            students.Add(new Student("Виктория",18));
            students.Add(new Student("Андрей   ",16));

            Console.WriteLine("До сортировки");
            foreach(Student aStudent in students)
            {
                  Console.WriteLine(aStudent.ToString());
            }

            students.Sort();

            Console.WriteLine("\nПосле сортировки");
            foreach(Student aStudent in students)
            {
                  Console.WriteLine(aStudent.ToString());
            }
            Console.WriteLine("Нажми клавишу для выхода . . . ");
            Console.ReadKey(true);
      }
      }
}
```

Результат:

```
До сортировки
Андрей   19
Андрей   20
Вова     17
Вова     19
Вова     16
Андрей   18
Денис    18
Вова     17
Виктория 18
Андрей   16

После сортировки
Андрей   20
Андрей   19
Андрей   18
Андрей   16
Виктория 18
Вова     19
Вова     17
Вова     17
Вова     16
Денис    18
```

Вопросы для самоконтроля

1. Что такое коллекция?
2. Чем массив отличается от коллекции?
3. Что такое типизированные коллекции?
4. В чем особенности работы с коллекцией List<T>?
5. В чем особенности работы с коллекцией Dictionary<T,V>?
6. В чем особенности работы с коллекцией Queue<T>?
7. В чем особенности работы с коллекцией SortedList<TKey, TValue>?
8. Что такое инициализатор коллекции?

ГЛАВА 18
ПОЛЬЗОВАТЕЛЬСКИЕ КОЛЛЕКЦИИ В C#

В целом все основные задачи, связанные с использованием коллекций, позволяют решить набор стандартных коллекций. Однако существуют ситуации, когда необходимо создать свою собственную «пользовательскую» коллекцию.

Из темы про коллекции известно, что коллекции не что иное, как массивы снабженные (дополненные) определенным набором методов, облегчающих их использование. Рассмотрим пример создания и использования пользовательской коллекции. Первым создадим класс, который будет являться элементом коллекции. В классе реализуем его члены – свойства и конструктор.

```
public class Student
{
  public string Name { get; set; }
  public int Age { get; set; }

  public Student(string arg1, int arg2)
  {
    this.Name = arg1;
    this.Age = arg2;
  }
}
```

Далее создадим класс, представляющий собой пользовательскую коллекцию для работы с нашими элементами. Для того чтобы имелась возможность перебирать элементы коллекции при помощи оператора foreach, необходимо реализовать интерфейсы IEnumerable и IEnumerator. Перебираемая коллекция должна реализовать IEnumerable, который выглядит следующим образом:

```
namespace System.Collections
{
  public interface IEnumerable
  {
    IEnumerator GetEnumerator();
  }
}
```

Как видно, в этом интерфейсе всего один метод GetEnumerator, который возвращает ссылку на интерфейс IEnumerator (перечислитель), в котором уже определен весь необходимый инструментарий для перебора элементов коллекции.

```
namespace System.Collections
{
  public interface IEnumerator
  {
    object Current {get;}
    bool MoveNext();
    void Reset();
  }
}
```

> Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции.

Таким набором есть свойство только для чтения Current и два метода MoveNext() (для перемещения указателя к следующему элементу) и Reset() (для возвращения указателя на начало коллекции). Если быть точным, то на несуществующий элемент, который не указывает ни на один элемент внутреннего массива коллекции.

Реализация этих интерфейсов представлена в следующем классе:

```
public class MyCollection : IEnumerable, IEnumerator
{
  readonly Student[] students = new Student[10];

  public Student this[int index]
  {
    get { return students[index]; }
    set { students[index] = value; }
  }

  int pos = -1;

  // Реализация интерфейса IEnumerator:
  bool IEnumerator.MoveNext()
  {
    if (pos < students.Length - 1)
      {
        pos++;
        return true;
      }
    ((IEnumerator)this).Reset();
    return false;
  }

  void IEnumerator.Reset()
  {
    pos = -1;
  }

  object IEnumerator.Current
  {
    get { return students[pos]; }
  }

  // Реализация интерфейса IEnumerable:
  IEnumerator IEnumerable.GetEnumerator()
  {
    return (IEnumerator)this;
  }
}
```

Поясним подробнее этот класс. Первым полем в классе мы объявили массив только для чтения из 10 элементов типа Student.

```
readonly Student[] students = new Student[10];
```

Таким образом, наша пользовательская коллекция будет уметь хранить до 10 элементов одного типа, по сути она будут обобщенной. Чтобы превратить ее в необобщенную, необходимо использовать тип object для массива, но это потребует дополнительных манипуляций при работе с элементами массива. Для учебного примера допустима такая неточность.

Чтобы коллекция, как и массив, могла обеспечивать доступ к любому элементу по индексу, необходимо реализовать индексатор.

```
public Student this[int index]
{
  get { return students[index]; }
  set { students[index] = value; }
}
```

Индексатор, в нашем случае, будет представлять собой специальное свойство с именем «this» того же типа, что и элемент коллекции, и в прямоугольных скобках принимать целочисленный аргумент index – порядковый номер (индекс) элемента коллекции.

Следующей строкой «сбросим» указатель на начало коллекции.

```
int pos = -1;
```

Теперь необходимо реализовать интерфейсы. Первым реализуем интерфейс IEnumerator в котором три члена:

– метод MoveNext() обеспечивает перемещение указателя на следующий элемент, кстати, можно реализовать свой собственный алгоритм перемещения указателя по коллекции. Метод возвращает true тогда, когда успешно удалось переместить указатель на следующий элемент, и false, если переместить указатель не получилось, например достигнут конец коллекции, при этом указатель снова устанавливается в –1 вызовом метода Reset():

```
bool IEnumerator.MoveNext()
{
  if (pos < students.Length - 1)
    {
      pos++;
      return true;
    }
  ((IEnumerator)this).Reset();
  return false;
}
```

– метод Reset() обеспечивает сброс указателя в –1:

```
void IEnumerator.Reset()
{
  pos = -1;
}
```

– свойство Current возвращает текущий элемент, на который указывает индекс:

```
object IEnumerator.Current
{
  get { return students[pos]; }
}
```

Второй интерфейс содержит только один член – это метод GetEnumerator(). Его реализация выглядит следующим образом:

```
IEnumerator IEnumerable.GetEnumerator()
{
  return (IEnumerator)this;
}
```

Управляющий метод Main() выглядит следующим образом:

```
static void Main()
    {
        var collection = new UserCollection();

        collection[0] = new Student(«Петров», 18);
        collection[1] = new Student(«Сидоров», 18);
        collection[2] = new Student(«Иванов», 19);
        ...
        collection[9] = new Student(«Друзья», 20);

        foreach (Student student in collection)
        {
            Console.WriteLine("{0}, {1}", student.Name, student.Age);
        }

        Console.WriteLine(new string('-', 5));

        // Так работает foreach
        IEnumerator enumerator = (collection as IEnumerable).GetEnumerator();
        while (enumerator.MoveNext())
        {
            Student student = enumerator.Current as Student;
            Console.WriteLine("{0}, {1}", student.Name, student.Age);
        }
        // Delay.
        Console.ReadKey();
    }
```

Выше было сказано, что коллекция обязана реализовать два интерфейса IEnumerator и IEnumerable, однако так ли это? Попробуем убрать из строки

```
public class MyCollection : IEnumerable, IEnumerator
```

реализацию интерфейсов и слегка изменим код класса UserCoolection, уберем явную реализацию интерфейса, ведь мы теперь их не реализуем, получим следующий код:

```
public class UserCollection
{
    readonly Student [] students = new Student [10];

    public Student this[int index]
    {
        get { return students[index]; }
        set { students[index] = value; }
    }

    int pos = -1;

    // Реализация интерфейса IEnumerator:
    public bool MoveNext()
    {
        if (pos < students.Length - 1)
        {
            pos++;
            return true;
        }
        Reset();
```

```
        return false;
    }

    public void Reset()
    {
        pos = -1;
    }

    public object Current
    {
        get { return students[pos]; }
    }

    // Реализация интерфейса IEnumerable:
    public UserCollection GetEnumerator()
    {
        return this;
    }
}
```

После компиляции кода коллекция останется работоспособной, т. е. наличие строки

```
: IEnumerable, IEnumerator
```

в объявлении класса не обязательно. Главное, чтобы были реализованы методы для прохождения по коллекции с теми же именами. Более того, если попытаться закомментировать поочередно каждый из методов и выполнить программу, то видно, что метод Reset() тоже не является обязательным.

Предыдущие два примера дали нам представление о том, как внутри выглядят коллекции, на чем они базируются и какой минимальный инструментарий необходим для их работы. Кроме того, был продемонстрирован принцип функционирования оператора foreach, который в своей работе использует методы MoveNext(), Reset() и свойство Current.

Важно понимать, что настоящие коллекции содержат гораздо больше методов.

Рассмотрим создание пользовательских типизированных коллекций с обобщенными интерфейсами.

В C# предусмотрены типизированные интерфейсы IEnumerable<T> и IEnumerator<T>. Посмотрим на то, как они реализованы.

```
public interface IEnumerable<out T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
  T Current
    {
      get;
    }
}
```

Здесь видно, что, во-первых, они реализуют нетипизированные IEnumerator и IEnumerable, во-вторых, в интерфейсе IEnumerator<T> определено только свойство Current, а методы MoveNext() и Reset() нет и, в-третьих, интерфейс IEnumerator<T> реализует IDisposable.

Дело в том, что если класс реализует интерфейс, который, в свою очередь, реализует еще один интерфейс, то класс обязан реализовать всю цепочку интерфейсов. Методы MoveNext() и Reset() не зависят от типа поля, а свойство Current необходимо будет реализовать. Ниже в примере показан класс UserCoollection с обобщенными интерфейсами.

```
public class UserCollection<T> : IEnumerable<T>, IEnumerator<T>
{
  readonly T[] students = new T[4];

  public T this[int index]
  {
    get { return students[index]; }
    set { students[index] = value; }
  }

  int pos = -1;

  // IEnumerator<T>:
  bool IEnumerator.MoveNext()
  {
    if (pos < students.Length - 1)
    {
        pos++;
        return true;
    }
    return false;
  }

  void IEnumerator.Reset()
  {
    pos = -1;
    }

  object IEnumerator.Current
  {
    get { return students[pos]; }
    // Можно заменить предыдущую строку, например, на
    // get { return -10; }
    // и все будет работать
  }

  T IEnumerator<T>.Current {
    get { return students[pos]; }
  }

  // IEnumerable<T>:
  IEnumerator IEnumerable.GetEnumerator()
  {
    return this;
  }

  IEnumerator<T> IEnumerable<T>.GetEnumerator()
  {
    return this;
  }

  // IDisposable:
  void IDisposable.Dispose()
  {
    ((IEnumerator)this).Reset();
  }
}
```

Как видно из примера, методы MoveNext() и Reset() явно реализованы от интерфейса IEnumerator. По сути необходимо реализовать и обобщенный интерфейс, и необобщенный. Так, у нас два GetEnumerator(), два указателя Current и т. д., причем IEnumerator.Current ни разу не вызывается, что легко проверить, поставив любую константу в оператор return.

Вопросы для самоконтроля

1. В чем отличие пользовательских коллекций от стандартных?
2. Для чего нужен перечислитель в коллекции?
3. Как используются методы MoveNext() и Reset() оператором foreach при работе в коллекциях?
4. Какие интерфейсы необходимо реализовать при создании пользовательской коллекции?

ГЛАВА 19
ДЕЛЕГАТЫ И СОБЫТИЯ

С событиями мы сталкиваемся, когда, например, программируем графический пользовательский интерфейс. Так, во время взаимодействия с графическими элементами, происходит генерирования событий, на которые могут быть подписаны один и более методов-обработчиков. Событие может быть сгенерировано и без взаимодействия с пользователем.

> Событие – это именованный делегат, при вызове которого будут запущены все подписавшиеся на момент вызова события методы заданной сигнатуры.

Даже не подозревая об этом, большинство уже сталкивались с событиями в консольных приложениях. Рассмотрим процесс создания кнопки на окне приложения (рисунок 19.1). Разместим кнопку на форме, сфокусируем её (1), нам станет доступным просмотр событий активного элемента управления (2). Как видим, элемент управления «Кнопка» может генерировать множество событий (3), в том числе стандартное Click (4).

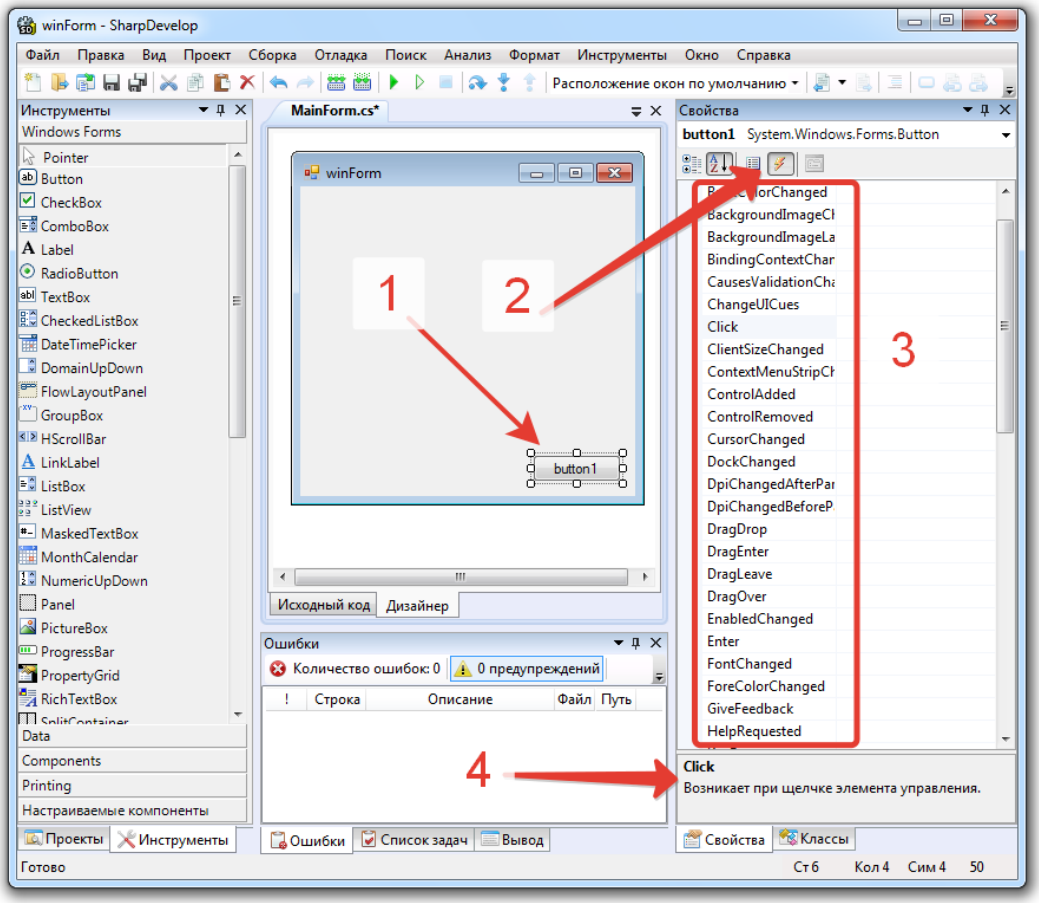


Рисунок 19.1 – События элемента управления «Кнопка»

Событие – это ситуация, при возникновении которой произойдет действие или несколько действий.

Для того чтобы создать обработчик стандартного события Click, достаточно дважды клинкнуть левой кнопкой мыши на элементе управления, и среда автоматически сгенерирует метод-обработчик события (рисунок 19.2).

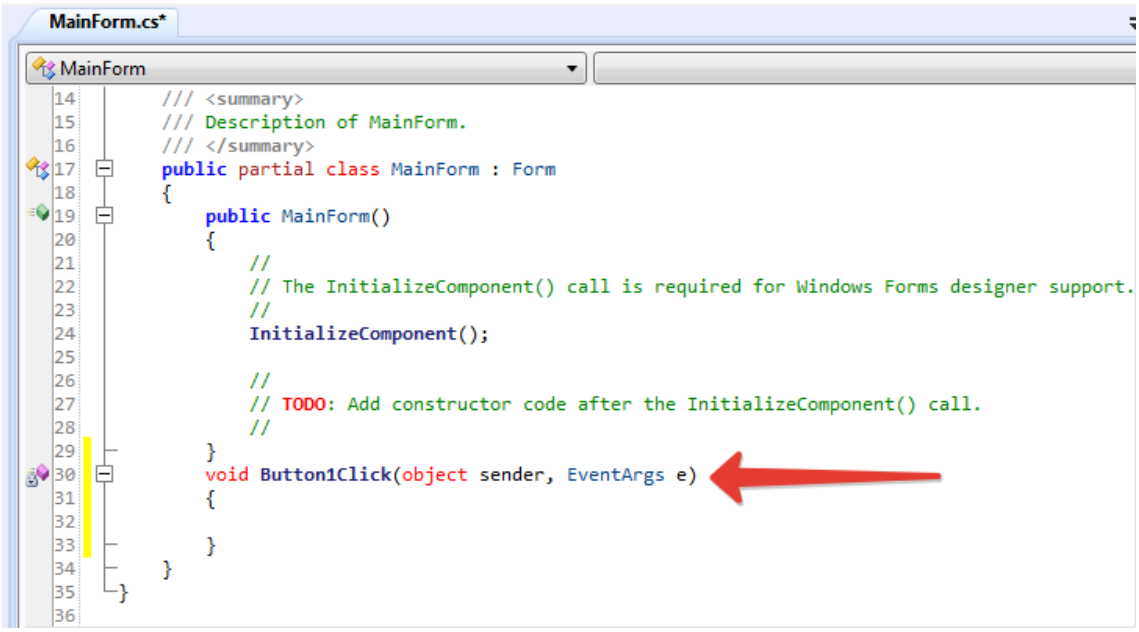


Рисунок 19.2 – Обработчик события

> Обработчики событий – это методы в объекте, привязанные к событию, возникающему в приложении.

Обобщая сказанное, можно поведать, что событие предоставляет возможность классу сообщать об изменениях, а пользователям, подписанным на события, реагировать.

События объявляются следующим образом:

```
public event EventHandler[<класс>] Имя;
```

event – ключевое слово, используемое для объявления события в классе [https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/event].

EventHandler – стандартный делегат, предоставляющий метод, обработчик события. Причем событие не генерирует никаких данных. Для их генерации нужно определить собственный класс или структуру и создать делегат, который в качестве второго аргумента будет содержать этот тип данных.

Класс – необязательный параметр, класс описывает передаваемые параметры.

Имя – имя события (правильный идентификатор).

События в C# базируются на понятии делегат, поэтому дальнейшее изучение событий без понимания делегатов невозможно.

Делегаты

Концепция делегатов была введена в Visual J++, а затем перенесена в C#. Делегаты C# являются в .NET Framework реализацией в качестве производного класса от System.Delegate.

> Делегаты – это именованные указатели на методы; относятся к ссылочным типам.

По своей сути делегаты идентичны указателям на функции в C++, однако делегаты полностью объектно-ориентированные, каждый делегат может ссылаться на методы, хранящиеся в списке вызовов делегата (invocation list); их вызов происходит последовательно.

Полезной особенностью делегата можно считать то, что делегат «не заботится» об экземплярах классов, инкапсулируемых им, при этом важным остается только совместимость этих методов с типом делегата, что делает делегаты подходящим инструментом для анонимного вызова.

Создание экземпляра делегата (объявление делегата) выглядит следующим образом:

```
[атрибуты] [модификаторы] delegate тип_возвращаемого значения;
```

Вторым способом создания делегата является преобразование в тип делегата.

Экземпляр делегата может ссылаться на:

– статический метод;
– другой делегат;
– целевой объект (не может быть null).

Какой угодно метод в делегат положить нельзя, только методы, совпадающие по сигнатуре с делегатом.

В следующем примере объявлены два типа делегата.

```
namespace TestDelegate
{
  public delegate int MyDelegate_1 (string s);

  public class A
    {
      public delegate void MyDelegate_2 (byte a, byte b);
    }
}
```

TestDelegate.MyDelegate_1 объявлен на уровне пространства имён, следовательно, он «совместим» с любым методом, который имеет один параметр типа string и возвращает значение типа int.

DelegateArticle.A.MyDelegate_2 (объявленный внутри класса) и является его членом. Он «совместим» с любым методом, который имеет два byte параметра и возвращает void.

Оба типа делегата имеют модификатор доступа public. По отношению модификаторов доступа типы делегатов ведут себя так же, как классы и структуры. Если для типа делегата явно не указан модификатор доступа и этот тип объявлен внутри пространства имён, то он будет доступен для всех объектов, также находящихся внутри этого пространства имён. Если же тип делегата без модификатора объявлен внутри класса или структуры, то он будет закрытым аналогично действию модификатора rivate.

При объявлении типа делегата нельзя использовать модификатор static.

Использование делегатов можно описать четырьмя шагами:

- определение объекта делегата с сигнатурой, точно соответствующей сигнатуре метода, который пытаемся связать;
- определение метода с сигнатурой делегата, определенного на первом шаге;
- создание объекта делегата и связывание их с методами;
- вызов связанного метода с помощью объекта делегата.

Следующий пример иллюстрирует вышеупомянутые шаги в реализации одного делегата и четырех классов.

```
using System;
// Шаг 1. Определение делегата с сигнатурой связываемого метода
public delegate void MyDelegate(string input);

// Шаг 2. Определение метода с сигнатурой определенного делегата
class A
{
  public void delegateM1(string input){
    Console.WriteLine("Это delegateM1 с параметром {0}", input);
  }
  public void delegateM2(string input){
    Console.WriteLine("Это delegateM2 с параметром {0}", input);
  }
}

// Шаг 3. Создание объектов делегата и связывание с методами
class B
{
  public MyDelegate createDelegate(){
    A oA=new A();
    MyDelegate d1 = new MyDelegate(oA.delegateMethod1);
    MyDelegate d2 = new MyDelegate(oA.delegateMethod2);
    MyDelegate d3 = d1 + d2;
    return d3;
  }
}

// Шаг 4. Вызов метода с помощью делегата
class C{
  public void callDelegate(MyDelegate d,string input){
    d(input);
  }
}

class Program
{
  static void Main(string[] args){
    B oB = new B();
    MyDelegate d = oB.createDelegate();
    C oC = new C();
    oC.callDelegate(d,"Вызов делегата...");
  }
}
```

В C# имеется возможность комбинировать делегаты, т. е. производить над ними операции объединения и вычитания, что позволит при вызове одного экземпляра делегата вызывать целый набор методов. Эти методы могут быть из различных экземпляров различных классов.

Принято считать, что экземпляр делегата хранит ссылки на метод и на экземпляр объекта, и это верно для тех случаев, когда экземпляр делегата представляет один метод. Однако существуют такие экземпляры делегатов, которые являются списками простых делегатов, все из которых основываются на одном типе делегата (т. е. имеют одинаковую сигнатуру методов, на которые ссылаются) и называются комбинированными делегатами. Более того, несколько комбинированных делегатов могут быть скомбинированы между собой, фактически становясь одним большим списком простых делегатов.

> Список простых делегатов в комбинированном делегате называется «списком вызовов» или «списком действий».

Следовательно, список вызовов – это список пар ссылок на методы и экземпляры объектов, которые (пары) расположены в порядке вызова.

Подобно работе со строками, когда при применении метода String.PadLeft к экземпляру строки метод не изменяет этот экземпляр, а возвращает новый, работает объединение и вычитание экземпляров делегатов. Отсюда можно сделать следующий вывод:

Экземпляры делегатов всегда неизменяемы (immutable).

Объединение (или «сложение») двух экземпляров делегатов, как правило, выполняется при помощи оператора сложения; аналогично вычитание («удаление») одного экземпляра делегата из другого производится при помощи оператора вычитания. Однако при вычитании одного комбинированного делегата из другого вычитание производится в рамках списка вызовов. Если в оригинальном (уменьшаемом) списке вызовов нет ни одного из тех простых делегатов, которые находятся в вычитаемом списке вызовов, то результатом операции (разностью) будет оригинальный список. В противном случае, если в оригинальном списке присутствуют простые делегаты, присутствующие и в вычитаемом, то в результирующем списке будут отсутствовать лишь последние вхождения простых делегатов.

В таблице ниже через d1, d2, d3 обозначены простые делегаты. Делегаты в прямоугольных скобках, например [d1, d2, d3], подразумевают комбинированный делегат, состоящий из нескольких простых, и именно в таком порядке, т. е. при вызове сначала будет вызван d1, потом d2, а затем d3. Пустой список вызовов представлен значением null (таблица 19.1).

Таблица 19.1 – Пример сложения и вычитания делегатов

Выражение	Результат
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] – d1	d2
[d1, d2] – d2	d1
[d1, d2, d1] – d1	[d1, d2]
[d1, d2, d3] – [d1, d2]	d3
[d1, d2, d3] – [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] – [d1, d2]	[d1, d2, d3]
[d1, d2] – [d1, d2]	null

Операции сложения и вычитания экземпляров делегатов могут быть выполнены при помощи статических методов Delegate.Combine и Delegate.Remove.

Операторы сложения и вычитания – это так называемый «синтаксический сахар», который в конечном итоге будет заменен компилятором C# на вызовы методов Combine и Remove.

Операторы сложения и вычитания работают как часть операции присваивания, и выражение d1 += d2 полностью эквивалентно выражению d1 = d1+d2.

Экземпляры делегатов, участвующие в сложении и вычитании, не изменяются в процессе операции.

Так, переменная d1 сменит ссылку на новый комбинированный делегат, состоящий из «старого» d1 и d2.

|| Добавление и удаление делегатов происходит с конца списка.

Последовательность вызовов (x + … = y; x – … = y) эквивалентна пустой операции, т. е. переменная x будет содержать неизменный список подписчиков.

Если сигнатура типа делегата объявлена такой, что возвращает значение (т. е. возвращаемое значение не является void) и «на основе» этого типа создан комбинированный экземпляр делегата, то при его вызове в переменную будет записано возвращаемое значение, «предоставленное» последним простым делегатом в списке вызовов комбинированного делегата. Если есть комбинированный делегат (содержащий список вызовов, состоящий из множества простых делегатов), и при его вызове в каком-то простом делегате произойдёт исключение, то в этом месте вызов комбинированного делегата прекратится, исключение будет проброшено, и все остальные простые делегаты из списка вызовов так никогда и не будут вызваны.

Вопросы для самоконтроля

1. Что такое событие?
2. Что такое обработчик события?
3. Как объявляются события в классе?
4. Как происходит подписывание методов-обработчиков на события?
5. Что такое делегаты?
6. Как выглядит объявление делегата?
7. Что такое список вызовов?
8. Возможно ли объединение/сложение делегатов?
9. Как происходит добавление или удаление делегатов?

ГЛАВА 20
СОБЫТИЯ

События чем-то напоминают свойства. Ранее было дано определение, что свойства – это специальный метод, состоящий из двух блоков (блок get и блок set). С событиями аналогичная ситуация. Выглядят события так же, как экземпляры делегатов (в смысле добавления и вычитания), но ими не являются. События, как и свойства, состоят из пары методов, связанных между собой так, что компилятор в них автоматически распознает не простые методы, а события. Эти методы соответствуют операциям добавления и удаления; каждая из них принимает по одному параметру с экземпляром делегата, который имеет тип, одинаковый с типом события.

При генерировании события происходит поочерёдный вызов обработчиков.

В C# вызов обработчиков события не является частью события.

Методы добавления и удаления вызываются следующим образом: имя_события + … = делегат и имя_события – … = делегат. Имя_события может быть указано по ссылке на экземпляр объекта (myForm.Click) или по названию типа (MyClass.SomeEvent), однако статические события используются редко. Объявление событий происходит двумя способами. Рассмотрим пример объявления событий с явной реализацией методов. Этот способ очень похож, как ранее говорилось, на объявления свойств.

В примере ниже методы add и remove не содержат никаких операций с экземплярами делегатов, которые туда передаются, а просто выводят в консоль сообщения. При выполнении примера можно заметить, что метод remove будет вызван, не взирая на то, что мы ему передали для удаления значение null.

```
using System;

class A
 {
     //публичное событие MyEvent с типом EventHandler
     public event EventHandler MyEvent
     {
         add
         {
             Console.WriteLine ("Добавление");
         }

         remove
         {
             Console.WriteLine ("Удаление");
         }
     }

     static void Main()
     {
         A oA = new A();

         oA.MyEvent += new EventHandler (oA.DoNothing);
         oA.MyEvent -= null;
     }

    //Метод, сигнатура которого совпадает с сигнатурой типа делегата
EventHandler
     void DoNothing (object sender, EventArgs e)
     {
     }
 }
```

Ситуации, при которых возникает необходимость игнорировать полученное значение value, возникают довольно редко. Однако бывают случаи, когда использование простой переменной делегата для содержания подписчиков не подходит, например, когда класс содержит множество событий, но подписчики будут использовать только некоторый из них, можно создать ассоциативный массив, где в качестве ключа будет использоваться описание события, а в качестве значения – сам делегат с его подписчиками. Такой прием используется в Windows Forms, где класс может содержать много событий без напрасного использования памяти под переменные, которые в большинстве случаев будут равными null.

20.1 Field-like события

Язык C# предоставляет простой способ объявления переменной делегата и события в одну строку, такой способ называется «field-like событием» (field-like event) и объявляется аналогично «длинной» форме, но без «тела» с методами add и remove.

```
public event EventHandler MyEvent;
```

Здесь происходит создание переменной делегата и события с одинаковым типом. Доступ к событию определяется в объявлении события с помощью модификатора доступа (т. е. в примере выше публичное событие), однако переменная делегата всегда приватна. Неявное (implicit) тело события разворачивается компилятором во вполне очевидные операции добавления и удаления экземпляров делегата к/из переменной делегата, и эти действия выполняются под блокировкой (lock).

Для C# V1.1 событие MyEvent из примера выше эквивалентно следующему коду:

```
private EventHandler _myEvent;

public event EventHandler MyEvent
 {
     add
     {
         lock (this)
         {
             _myEvent += value;
         }
     }
     remove
     {
         lock (this)
         {
             _myEvent -= value;
         }
     }
 }
```

Когда в коде происходит ссылка на MyEvent внутри тела самого типа (включая вложенные типы) компилятор генерирует код, который ссылается на переменную делегата (_myEvent). Во всех остальных контекстах компилятор генерирует код, который ссылается на событие.

20.2 Событийная модель

Событийная модель очень хорошо подходит для моделирования реальных жизненных ситуаций (например, для моделирования потока управления датчиками в системе автопилотирования транспорта, где все дорожные примитивы взаимодействуют между собой путем генерирования событий и реакции на них). Рассмотрим пример, в котором есть

два класса A и B. Эти классы взаимодействуют друг с другом, используя инструмент события.

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

namespace EventExample
{
    public delegate void MyDelegate(DateTime date, byte
WorkTime);//Определяем делегат

    class Program
    {
        public class A
        {
            // Объявляем событие от класса A к классу B
            public event MyDelegate EventFromAToB;

            /// <summary>
            /// Главный метод класса.
            /// Задача метода - сгенерировать событие и передать ему заданные
аргументы.
            /// </summary>
            /// <param name="date">Передается текущий такт. В данном примере
тактами выступает месяц переменной ДатаВремя</param>
            /// <param name="WorkTime">Параметр задает количество циклов (за-
держку), которое необходимо методу для работы. Параметр генерируется случай-
ным образом в управляющем модуле и передается в обработчик события через Mian
метод</param>
            public void MainA(DateTime date, byte WorkTime)
            {
                /// <summary>
                /// Генерирование события классом
                /// </summary>
                /// <param name="date">Передается текущий такт. В данном при-
мере тактами выступает месяц переменной ДатаВремя</param>
                /// <param name="WorkTime">Параметр задает количество циклов
(задержку), которое необходимо методу для работы.
                /// Параметр генерируется случайнам образом в управляющем мо-
дуле и передается в обработчик события через Mian метод</param>
                EventFromAToB(date, WorkTime);
            }

            /// <summary>
            /// Метод обработчик события от класса B
            /// </summary>
            /// <param name="date">Принимает такт (параметр типа ДатаВре-
мя)</param>
            /// <param name="WorkTime">Время работы обработчика</param>
            public void RaiseEventFromBToA(DateTime date, byte WorkTime)
            {
                /// Заданное время обработчик будет "выполнять работу"
                /// Работой обрабочика является вывод на экран
                /// линии жизни объекта.
                for (int i = 1; i <= WorkTime; i++)
                    Console.WriteLine("\t{0}\t{1}\t{2}", date.Day, "A", "|");
            }
        }
```

```
        public class B
        {
            // Объявляем событие от класса B к классу A
            public event MyDelegate EventFromBToA;

            /// <summary>
            /// Главный метод класса.
            /// Задача метода - сгенерировать событие и передать ему заданные
аргументы.
            /// </summary>
            /// <param name="date">Передается текущий такт. В данном примере
тактами выступает месяц переменной ДатаВремя</param>
            /// <param name="WorkTime">Параметр задает количество циклов (за-
держку), которое необходимо методу для работы. Параметр генерируется случай-
ным образом в управляющем модуле и передается в обработчик события через Mian
метод</param>
            public void MainB(DateTime date, byte WorkTime)
            {
                /// <summary>
                /// Генерирование события классом
                /// </summary>
                /// <param name="date">Передается текущий такт. В данном при-
мере тактами выступает месяц переменной ДатаВремя</param>
                /// <param name="WorkTime">Параметр задает количество циклов
(задержку), которое необходимо методу для работы.
                /// Параметр генерируется случайным образом в управляющем мо-
дуле и передается в обработчик события через Mian метод</param>
                EventFromBToA(date, WorkTime);
            }

            /// <summary>
            /// Метод обработчик события от класса A
            /// </summary>
            /// <param name="date">Принимает такт (параметр типа ДатаВре-
мя)</param>
            /// <param name="WorkTime">Время работы обработчика</param>
            public void RaiseEventFormAtoB(DateTime date, byte WorkTime)
            {
                /// Заданное время обработчик будет "выполнять работу"
                /// Работой обрабочика является вывод на экран
                /// линии жизни объекта
                for (int i = 1; i <= WorkTime; i++)
                    Console.WriteLine("\t{0}\t{1}\t{2}", date.Day, "|", "B");
            }

        }
        static void Main(string[] args)
        {
            /// Счетчик тактов
            byte i = 0;
            /// Переменная хранит время работы объекта (метода)
            byte WorkTime = 1;
            /// Создадим экземпляр объекта date от класса DateTime
            DateTime date = new DateTime();
            /// Создадим экземпляр объекта rnd от класса Random
            Random rnd = new Random();
            /// Создадим экземпляр объекта oA от класса A
            A oA = new A();
            /// Создадим экземпляр объекта oB от класса B
            B oB = new B();
            /// Подпишемся на событие
            /// экземпляр.Событие += экземпляр.Обработчик
```

```
            oA.EventFromAToB += oB.RaiseEventFormAtoB;
            oB.EventFromBToA += oA.RaiseEventFromBToA;

            /// Выведем шапку диаграммы последовательности
            Console.WriteLine("\t{0}\t{1}\t{2}","такт", "A", "B");
            Console.WriteLine("\t{0}\t{1}\t{2}", "____","|", "|");

            /// Заданное количество тактов ...
            while (i < 5)
            {
                /// Сгенерируем случайное число тактов, которое должен будет
               /// отработатьобъект. Число генерируется в диапазоне от 1 до 9
                WorkTime = (byte)rnd.Next(1, 9);

                /// увеличим текущую дату на i дней
                date = DateTime.Today.AddDays(i);
                /// Вызовем методы Main объектов, содержащих обработчики со-
бытий и генераторы событий
                oB.MainB(date, WorkTime);
                oA.MainA(date, WorkTime);
                /// Декоративная пауза между периодами (тактами)
                Thread.Sleep(1000);
                i++;
            }
            Console.ReadKey();
        }
    }
}
```

В этом примере управляющий модуль (Main-метод класса Program) определяет, какой объект на какой метод будет реагировать путем подписывания обработчиков событий.

```
/// Подпишемся на событие
/// экземпляр.Событие += экземпляр.Обработчик
oA.EventFromAToB += oB.RaiseEventFormAtoB;
oB.EventFromBToA += oA.RaiseEventFromBToA;
```

Далее в цикле, где шаг цикла, – новый день, происходит вызов Main-методов экземпляров классов.

```
oB.MainB(date, WorkTime);
oA.MainA(date, WorkTime);
```

А Main-методы генерируют события:

```
EventFromAToB(date, WorkTime);
```

Реакцией на события будет обработчик вида:

```
public void RaiseEventFromBToA(DateTime date, byte WorkTime)
{
   /// Заданное время обработчик будет "выполнять работу"
   /// Работой обрабочика является вывод на экран
   /// линии жизни объекта
   for (int i = 1; i <= WorkTime; i++)
      Console.WriteLine("\t{0}\t{1}\t{2}", date.Day, "A", "|");
}
```

В результате своей работы строится UML-диаграмма последовательности:

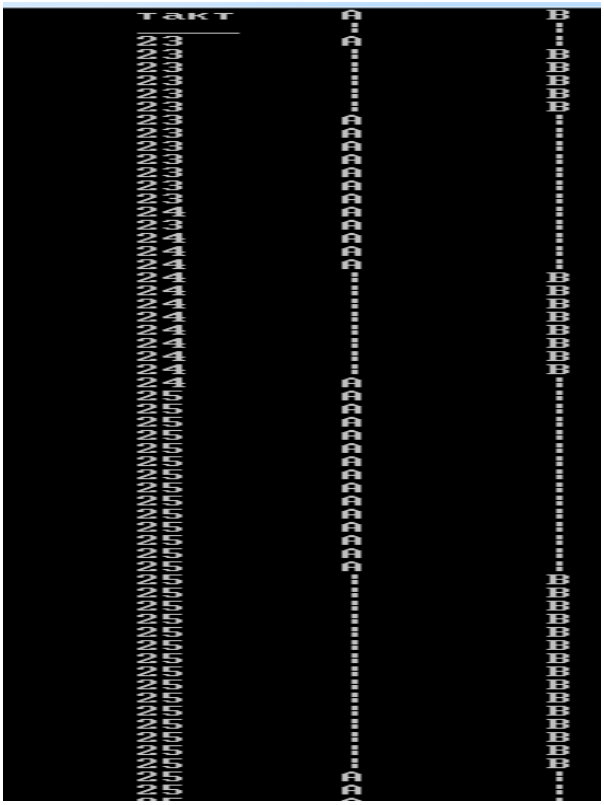


Рисунок 20.1 – Результат работы программы

Дальнейшей задачей является задача распараллеливания потоков. В объективной реальности обработчики событий должны работать и работают параллельно. Для реализации параллельности, которая будет рассмотрена в других разделах, используется пространство имен System.Threading.

Вопросы для самоконтроля

1. Как происходит подписывание на событие?
2. Что такое Field-like события?
3. В чем смысл событийной модели поведения объектов?

ГЛАВА 21
СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ

.NET Framework содержит несколько средств, используя которые можно выполнять сериализацию и десериализацию:

1. XmlSerializer находится в пространстве имен System.Xml.Serialization, имеет следующие основные характеристики:

1.1. Позволяет сериализовать как текстовые, так и двоичные данные в формат XML;

1.2. Сериализует объекты-потомки, но без поддержки циклических ссылок;

1.3. Выполняет сериализацию только public полей и свойств, кроме случаев, когда их сериализация явно запрещена;

1.4. Позволяет настраивать схему XML.

2. BinaryFormatter находится в пространстве имен System.Runtime.Serialization.Formatters. Binary:

2.1. Выполняет сериализацию в специальный закрытый формат, который распознается только классом BinaryFormatter;

2.2. Выполняет сериализацию только public полей и свойств;

2.3. В отличие от XML работает с циклическими ссылками;

2.4. Не требует предварительного знания типов сериализуемых объектов;

2.5. Объекты, подлежащие сериализации, должны быть помечены атрибутом [Serializable].

3. SoapFormatter находится в пространстве имен System.Runtime.Serialization.Formatters. Soap:

3.1. Практически аналогичен формату Binary, но является более подходящим для обмена данными с другими системами, менее компактный;

3.2. Не поддерживает обобщенные типы и обобщенные коллекции.

4. DataContractSerializer находится в пространстве имен System.Runtime.Serialization:

4.1. Выполняет сериализацию в формат XML;

4.2. Используется механизмом WCF, но может использоваться как самостоятельный инструмент сериализации;

4.3. Выполняет сериализацию типов и полей, отмеченных атрибутами [DataContract] и [DataMember];

4.4. Класс, отмеченный атрибутом [Serializable], будет сериализован полностью.

5. NetDataContractSerializer находится в пространстве имен System.Runtime.Serializatio:

5.1. Аналогичен DataContractSerializer, но встраивает в сериализованные данные информацию о типе;

5.2. Не требует предварительного знания типов сериализуемых объектов.

6. DataContractJsonSerializer из пространства имен System.Runtime.Serialization:

6.1. Аналогичен DataContractSerializer, выполняет сериализацию в формат JSON.

> Сериализация – это процесс сохранения состояния объекта путем преобразования объекта в поток данных.
> Десериализация – это процесс восстановления состояния объекта из потока данных.

Сериализация выполняется для сохранения или передачи в память или файл (базу данных) объекта. При сериализации объект направляется (сериализируется) в поток (stream) вместе с данными и определенной информацией о себе, например версия и имя сборки.

Сериализация позволяет не только сохранять объекты на диске и восстанавливать при необходимости, но и обеспечивает возможность организовать обмен данными между

приложениями (например, через web), обеспечивать защиту данных пользователей в приложениях и т. д.

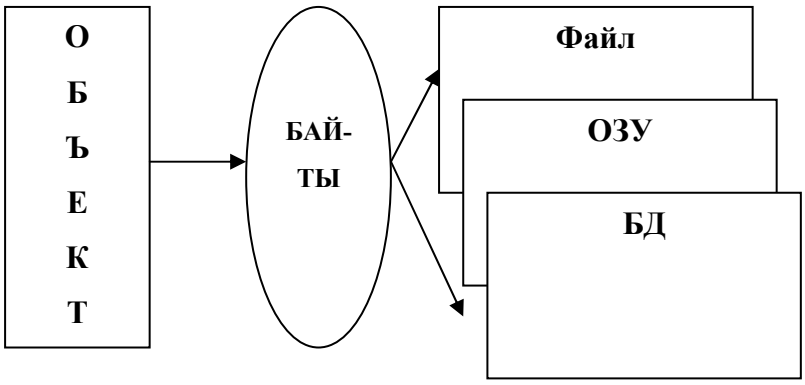


Рисунок 21.1 – Схема сериализации

Можно провести аналогию на некотором абстрактном примере. Предположим, мы занимаемся разработкой программного обеспечения для обеспечения работоспособности интеллектуального транспорта. Программа отлажена и успешно работает на объектах системы интеллектуального транспорта. Но вышла новая версия аппаратной части, и появились новые возможности, естественно, в программном обеспечении это доработали, теперь возникла задача обновить ПО. Однако остановить всю систему для обновления нет возможности. Нужно выполнить это «на лету», не останавливая систему. Для этого очень удобно использовать сериализацию и десериализацию, которая позволяет восстановить состояние объекта, а в нашем случае восстановление будет выглядеть следующим образом: на рабочем объекте будет вначале протестировано все ПО, произведена отладка, а потом уже это состояние объекта будет «развернуто» на реальном работающем объекте без его остановки. Грубо сериализацией можно назвать процесс гибернетизации в ОС Windows.

Для выполнения сериализации необходимо наличие объекта, потока, в который будет выполнена сериализация, и класс Formatter. Рассмотрим простой пример, выполняющий сериализацию и десериализацию объекта в XML формате.

```
using System;
using System.IO;
using System.Xml.Serialization;

namespace CollectionSort
{
    [Serializable]
    public class Student
    {
        public string Name {get; set;}
        public int Age {get; set;}
        public Student()
        {}

        public Student(string name, int age)
        {
            this.Name = name;
            this.Age = age;
        }

        public new string ToString()
        {
```

```
            return Name + " " + Age;
        }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Сериализация-Десериализация");

            Student aStudent = new Student("Первый",20);

            XmlSerializer xml = new XmlSerializer(aStudent.GetType());
            FileStream f = new FileStream("serialization.xml",
FileMode.Create,FileAccess.ReadWrite, FileShare.Read);
            xml.Serialize(f, aStudent);

            // позицию сместить до создания XmlTextReader
            f.Position = 0;
            f.Seek(0, SeekOrigin.Begin);

            Student st = xml.Deserialize(f) as Student;
            Console.WriteLine(aStudent.ToString());

            Console.WriteLine("Нажми клавишу для выхода . . . ");
            Console.ReadKey(true);
        }
    }
}
```

Подключение пространства имен System.IO вызвано необходимостью работы с файловым потоком и использованием класса FileStream. Пространство имен System.Xml.Serialization необходимо для выполнения сериализации с использованием XmlSerializer.

В примере создан класс Student и помечен как [Serializable]. Вторым условием для XML-сериализации является обязательное наличие конструктора без параметров.

```
public Student()  {}
```

В Main() методе создается экземпляр класса Student:

```
Student aStudent = new Student("Первый",20);
```

и выполняется его сериализация в файловый поток:

```
XmlSerializer xml = new XmlSerializer(aStudent.GetType());
FileStream f = new FileStream("serialization.xml",
FileMode.Create,FileAccess.ReadWrite, FileShare.Read);
xml.Serialize(f, aStudent);
```

После выполнения этого участка кода в файле serialization.xml имеем:

```
<?xml version="1.0" ?>
- <Student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<Name>Первый</Name>
<Age>20</Age>
</Student>
```

Дальше необходимо выполнить позиционирование указателя в файле на начало файла XML. Без этого при попытке выполнить десериализацию будет вызвано исключение:

```
System.InvalidOperationException: В документе XML (0, 0) присутствует ошибка.
---> System.Xml.XmlException: отсутствует корневой элемент.
```

После позиционирования указателя можно выполнять десериализацию объекта из файлового потока. Следует обратить внимание на то, что метод Deserialize() возвращает тип object, поэтому обязательно нужно выполнять приведение типов.

```
Student st = xml.Deserialize(f) as Student;
```

При наследовании классов, помеченых как [Serializable], необходимо учитывать, что атрибут [Serializable] автоматически не наследуется, следовательно, если есть необходимость сериализовать производный класс, то он тоже должен быть помечен как [Serializable].

Еще один пример иллюстрирует возможность сериализации коллекций. В примере создается коллекция из нескольких элементов, выполняется сериализация их в файловый поток, затем производится изменение коллекции (удаление одного из элементов) и десериализация, т. е. восстановление ее из файлового потока.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

namespace CollectionSort
{
    [Serializable]
    public class Student
    {
        public string Name {get; set;}
        public int Age {get; set;}

        public string Temp;

        public Student()
        {}

        public Student(string name, int age)
        {
            this.Name = name;
            this.Age = age;
            this.Temp = "строка";
        }

        public new string ToString()
        {
            return Name + " " + Age;
        }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Сериализация-Десериализация\n\n");
```

```
            List<Student> aStudent = new List<Student>();

            aStudent.Add(new Student("Ваня", 20));
            aStudent.Add(new Student("Коля", 21));

            // Настроим XML сериализацию и приготовим файловый поток
            XmlSerializer xml = new XmlSerializer(aStudent.GetType());
            FileStream f = new FileStream("serialization.xml",
FileMode.Create,FileAccess.ReadWrite, FileShare.Read);
            // Выполним сериализацию коллекции в файловый поток
            xml.Serialize(f, aStudent);

            foreach(Student i in aStudent)
            {
                Console.WriteLine(i.ToString());
            }
            Console.WriteLine("------------------------------");
            aStudent.RemoveAt(0);
            foreach(Student i in aStudent)
            {
                Console.WriteLine(i.ToString());
            }
            Console.WriteLine("------------------------------");
            // позицию сместить до создания XmlTextReader
            f.Position = 0;
            f.Seek(0, SeekOrigin.Begin);

            // Выполним десериализацию из файлового потока
            aStudent = (List<Student>)xml.Deserialize(f);

            foreach(Student i in aStudent)
            {
                Console.WriteLine(i.ToString());
            }

            Console.WriteLine("\n\nНажми клавишу для выхода . . . ");
            Console.ReadKey(true);
        }
    }
}
```

Существует возможность выполнять настраиваемую сериализацию, при которой некоторые члены класса не будут сериализированы. В примере ниже вместо коллекции используется массив, т. к. SOAP не позволяет работать со стандартными коллекциями. Поле Temp помечено как [NonSerialized]. Это приводит к тому, что поле не подлежит сериализации и десериализации.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

namespace SerializationTuning
{
    [Serializable]
    public class Student
    {

        public string Name {get; set;}
        public int Age {get; set;}
        [NonSerialized]
```

```
        public string Temp;

        public Student()
        {}

        public Student(string name, int age)
        {
            this.Name = name;
            this.Age = age;
            this.Temp = "строка";
        }
        public void M()
        {
            Console.WriteLine("Некий метод");
        }
        public new string ToString()
        {
            return Name + " " + Age +"\n" + Temp;
        }
    }

    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Сериализация-Десериализация\n\n");

            Student aStudent = new Student("Ваня", 20);
            Student bStudent = new Student("Коля", 21);
            Student[] students= new Student[] { aStudent, bStudent};

            // Настроим SOAP сериализацию
            SoapFormatter soap = new SoapFormatter();
            // Получаем файловый поток
            FileStream f = new FileStream("serialization.soap",
FileMode.Create,FileAccess.ReadWrite, FileShare.Read);
            // Выполним сериализацию коллекции в файловый поток
            soap.Serialize(f, students);

            // отобразим исходный массив
            foreach(Student i in students)
            {
                Console.WriteLine(i.ToString());
            }
            Console.WriteLine("------------------------------");

            // "испортим" массив
            students[0].Name = "Пушкин";
            students[0].Age = 999;
            students[0].Temp = "-----";
            // отобразим "испорченный" массив
            foreach(Student i in students)
            {
                Console.WriteLine(i.ToString());
            }
            Console.WriteLine("------------------------------");

            // позицию сместить в начало
            f.Position = 0;
            f.Seek(0, SeekOrigin.Begin);

            // Выполним десериализацию из файлового потока
```

```
            students = (Student[])soap.Deserialize(f);

            // Отобразим массив после десериализации
            foreach(Student i in students)
            {
                Console.WriteLine(i.ToString());
            }

            Console.WriteLine("\n\nНажми клавишу для выхода . . . ");
            Console.ReadKey(true);
        }
    }
}
```

В результате работы примера будет получен следующий файл serialization.soap

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<SOAP-ENC:Array SOAP-ENC:arrayType="a1:Student[2]"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SerializationTuning/Serial
izationTuning%2C%20Version%3D1.0.6920.23321%2C%20Culture%3Dneutral%2C%20Publi
cKeyToken%3Dnull">
<item href="#ref-3"/>
<item href="#ref-4"/>
</SOAP-ENC:Array>
<a1:Student id="ref-3"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SerializationTuning/Serial
izationTuning%2C%20Version%3D1.0.6920.23321%2C%20Culture%3Dneutral%2C%20Publi
cKeyToken%3Dnull">
<_x003C_Name_x003E_k__BackingField id="ref-
5">P’P°PSCЦ</_x003C_Name_x003E_k__BackingField>
<_x003C_Age_x003E_k__BackingField>20</_x003C_Age_x003E_k__BackingField>
</a1:Student>
<a1:Student id="ref-4"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SerializationTuning/Serial
izationTuning%2C%20Version%3D1.0.6920.23321%2C%20Culture%3Dneutral%2C%20Publi
cKeyToken%3Dnull">
<_x003C_Name_x003E_k__BackingField id="ref-
6">РьPsP»СЦ</_x003C_Name_x003E_k__BackingField>
<_x003C_Age_x003E_k__BackingField>21</_x003C_Age_x003E_k__BackingField>
</a1:Student>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

А в консоле результат будет выглядеть следующим образом:

```
Сериализация-Десериализация


Ваня 20
строка
Коля 21
строка
------------------------------
Пушкин 999
-----
```

```
Коля 21
строка
------------------------------
Ваня 20

Коля 21



Нажми клавишу для выхода . . .
```

Из результата работы примера видно, что поле Temp было проинициализировано конструктором, отображено, далее один из элементов массива был изменен (в том числе и поле Temp), а при восстановлении (десериализации) объекта поле Temp, помеченное как [NonSerialized], вообще осталось пустым.

Процесс сериализации и десериализации контроллируем. Контроль выполняется при помощи таких атрибутов, как OnDeserializedAttribute, OnDeserializingAttribute, OnSerializedAttribute, OnSerializingAttribute, которые добавляются к методам в классе и выполняются в момент сериализации/десериализации, по окончании сериализации/десериализации; в качестве аргумента методы должны принимать параметр StreamingContext.

Следующий пример иллюстрирует их применение:

```
[Serializable]
public class Student
{
  public int Age { get; set; }
  public string Name { get; set; }
  [NonSerialized]
  private int Kurs = 1;

  [OnSerializing()]
  internal void OnSerializing(StreamingContext context)
  {
    Console.WriteLine("Во время сериализации");
  }
  [OnSerialized()]
  internal void OnSerialized(StreamingContext context)
  {
    Console.WriteLine("По окончании сериализации");
  }
  [OnDeserializing()]
  internal void OnDeserializing(StreamingContext context)
  {
    Console.WriteLine("Во время десериализации");
  }
  [OnDeserialized()]
  internal void OnDeserialized(StreamingContext context)
  {
    Console.WriteLine("По окончании десериализации");
  }
}
```

Вопросы для самоконтроля

1. Что такое сериализация и десериализация?
2. Как происходит настраивание сериализации?
3. Контроль процесса сериализации и десериализации.

ГЛАВА 22
ДИАГРАММЫ UML

Все диаграммы UML можно условно разбить на две группы, первая из которых – общие диаграммы. Общие диаграммы практически не зависят от предмета моделирования и могут применяться в любом программном проекте без оглядки на предметную область, область решений и т. д.

22.1 Диаграмма использования

> Диаграмма использования (use case diagram) – это наиболее общее представление функционального назначения системы.

Диаграмма использования призвана ответить на главный вопрос моделирования: что делает система во внешнем мире?

На диаграмме использования применяются два типа основных сущностей: варианты использования и прецеденты, между которыми устанавливаются следующие основные типы отношений:

- ассоциация между действующим лицом и вариантом использования;
- обобщение между действующими лицами;
- обобщение между вариантами использования;
- зависимости (различных типов) между вариантами использования.

На диаграмме использования, как и на любой другой, могут присутствовать комментарии. Более того, это настоятельно рекомендуется делать для улучшения читаемости диаграмм (рисунок 22.1).

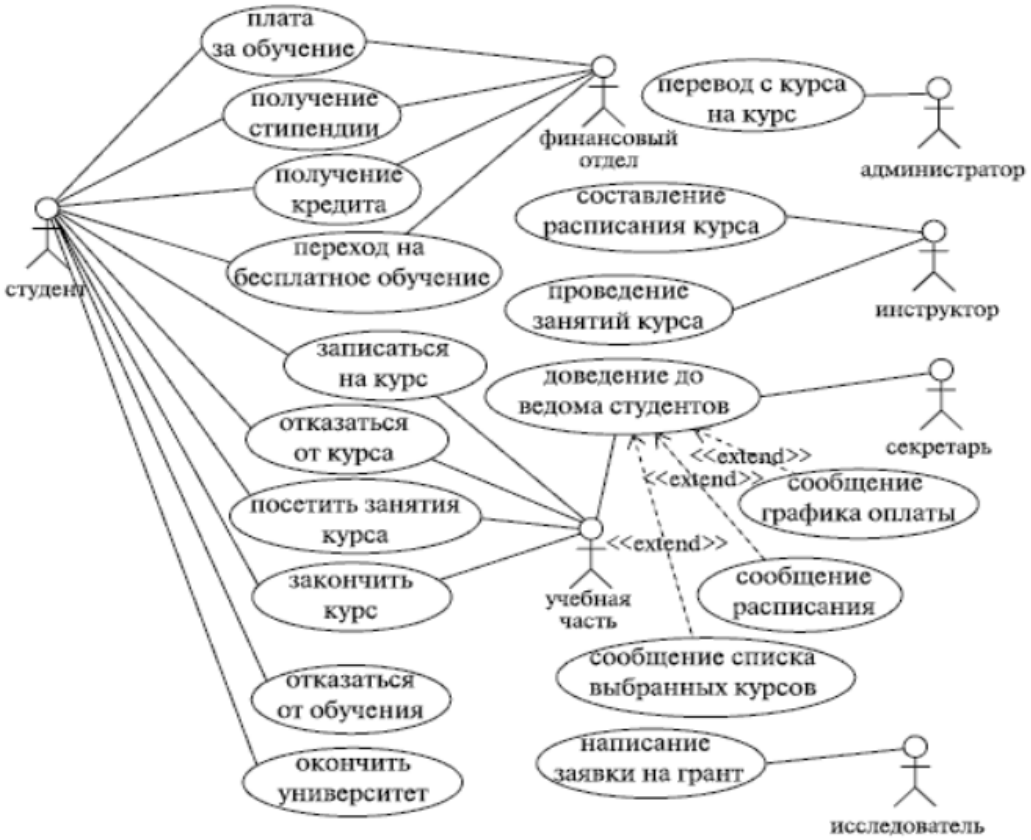


Рисунок 22.1 – Диаграмма вариантов использования автоматизированной системы «Студент»

22.2 Диаграмма классов

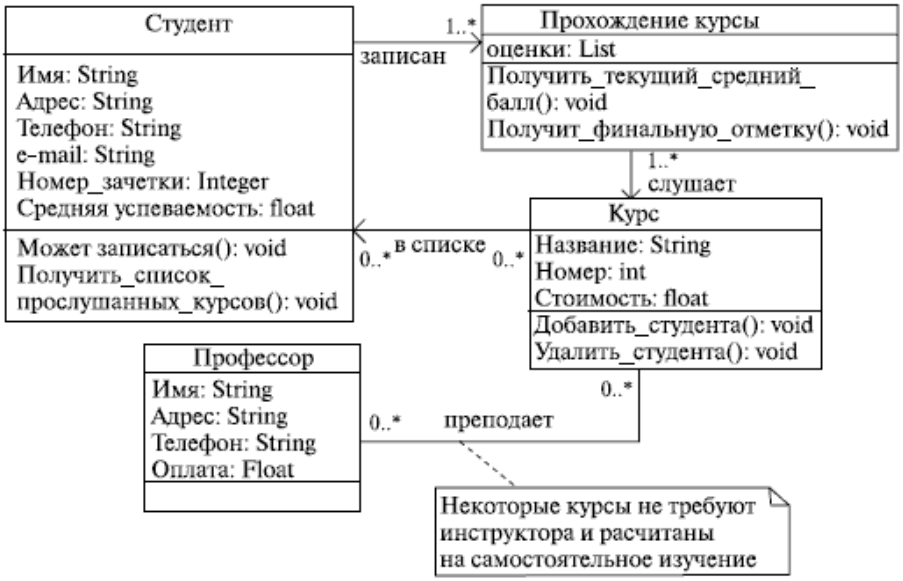
> Диаграмма классов (class diagram) – основной способ описания структуры системы.

Это не удивительно, поскольку UML в первую очередь объектно-ориентированный язык, и классы являются основным (если не единственным) «строительным материалом».

На диаграмме классов применяется один основной тип сущностей: классы (включая многочисленные частные случаи классов: интерфейсы, примитивные типы, классы-ассоциации и многие другие), между которыми устанавливаются следующие основные типы отношений (рисунок 22.2–22.3):

- ассоциация между классами (с множеством дополнительных подробностей);
- обобщение между классами;
- зависимости (различных типов) между классами;
- между классами и интерфейсами.

Рисунок 22.2 – Иерархия классов и взаимодействие между классами



Свойства и методы классов

Рисунок 22.3 – Свойства и методы классов

22.3 Диаграмма состояний

> Диаграмма состояний (statechart diagram) – это один из способов детального описания поведения в UML на основе явного выделения состояний и описания переходов между состояниями.

В сущности, диаграммы состояний представляют собой граф переходов состояний, нагруженный множеством дополнительных деталей и подробностей.

На диаграмме состояний применяют один основной тип сущностей – состояния, один тип отношений – переходы, но и для тех и для других определено множество разновидностей, специальных случаев и дополнительных обозначений. Перечислять их все во вступительном обзоре не имеет смысла (рисунок 22.4–22.5).

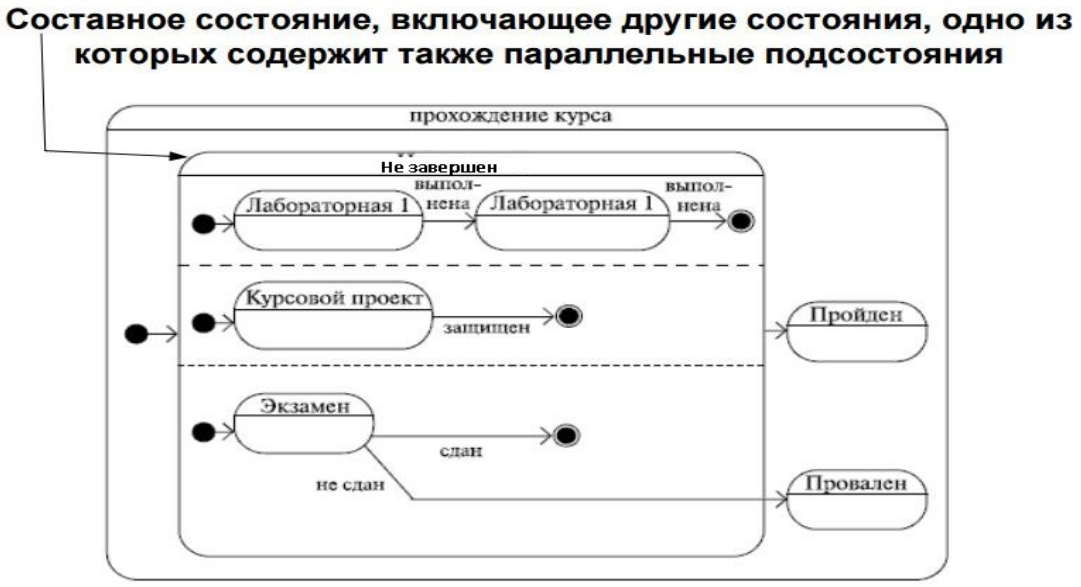


Рисунок 22.4 – Составное состояние системы

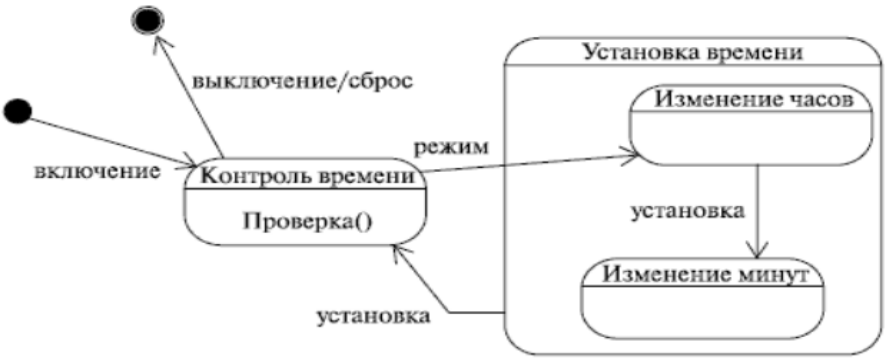


Рисунок 22.5 – Состояние системы

22.4 Диаграмма деятельности (активности)

Диаграмма деятельности (activity diagram) – способ описания поведения на основе указания потоков управления и потоков данных.

Диаграмма деятельности – еще один способ описания поведения, который визуально напоминает старую добрую блок-схему алгоритма. Однако за счет модернизированных обозначений, согласованных с объектно-ориентированным подходом, а главное, за счет новой семантической составляющей (свободная интерпретация сетей Петри), диаграмма деятельности UML является мощным средством для описания поведения системы.

На диаграмме деятельности применяют один основной тип сущностей – действие, а также один тип отношений – переходы (передачи управления и данных). Также используются такие конструкции, как развилки, слияния, соединения, ветвления, которые похожи на сущности, но таковыми на самом деле не являются, а представляют собой графический способ изображения некоторых частных случаев многоместных отношений (рисунок 22.6–22.7).

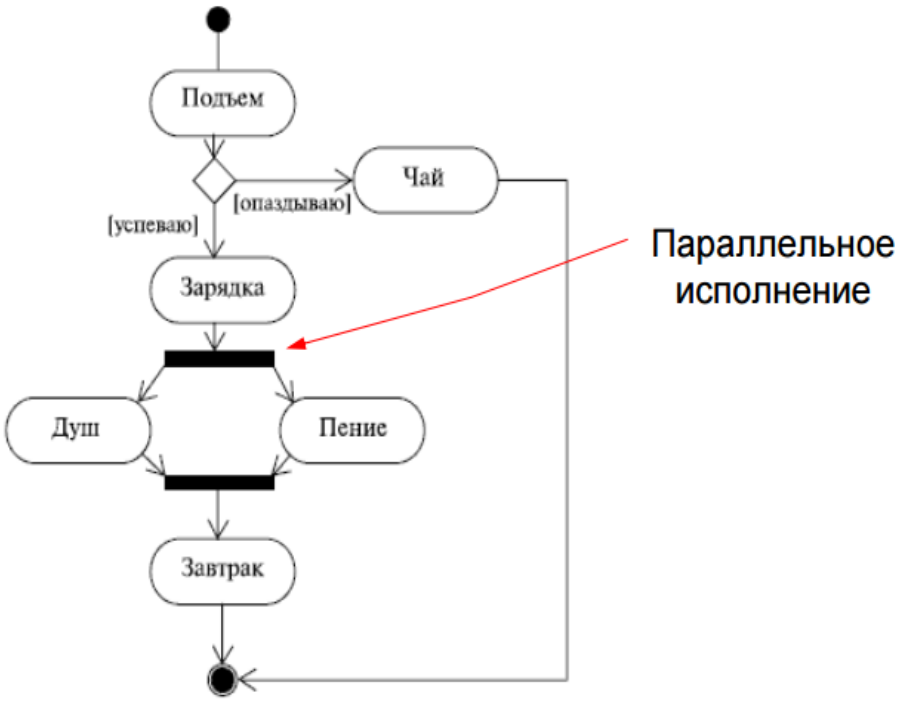


Рисунок 22.6 – Диаграмма деятельности системы с элементами параллелизма

Рисунок 22.7 – Диаграмма деятельности системы

22.5 Диаграмма последовательности

> Диаграмма последовательности (sequence diagram) – это способ описания поведения системы на основе указания последовательности передаваемых сообщений.

Фактически диаграмма последовательности – это запись протокола конкретного сеанса работы системы (или фрагмента такого протокола). В объектно-ориентированном программировании самым существенным во время выполнения является пересылка сообщений между взаимодействующими объектами. Именно последовательность посылок сообщений отображается на данной диаграмме, отсюда и название.

На диаграмме последовательности применяют один основной тип сущностей – экземпляры взаимодействующих классификаторов (в основном классов, компонентов и действующих лиц), а также один тип отношений – связи, по которым происходит обмен сообщениями. Предусмотрено несколько способов посылки сообщений, которые в графической нотации различаются видом стрелки, соответствующей отношению.

Важным аспектом диаграммы последовательности является явное отображение течения времени. В отличие от других типов диаграмм, кроме разве что диаграмм синхронизации, на диаграмме последовательности имеет значение не только наличие графических связей между элементами, но и взаимное расположение элементов на диаграмме. Считается, что имеется (невидимая) ось времени, по умолчанию направленная сверху вниз, и то сообщение, которое отправлено позже, нарисовано ниже.

Ось времени может быть направлена горизонтально, в этом случае считается, что время течёт слева направо (рисунок 22.8).

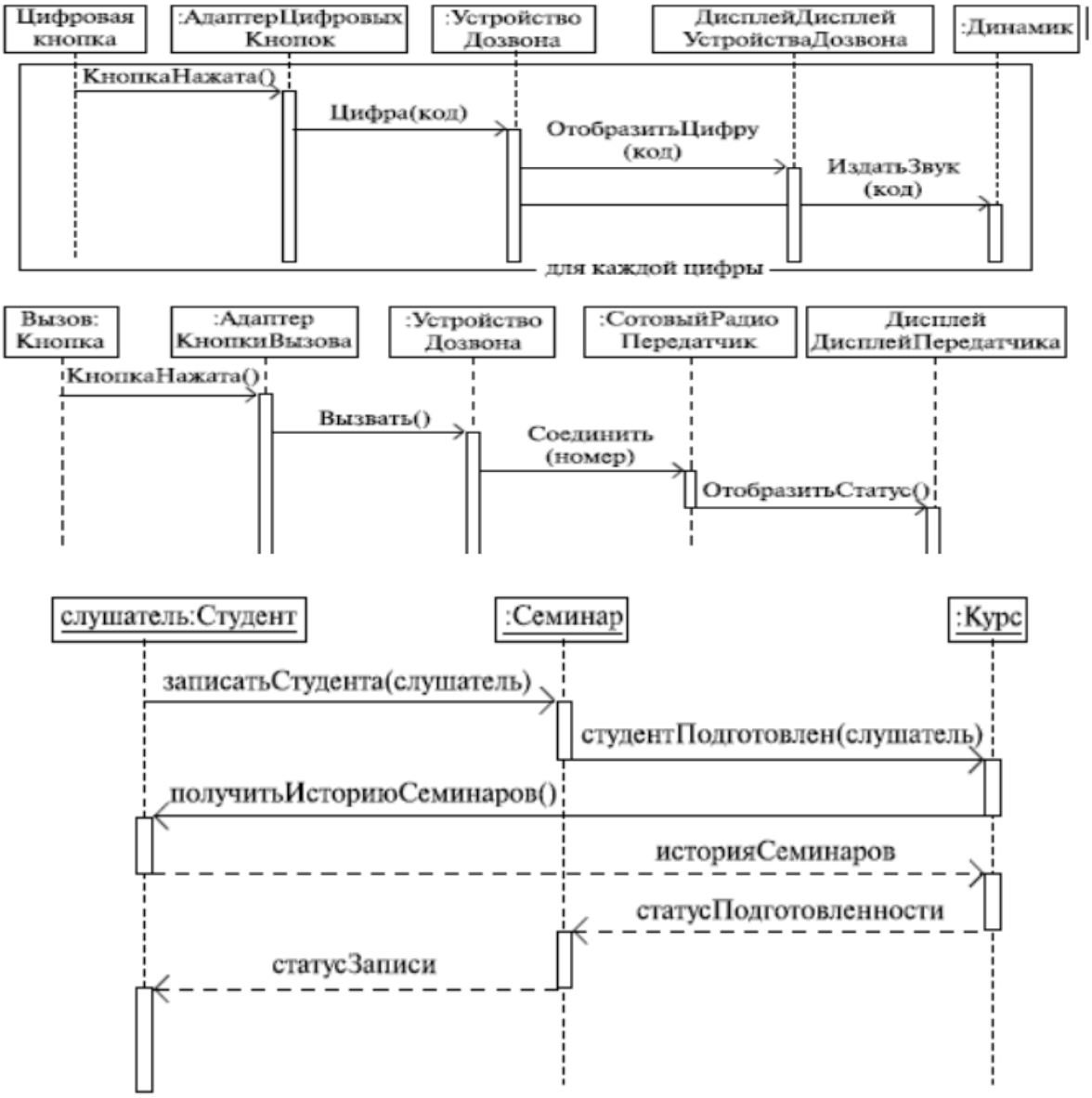


Рисунок 22.8 – Диаграмма последовательности

22.6 Диаграмма коммуникации

> Диаграмма коммуникации (communication diagram) – способ описания поведения, семантически эквивалентный диаграмме последовательности.

Фактически это такое же описание последовательности обмена сообщениями взаимодействующих экземпляров классификаторов, только выраженное другими графическими средствами. Более того, большинство инструментов умеет автоматически преобразовывать диаграммы последовательности в диаграммы коммуникации и обратно.

Таким образом, на диаграмме коммуникации, как и на диаграмме последовательности, применяют один основной тип сущностей – экземпляры взаимодействующих классификаторов и один тип отношений – связи. Однако здесь акцент делается не на времени, а на структуре связей между конкретными экземплярами.

22.7 Диаграмма компонентов

> Диаграмма компонентов (component diagram) показывает взаимосвязи между модулями (логическими или физическими), из которых состоит моделируемая система.

Основной тип сущностей на диаграмме компонентов – это сами компоненты, а также интерфейсы, посредством которых указывается взаимосвязь между компонентами. На диаграмме компонентов применяются следующие отношения:

- реализации между компонентами и интерфейсами (компонент реализует интерфейс);
- зависимости между компонентами и интерфейсами (компонент использует интерфейс).

22.8 Диаграмма размещения

Диаграмма размещения (deployment diagram), наряду с отображением состава и связей элементов системы, показывает, как они физически размещены на вычислительных ресурсах во время выполнения.

Таким образом, на диаграмме размещения, по сравнению с диаграммой компонентов, добавляется два типа сущностей: артефакт, который является реализацией компонента, и узел (может быть как классификатор, описывающий тип узла, так и конкретный экземпляр), а также отношение ассоциации между узлами, показывающее, что узлы физически связаны во время выполнения.

Вопросы для самоконтроля

1. В чем роль диаграммы использования?
2. Что позволяет описать диаграмма классов?
3. Какая диаграмма описывает состояния объекта и переходы между ними?

ГЛАВА 23
РЕФЛЕКСИЯ В C#

Достаточно интересным в объектно-ориентированном программировании является понятие рефлексия. Перед тем как увидеть и попробовать рефлексию в программировании, следует обратиться к определению этого понятия в философии. По мнению автора пособия, ООП – это есть философия программирования.

Одно из определений рефлексии звучит так: *рефлексия есть мысль, направленная на мысль.* В философии рефлексия – это инструмент размышления о самом себе, это то, что отличает человека от всех живых существ.

Проецируя это понятие на ООП, можно сказать, что рефлексия позволяет программе анализировать своё поведение, отслеживать и даже изменять свою собственную структуру. По сути, это ещё один шаг к искусственному интеллекту, когда кибернетические сущности смогут самообучаться и развиваться.

> Рефлексия – это система, предоставляющая выполняющему коду информацию о нём же самом. То есть позволяет читать выполняющему коду информацию о метаданных, которые он генерирует.

Рефлексия в C# активно использует класс Type. Рефлексия может понадобиться для различных исследований типа. Например, можно получить список полей в классе, при этом можно получить как открытые поля, так и закрытые. В следующем примере можно получить список всех полей класса и названия их типов.

```
using System;
using System.Reflection;

namespace TestReflection
{
    class Program
    {
        static void Main(string[] args)
        {
            var testClass = new TestReflectionClass();
            Type type = testClass.GetType();
            var fields =
type.GetFields(BindingFlags.Instance|BindingFlags.NonPublic|BindingFlags.Publ
ic);
            foreach (var fieldInfo in fields)
            {
                Console.WriteLine("{0}: {1}", fieldInfo.Name,
fieldInfo.FieldType);
            }
        }
    }

    public class TestReflectionClass
    {
        private string _firstName;
        private string _lastName;
        private int _old;
        public string Description;
    }
}
```

Результат работы примера программы с выводом на экран (рисунок 23.1)

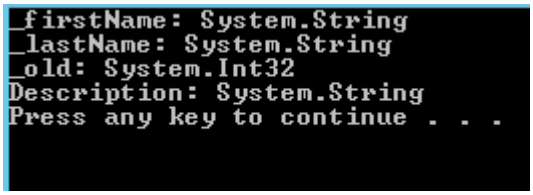


Рисунок 23.1 – Результат работы программы

Еще один пример, показывающий, как можно использовать методы класса через рефлексию (рисунок 23.2):

```
using System;
using System.Reflection;

namespace TestReflection
{
    class Program
    {
        static void Main(string[] args)
        {
            Type t = typeof(String);

            MethodInfo substr = t.GetMethod("Substring",
                new Type[] { typeof(int), typeof(int) });

            Object result =
                substr.Invoke("Hello, World!", new Object[] { 7, 5 });
            Console.WriteLine("{0} returned \"{1}\".", substr, result);
        }
    }

}
```

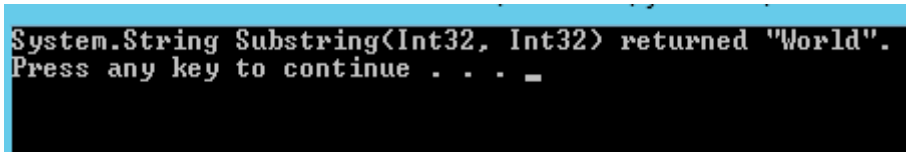


Рисунок 23.2 – Результат работы программы с рефлексией

Несмотря на то, что метод может быть объявлен как private, его можно вызвать из другого класса, используя рефлекцсию. Однако следует учитывать, что такой подход нарушает принципы ООП, хотя такая возможность присутствует.

```
using System;
using  System.Reflection;

namespace TestReflection
{
    class Program
    {
        static void Main(string[] args)
        {
            var testClass = new TestReflectionClass();
            Type type = testClass.GetType();
            MethodInfo info = type.GetMethod("PrivateMethod",
BindingFlags.Instance | BindingFlags.NonPublic);
```

```
            info.Invoke(testClass, null);
        }
    }

    public class TestReflectionClass
    {
        private void PrivateMethod()
        {
            Console.WriteLine("This is private method");
        }
    }
}
```

Вопросы для самоконтроля

1. Что такое рефлексия?
2. Что позволяет выполнять рефлексия?
3. Какой класс используется рефлексией?

ГЛАВА 24
ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

> Паттерн (от англ. Pattern) – образец, шаблон.
> Паттерны проектирования – это типичные, часто применяемые архитектурные решения.

Даже начинающие программисты уже использовали паттерны, сами того не подозревая, при этом у большинства из них есть даже названия.

Паттерны проектирования, как и абстракция, свойственны не только ООП разработке, но и другим парадигмам.

Паттерны служат для упрощения решения архитектурных проблем, которые либо уже обнаружились, либо, вероятнее всего, обнаружатся в ходе развития проекта. Ещё одна ценность от паттернов – формализация терминологии, т. к. чаще просто сообщить о том, что «в этом месте кода используется («цепочка обязанностей»)», чем долго рисовать поведение и отношения объектов на диаграммах.

Паттерны не привязаны к какому-либо конкретному языку программирования. Это просто подход к проектированию чего-либо. Если смотреть глубже, то многие паттерны ООП были созданы на основе реальных жизненный ситуаций в проектировании вполне себе реальных объектов.

24.1 Порождающие паттерны

> Порождающие паттерны – это такие, которые создают новые объекты или позволяют получить доступ к уже существующим.

Singleton (одиночка)

Один из самых известных и, пожалуй, самых спорных паттернов. Пусть в городе требуется организовать связь между жителями. С одной стороны, можно связать всех жителей между собой, проложив между ними кабели телефонных линий, но такая система неверна. Например, как затратно будет добавить ещё одного жителя в связи (протянуть по ещё одной линии к каждому жителю). Чтобы этого избежать, создаётся телефонная станция, которая и будет выступать в роли «одиночки». Она всегда одна, и если кому-то потребуется связаться с кем-то, то он может это сделать через данную телефонную станцию, потому что все обращаются только к ней. Соответственно, для добавления нового жителя нужно будет изменить только записи на самой телефонной станции. Один раз создав телефонную станцию, все могут пользоваться ей и только ей одной, в свою очередь эта станция помнит всё, что с ней происходило с момента её создания, и каждый может воспользоваться этой информацией, даже если он только приехал в город.

Основной смысл «одиночки» в том, чтобы, когда возникала ситуация (например, что нужна телефонная станция), ответом было, что она уже построена где-то, а не «Давай её сделаем заново». «Одиночка» всегда один.

Примечание – Несмотря на удобство применения данного паттерна, он является одним из самых спорных при разработке. Рекомендуется его применять, только если нет никакого другого способа решения, потому как это создаёт значительные сложности при тестировании кода.

Registry (реестр, журнал записей)

Данный паттерн предназначен для хранения записей, которые в него помещают, и возвращения этих записей (по имени), если они потребуются. В примере с телефонной станцией, она является реестром по отношению к телефонным номерам жителей.

Паттерны «одиночка» и «реестр» постоянно встречаются нам в повседневной жизни. Например, бухгалтерия в фирме является «одиночкой», так как она всегда одна и помнит, что с ней происходило с момента её начала работы. Фирма не создаёт каждый раз новую бухгалтерию, когда ей требуется выдать зарплату. В свою очередь бухгалтерия является и «реестром», так как в ней есть записи о каждом работнике фирмы.

Примечание – «Реестр» нередко является «одиночкой», однако это не всегда должно быть именно так. Например, можно заводить в бухгалтерии несколько журналов, в одном – работники от «А» до «М», в другом – от «Н» до «Я». Каждый такой журнал будет «реестром», но не «одиночкой», потому как журналов уже 2. Хотя иногда «реестр» служит именно для хранения «одиночек».

Сам паттерн «реестр» не является «порождающим паттерном» в полном смысле этого термина, однако его удобно рассматривать именно во взаимосвязи с ними.

Multiton (пул «одиночек»)

Как понятно из названия паттерна, это по своей сути «реестр», содержащий несколько «одиночек», каждый из которых имеет своё «имя», по которому к нему можно получить доступ.

Object pool (пул объектов)

По аналогии с «пулом одиночек» данный паттерн также позволяет хранить уже готовые объекты, однако они не обязаны быть «одиночками».

Factory (фабрика)

Суть паттерна практически полностью описывается его названием. Когда требуется получать какие-то объекты, например пакеты сока, совершенно не нужно знать, как их делают на фабрике. Достаточно сказать: «Сделайте мне пакет апельсинового сока». А «фабрика» возвращает вам требуемый пакет. Как? Всё это решает сама фабрика, например «копирует» уже существующий эталон. Основное предназначение «фабрики» в том, чтобы можно было при необходимости изменять процесс «появления» пакета сока, а самому потребителю ничего об этом не нужно было сообщать, чтобы он запрашивал его как и прежде.

Как правило, одна фабрика занимается «производством» только одного рода «продуктов». Не рекомендуется «фабрику соков» создавать с учётом производства автомобильных покрышек. Как и в жизни, паттерн «фабрика» часто создаётся «одиночкой».

Builder (строитель)

Данный паттерн очень тесно переплетается с паттерном «фабрики». Основное различие заключается в том, что «строитель» внутри себя, как правило, содержит все сложные операции по созданию объекта (пакета сока). Достаточно сказать, что хочу сока, а строитель запускает уже целую цепочку различных операций (создание пакета, печать на нем изображений, заправка в него сока, учёт того, сколько пакетов было создано, и т. п.).

Если потребуется другой сок, например ананасовый, достаточно также сказать только то, что нужно, а «строитель» уже позаботится обо всем остальном (какие-то процессы повторит, какие-то сделает заново и т. п.). В свою очередь процессы в «строителе» можно легко менять (например, изменить рисунок на упаковке), однако потребителю сока этого знать не требуется, он также будет легко получать требуемый ему пакет сока по тому же запросу.

Примечание – Чтобы лучше понять разницу между фабрикой и строителем, можно использовать следующую метафору.

«Фабрика» – это автомат по продаже напитков, в нем уже есть всё готовое (или «осталось разогреть»), достаточно только сказать, что нужно (нажимаете кнопку).
«Строитель» – это завод, который производит эти напитки и содержит в себе все сложные операции, может собирать сложные объекты из более простых (упаковка, этикетка, вода, ароматизаторы и т. п.) в зависимости от запроса.

Prototype (прототип)

Данный паттерн напоминает «фабрику», он также служит для создания объектов, однако с немного другим подходом. Предположим, что есть пустой пакет (из-под сока), а требуется полный с апельсиновым соком. Сообщим пакету, что хотим апельсиновый сок. Пакет в свою очередь создаёт свою копию и заполняет её запрошенным соком. В данном случае пустой пакет и является «прототипом», и в зависимости от того, что требуется, он создаёт на своей основе требуемые объекты (пакеты сока).

Клонирование не обязательно должно производиться на самом «пакете», это может быть и какой-то другой «объект», главное лишь то, что данный «прототип» позволяет получать его экземпляры.

Factory method (фабричный метод)

Ключевой сложностью объяснения данного паттерна является то, что это «метод», поэтому метафора метода будет использована как действие, например слово «Хочу!». Соответственно, паттерн описывает то, как должно выполняться это «Хочу!».

Допустим, фабрика производит пакеты с разными соками. Теоретически можно на каждый вид сока делать свою производственную линию, но это неэффективно. Удобнее сделать одну линию по производству пакетов-основ, а разделение ввести только на этапе заливки сока, который можно определять просто по названию сока. Однако откуда взять название? Для этого создаётся основной отдел по производству пакетов-основ, и предупреждаем все подотделы, что они должны производить нужный пакет с соком по простому «Хочу!» (т. е. каждый подотдел должен реализовать паттерн «фабричный метод»). Поэтому каждый подотдел заведует только своим типом сока и реагирует на слово «Хочу!».

Таким образом, если нам потребуется пакет апельсинового сока, то мы просто скажем отделу по производству апельсинового сока слово «Хочу!», а он в свою очередь скажет основному отделу по созданию пакетов сока: «Сделай свой обычный пакет, а вот сок, который туда нужно залить».

Примечание – «Фабричный метод» является как бы основой для «фабрики», «строителя» и «прототипа». В разработке часто именно так и получается, сперва реализуют фабричный метод, а по мере усложнения кода выбирают, во что именно его преобразовать, в какой из перечисленных паттернов. При использовании «фабричного метода» каждый объект как бы сам является «фабрикой».

Lazy initialization (отложенная инициализация)

Иногда требуется что-то иметь под рукой на всякий случай, но не всегда хочется прилагать каждый раз усилия, чтобы это каждый раз получать/создавать. Для таких случаев используется паттерн «отложенная инициализация».

Допустим, вы работаете в бухгалтерии и для каждого сотрудника должны подготавливать «отчёт о выплатах». Вы можете в начале каждого месяца делать этот отчёт на всех сотрудников, но некоторые отчёты могут не понадобиться, и тогда, скорее всего, вы примените «отложенную инициализацию», то есть вы будете подготавливать этот отчёт только тогда, когда он будет запрошен начальством (вышестоящим объектом), однако начальство в любой момент времени может сказать, что у него этот отчёт уже есть, а готов он

уже или нет, оно не знает и знать не должно. Как вы уже поняли, данный паттерн служит для оптимизации ресурсов.

Dependency injection (внедрение зависимости)

Внедрение зависимости позволяет переложить часть ответственности за какой-то функционал на другие объекты. Например, если нам требуется нанять новый персонал, то мы можем не создавать свой отдел кадров, а внедрить зависимость от компании по подбору персонала, которая по первому нашему требованию («нам нужен человек») будет либо сама работать как отдел кадров, либо же найдёт другую компанию (при помощи «локатора служб»), которая предоставит данные услуги.

«Внедрение зависимости» позволяет перекладывать и взаимозаменять отдельные части компании без потери общей функциональности.

Service Locator (локатор служб)

«Локатор служб» является методом реализации «внедрения зависимости». Он возвращает разные типы объектов (компаний) в зависимости от кода инициализации. Пускай стоит задача – доставить пакет сока, созданный строителем, фабрикой или ещё кем-либо, куда захотел покупатель. Мы спрашиваем у локатора: «Дай нам службу доставки». И он нас соединяет со службой доставки по номеру телефона, который директор ему дал, а мы уже просим службу доставить сок по нужному адресу. Сегодня одна служба, а завтра может быть другая. Нам без разницы, какая это конкретно служба, решение принимает директор и сообщает об этом локатору служб. Нам важно лишь знать, что они могут доставлять то, что мы им скажем, туда, куда скажем, то есть службы реализуют интерфейс «Доставить <предмет> на <адрес>».

24.2 Структурирующие паттерны

Данные паттерны помогают внести порядок и научить разные объекты более правильно взаимодействовать друг с другом.

Adapter или wrapper (адаптер, обертка)

Данный паттерн полностью соответствует своему названию. Чтобы заставить работать «советскую» вилку через евро-розетку требуется переходник. Именно это и делает «адаптер», служит промежуточным объектом между двумя другими, которые не могут работать напрямую друг с другом.

Bridge (мост)

Представим ситуацию, когда вам требуется работать на разных автомобилях, однако, садясь в новый автомобиль, вам уже желательно знать, как им управлять. Таким образом вы сталкиваетесь с паттерном «мост». С одной стороны, вы имеете множество различных автомобилей (разные модели и марки), но среди них есть общая абстракция (интерфейс) в виде руля, педалей, коробки передач и т. д. Мы задаём как бы правила изготовления автомобилей, по которым мы можем создавать любые их виды, но за счёт сохранения общих правил взаимодействия с ними, мы можем одинаково управлять каждым из них. «Мостом» в данном случае является пара двух «объектов»: конкретного автомобиля и правил взаимодействия с этим (и любым другим) автомобилем.

Composite (компоновщик)

Довольно интересный паттерн, суть которого заключается в минимизации различий в управлении как группами объектов, так и индивидуальными объектами. Для примера

можно рассмотреть управление солдатами в строю. Существует строевой устав, который определяет, как управлять строем, и согласно этому уставу абсолютно не важно, кому отдаётся приказ (например, «шагом марш»): одному солдату или целому взводу. Соответственно, в устав (если его в чистом виде считать паттерном «компоновщик») нельзя включить команду, которую может исполнить только один солдат, но не может исполнить группа или наоборот.

Decorator (декоратор, оформитель)

Как понятно из названия, данный паттерн чаще всего используется для расширения исходного объекта до требуемого вида. Например, условно можно считать «декоратором» человека с кистью и красной краской. Таким образом, какой бы объект (или определенный тип объектов) мы не передали в руки «декоратору», на выходе мы будем получать красные объекты.

Facade (фасад)

Паттерн «фасад» используется для того, чтобы делать сложные вещи простыми. Возьмём для примера автомобиль. Представьте, если бы управление автомобилем происходило немного по-другому: нажать одну кнопку, чтобы подать питание с аккумулятора, нажать другую, чтобы подать питание на инжектор, третью, чтобы включить генератор, четвертую для того, чтобы зажечь лампочку на панели, и т. д. Всё это было бы очень сложно. Для этого такие сложные наборы действий заменяются более простыми и комплексными (например, «повернуть ключ зажигания»). В данном случае поворот ключа зажигания и будет тем самым «фасадом» для всего обилия внутренних действий автомобиля.

Front controller (единая точка входа)

Если проводить аналогии с реальными миром, то «единая точка входа» – это то, через что вы сейчас читаете данную статью (например, браузер). Она служит «единой точкой входа» для всего интернет-пространства. То есть вы используете один интерфейс (браузер) для получения доступа к разным объектам большой системы (сайтам в интернете). Данный паттерн в целом сильно похож на «фасад».

Flyweight (приспособленец)

Самым лучшим примером (который я смог найти в реальной жизни) для метафорического сравнения паттерна «приспособленец» является театральная постановка. Представьте, что нам требуется поставить пьесу. Однако по сценарию в этой пьесе задействованы несколько десятков людей, которые выполняют одинаковые действия, например участвуют в массовках различных сцен в разные промежутки времени, но между ними всё же есть какие-то различия, например костюмы. Нам бы стоило огромных денег нанимать для каждой роли отдельного актёра, поэтому мы используем паттерн «приспособленец». Мы создадим все нужные нам костюмы, но для каждой массовки будем переодевать небольшую группу актёров в требуемые для этой сцены костюмы. В результате мы имеем возможность ценой малых ресурсов создавать видимость управления большим количеством, казалось бы, разных объектов.

Proxy или surrogate (прокси, заместитель, суррогат)

Данный паттерн позволяет создавать какие-либо специальные механизмы доступа к объекту, что чаще всего направлено именно на улучшение производительности отдельных частей программы. В реальной жизни можно привести следующий пример: сотрудникам одного из подразделений фирмы регулярно требуется получать информацию о том, какого числа бухгалтерия планирует выплатить зарплату. С одной стороны, каждый из

них может индивидуально и регулярно ездить в бухгалтерию для выяснения этого вопроса (полагаю, такая ситуация нередко встречается во многих организациях). С другой стороны, при приближении планируемой даты подразделение может выбрать одного человека, который будет выяснять эту информацию у бухгалтерии, а в последствии уже все в подразделении могут выяснить эту информацию у него (что значительно быстрее). Вот именно этот человек и будет реализованным «прокси» паттерном, который будет предоставлять специальный механизм доступа к информации из бухгалтерии.

24.3 Паттерны поведения

Эта группа паттернов позволяет структурировать подходы к обработке поведения и взаимодействия объектов. Проще говоря, как должны проходить процессы, в которых существует несколько вариантов протекания событий.

Chain of responsibility (цепочка обязанностей)

Самым простым примером цепочки обязанностей можно считать получение какого-либо официального документа. Например, требуется получить справку со счета из банка. Так или иначе вы должны эту справку получить, однако кто именно её должен вам дать – пока не ясно. Вы приходите в местное отделение банка, вам говорят, что мы сейчас заняты, идите в другое отделение, дальше вы идёте в другое, там вам отвечают, что мы этим не занимаемся, вы идёте в региональное отделение и там получаете нужную справку. Таким образом, паттерн реализует «цепочку обязанностей», отдельные объекты которой (отделения банка) должны обработать ваш запрос. Соответственно, ваш запрос может быть обработан в первом же отделении, или же в нескольких, в зависимости от самого запроса и обрабатывающих объектов.

Command или action (команда, действие)

Паттерн «команда» очень похож в реальной жизни на кнопки выключателей света в квартирах и домах. Каждый выключатель делает одно простое действие – разъединяет или соединяет два провода, но что стоит за этими проводами, выключателю неизвестно. Что подключат, то и произойдёт. Точно также действует и паттерн «команда». Он лишь определяет общие правила для объектов (устройств) в виде соединения двух проводов для выполнения команды, а что именно будет выполнено, уже определяет само устройство (объект).

Interpreter (интерпретатор)

Сравнить данный паттерн можно с тем, как вы закладываете часто используемые действия в сокращённый набор слов, чтобы сам «интерпретатор» потом превратил этот набор в более комплексные осмысленные действия. По сути, каждый человек постоянно является «интерпретатором». Хотите провести жизненный эксперимент? Если из дома выходит кто-то из вашей семьи (муж, жена, ребёнок), скажите ему простой набор слов «Литр молока, половинку белого, 200 грамм творога». Вы ничего особенного не сказали, лишь перечислили набор продуктов, однако велик шанс того, что «интерпретатор» транслирует это в команду: зайди по дороге в продуктовый магазин, купи следующее … и принеси это домой. Паттерн «интерпретатор» призван сократить часто исполняемые действия в более короткое их описание.

Iterator (итератор, указатель)

Все помнят школьное «на первый второй рассчитайся!»? Вот именно в этот момент шеренга являлась реализацией паттерна «итератор», хотя в программировании это конеч-

но более функциональное понятие, но суть примерно та же. «Итератор» предоставляет правила доступа к списку каких-либо объектов независимо от того, что это за объекты. То есть не важно, какой именно класс построен и из каких учеников, должны быть общие правила подсчёта и обращения к каждому ученику по списку (например, 13-ый, выйти из строя). Нередко паттерн «итератор» используется для доступа к «реестру». Ссылки, которые вы видите на многих сайтах для переходов по страницам, вроде «следующая», «предыдущая», «в начало» и т. п., по своей сути также являются доступом «итератору», который отвечает за страницы сайта.

Mediator (посредник)

Вспомним пример из паттерна «одиночка». Телефонная станция в том примере также являлась паттерном «посредник», то есть обеспечивала взаимодействие группы объектов без необходимости обеспечения связи каждого объекта друг с другом. Однако дополнительной ответственностью этого «паттерна» является также управление этой группой через «посредника». Если мы возьмём пример с армейским строем, медиатором будет командир отделения, то есть нам нет необходимости взаимодействовать с каждым солдатом в отдельности, достаточно отдавать приказания лишь командиру отделения, а он уже сам решит, какие действия должны быть выполнены внутри его отделения.

Memento (хранитель)

Никогда не просили друга с сотовым телефоном на время запомнить (записать себе) тот номер, что диктуют вам по телефону, потому что вы не можете его запомнить сами (телефон занят)? В этот момент ваш друг реализовывал паттерн «хранитель». Он служит для тех случаев, когда какому-либо объекту требуется сохранить своё состояние (состояние знания номера) в другом объекте (вашем друге) и при необходимости его потом восстановить (спросить у друга номера и тем самым восстановить состояние, когда вы его знали). Также уместен аналог с тем, как в играх работает сохранение. Файл «сейва» как раз и будет тем самым паттерном «хранитель».

Observer или Listener (наблюдатель, слушатель)

Очень распространённый паттерн в реальной жизни. Например, если вы подписались на какую-либо email- (или смс-) рассылку, то ваш email (или номер сотового телефона) начинает реализовывать паттерн «наблюдатель». Как только вы подписываетесь на событие (например, новая статья или сообщение), всем, кто подписан на это событие, (наблюдателям) будет выслано уведомление, а они уже в свою очередь могут выбрать, как на это сообщение реагировать.

Blackboard (доска объявлений)

Данный паттерн служит для обеспечения взаимодействия между большим количеством объектов. Он является расширением паттерна «наблюдатель» и позволяет централизованно обслуживать как «наблюдателей», так и «создателей событий». По аналогии подпиской на email-уведомления будет сам сайт подписки, который обслуживает множество подписчиков и тех, кто для них создаёт информацию (сообщения).

Servant (слуга)

Как следует из названия, данный паттерн служит для предоставления группе объектов какого-либо общего функционала. Например, телефонная станция является для жителей города паттерном «слуга», если речь заходит о том, как узнать точное время (набрать номер 100).

State (состояние)

В реальной жизни каждый человек может прибывать в разных состояниях. Точно также порой требуется, чтобы объекты в программе вели себя по-разному, в зависимости от каких-либо их внутренних состояний.

Strategy (стратегия)

Используется для выбора различных путей получения результата. Внутри «стратегии» хранятся различные способы поведения, и чтобы выбрать, ему нужны определенные параметры (в данном случае это объем денежных средств). Требуется знать, как устроена сама «стратегия» и какие алгоритмы внутри неё.

Specification (спецификация, определение)

Паттерн «спецификация» позволяет описывать, подходит ли данный объект нам на основе каких-либо критериев. Например, мы имеем несколько контейнеров для погрузки на судно. Чтобы определить, грузить контейнер или нет на определенное судно, нам нужно выбрать метод, как это определять. Реализация такого метода и является паттерном «спецификация». В самом простом случае для каждого контейнера мы можем определить в паттерне «спецификация», совпадает ли страна назначения корабля со страной назначения контейнера. Соответственно, мы один раз вводим правило: сравнить две страны назначения. И применяем это правило ко всем контейнерам для проверки.

Subsumption (категоризация)

Данный паттерн является прямым последователем паттерна «спецификация». Он позволяет распределять объекты по категориям на основе каких-либо условий. Соответственно, (по аналогии с примером кораблей и контейнеров) это категоризация по тому, какие контейнеры в какие страны направляются.

Visitor (посетитель)

Данный паттерн можно сравнить с прохождением обследования в больнице. Однако «посетителем» в терминах паттернов здесь будут сами врачи. Чтобы было понятнее: у нас есть больной, которого требуется обследовать и полечить, но так как за разные обследования отвечают разные врачи, то мы просто присылаем к больному врачей в качестве «посетителей». Правило взаимодействия для больного очень простое: приглашают врача («посетителя»), чтобы он сделал свою работу, а врач («посетитель») приходит, обследует и делает всё необходимое. Таким образом, следуя простым правилам, можно использовать врачей для разных больных по одним и тем же алгоритмам. Как уже было сказано, паттерном «посетитель» в данном случае является врач, который может одинаково обслуживать разные объекты (больных), если его позовут.

Single-serving visitor (одноразовый посетитель)

Является частным случаем использования паттерна «посетитель». Если в случае с обычным «посетителем» у нас есть врач, которого мы можем отправить к разным больным (и при желании по несколько раз), то в данном паттерне можно привести аналогию, что мы нанимаем врача, отправляем его к одному больному и после обследования сразу увольняем.

Hierarchical visitor (иерархический посетитель)

Тот же самый паттерн «посетитель», однако в данном случае он отправляется не к одному больному, а в целую больницу и обходит там всех больных.

ТЕРМИНЫ

ActiveX	Является основой для определения независимым образом программных компонентов в языке программирования. Программные приложения могут состоять из одного или нескольких компонентов в целях обеспечения их функциональности
ADO.NET	ADO.NET нацелена на автономную работу с помощью объектов DataSet. Эти типы представляют локальные копии любого количества взаимосвязанных таблиц данных, каждая из которых содержит набор строк и столбцов. Объекты DataSet позволяют вызывающей сборке (наподобие веб-страницы или программы, выполняющейся на настольном компьютере) работать с содержимым DataSet, изменять его, не требуя подключения к источнику данных, и отправлять обратно блоки измененных данных для обработки с помощью соответствующего адаптера данных
API - интерфейс	Представляет набор готовых классов, интерфейсов и делегатов, которые можно использовать при разработке сторонних программных продуктов
COM	Объектная модель компонентов, компьютерная технология, разработанная компанией Microsoft
COM - сервер	Это специальным образом оформленное и зарегистрированное в системе приложение
COM+	Данная технология входит в состав серверных операционных систем Microsoft и предназначена для поддержки систем обработки транзакций
DLL-библиотека	Представляют собой библиотеки динамической компоновки сборок
DOM	Объектная модель документа, чаще всего использующаяся при Web-программировании
EDI	Взаимодействие на предприятиях между компьютерами в виде стандартизированных бизнес-операций стандартного формата
GAC	Каталог, размещающий в себе большинство сборок .NET Framework
GUID-идентификаторы	Статистически уникальный 128-битный идентификатор. Его главная особенность – уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов
IDE-среда	Среда, представляющая развитый графический интерфейс и имеющая множество готовых компонентов для упрощения разработки программного обеспечения
IntelliSense	Технология, использующаяся в Visual Studio и представляющая собой особенность дополнения частей кода при их наборе с клавиатуры программистом
JIT-компиляция	Процесс компиляции CIL-инструкций в соответствующий машинный код, при котором JIT-компилятор будет помещать результаты в кэш в соответствии с тем, как того требует целевая операционная система
MFC	Библиотека базовых классов Microsoft, предназначенная для разработчиков на C++
MonoTouch	Платформа, позволяющая запускать приложения, разработанные с использованием .NET в различных мобильных устройствах, смартфонах и т. д.
MVC	Схема использования нескольких шаблонов проектирования, с помощью которых модель данных приложения, пользовательский интерфейс и взаимодействие с пользователем разделены на три отдельных компонента так, что модификация одного из компонентов оказывает минимальное воздействие на остальные
Office Automation API	Представляет собой API-интерфейс, с помощью которого можно разрабатывать специальные плагины, гаджеты и т. п. для пакета Microsoft Office
Pascal	Язык программирования общего назначения. Один из наиболее известных языков программирования, широко применялся в промышленном программировании, обучении программированию в высшей школе, является базой для ряда других языков
Solution Explorer	Окно IDE-среды Visual Studio, в котором отображаются сведения о разрабатываемом проекте
Unicode	Представляет собой кодировку символов, в которой представлены символы практически всех известных языков
UNIX	Семейство переносимых, многозадачных и многопользовательских операционных систем
Visual InterDev	IDE-среда, применяемая для создания Web-приложений на основе Microsoft Active Server Pages

Win32	Общее наименование целого набора базовых функций интерфейсов программирования приложений операционных систем семейств Windows и Windows NT корпорации «Майкрософт»
Windows Forms	Инструментальный набор Windows Forms предоставляет типы, необходимые для построения графических пользовательских интерфейсов для настольных компьютеров, создания специализированных элементов управления, управления ресурсами (например, строками и значками) и выполнения других задач, возникающих при программировании для пользовательских компьютеров. Имеется и дополнительный API по имени GDI+ (представленный сборкой System.Drawing.dll), который предоставляет дополнительные типы, позволяющие программисту генерировать двухмерную графику, взаимодействовать с сетевыми принтерами и обрабатывать графические данные
XML	Расширяемый язык разметки, представляющий собой свод общих синтаксических правил
Деструктор	Специальный метод, вызываемый средой исполнения программы в момент, когда объект удаляется из оперативной памяти. Деструктор используется в тех случаях, когда в состав класса входят ресурсы, требующие явного освобождения (файлы, соединения с базами данных, сетевые соединения и т. п.)
Интерфейс	Набор методов и свойств объекта, находящихся в открытом доступе и призванных решать определенный круг задач, к примеру, интерфейс формирования графического представления объекта на экране или интерфейс сохранения состояния объекта в файле или базе данных
Конструктор	Специальный метод, выполняемый сразу же после создания экземпляра класса. Конструктор инициализирует поля объекта – приводит объект в начальное состояние, выделяет место в памяти для объекта. Конструкторы могут быть как с параметрами, так и без. Конструктор без параметров называют конструктором по умолчанию, который может быть только один. Имя метода конструктора чаще всего совпадает с именем самого класса
Метод	Процедура или функция, выполняющаяся в контексте объекта, для которого она вызывается. Методы могут изменять состояние текущего объекта или состояния объектов, передаваемых им в качества параметров
Модификатор доступа	Дополнительная характеристика членов класса, определяющая, имеется ли к ним доступ из внешней программы, или же они используются исключительно в границах класса и скрыты от окружающего мира. Модификаторы доступа разделяют все элементы класса на детали реализации и открытый или частично открытый интерфейс
Поле	Элемент данных класса: переменная элементарного типа, структура или другой класс, являющийся частью класса
Свойство	Специальный вид методов, предназначенный для модификации отдельных полей объекта. Имена свойств обычно совпадают с именами соответствующих полей
Состояние объекта	Набор текущих значений полей объекта
Статический член	Любой элемент класса, который может быть использован без создания соответствующего объекта
Член класса	Поля, методы и свойства класса

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Бадд, Т. Объектно-ориентированное программирование в действии / Т. Бадд ; пер. с англ. А. Бердников. – СПб. : Питер, 1997. – 464 с. : ил.

2. Бобровский, С. И. Разработка приложений для бизнеса : учеб. курс / С. И. Бобровский. – СПб. : Питер, 2007. – 560 с.

3. Буч, Г. Объектно-ориентированное проектирование с примерами применения / Г. Буч ; пер. с англ. А. А. Иванова [и др.] ; под ред. А. Н. Артамошкина. – К. : Диалектика ; М. : АО И.В.К., 1992. – 519 с. : ил.

4. Приемы объектно-ориентированного проектирования / Э. Гамма [и др.]. – СПб. : Питер, 2012. – 368 с.

5. Дейтел, П. Как программировать на Visual C# / П. Дейтел, Х. Дейтел. – 5-е изд. – СПб. : Питер, 2014. – 858 с.

6. Зиборов, В. Visual C# 2012 на примерах / В. Зиборов. – СПб. : БХВ-Петербург, 2015. – 966 с.

7. Ишкова, Э. А. Самоучитель C#. Начала программирования / Э. А. Ишкова. – 2-изд. – СПб. : Наука и техника, 2013. – 496 с.

8. Колесов, Ю. Б. Моделирование систем. Объектно-ориентированный подход : учебное пособие для вузов / Ю. Б. Колесов, Ю. Б. Сениченков. – СПб. : БХВ-Петербург, 2006. – 192 с. : ил.

9. Маклафин, Б. Объектно-ориентированный анализ и проектирование / Б. Маклафин. – СПб. : Питер, 2013. – 568 с.

10. Официальная страница Microsoft : руководство по программированию на C# [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide. – Дата доступа: 02.05.2017.

11. Павловская, Т. А. C#. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. – СПб. : Питер, 2014. – 432 с.

12. Пол, А. Объектно-ориентированное программирование на C++ / А. Пол ; пер. с англ. Д. Ковальчука. – 2-е изд. – М. : Невский Диалект, 1999 ; БИНОМ ; СПб. – 462 с. : ил. – Перевод изд.: Object-oriented programming using C++ / I.Pohl.

13. Пугачев, С. В. Разработка приложений для Windows 8 на языке C# / С. В. Пугачев. – СПб. : БХВ-Петербург, 2013. – 416 с.

14. Рой, О. Искусство автономного тестирования с примерами на C# / О. Рой. – М. : ДМК Пресс, 2016. – 428 с.

15. Синтес, А. Освой самостоятельно объектно-ориентированное программирование за 21 день / А. Синтес ; пер.с англ. А. И. Захарова [и др.] ; под ред. Я. К. Шмидского. – М. : Вильямс, 2002. – 672 с. : ил.

16. Троелсен, Э. Язык программирования C# 2010 и платформа .NET 4.0 / Э. Троелсен ; пер. с англ. Э. Троелсен. – М. : ООО «И. Д. Вильямс», 2013. – 1311 с.

17. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 5-е изд. – М. : Вильямс, 2015. – 633 с.

18. Фаронов, Ф. Ф. Программирование на языке C# / Ф. Ф. Фаронов. – СПб. : Питер, 2007. – 241 с.

19. Фленов, М. Е. Библия C# / М. Е. Фленов. – СПб. : БХВ-Петербург, 2011. – 560 с.

20. Фримен, Адам ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов / Адам Фримен. – М. : Вильямс, 2015. – 736 с.

21. Фролов, А. В. Язык C#: Самоучитель / А. В. Фролов, Г. В. Фролов. – М. : ДИАЛОГ-МИФИ, 2003. – 560 с.

22. Хорев, П. Б. Объектно-ориентированное программирование : учеб. пособие для студентов высшего профессионального образования / П. Б. Хорев. – М. : Издательский центр «Академия», 2012. – 411 с.

23. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. – М. : Академия, 2018. – 384 с.

24. Шилдт, Г. C# 4.0: Полное руководство / Г. Шилдт. – М. : ООО «И. Д. Вильямс», 2015. – 291 с.

25. Шилдт, Г. C# 3.0: Полное руководство / Г. Шилдт. – М. : ООО «И. Д. Вильямс», 2011. – 1056 с.

Учебное издание

Володько Людвик Павлович
Николаенко Владимир Леонидович
Николаенко Денис Владимирович

Объектно-ориентированное программирование

Учебное пособие

Ответственный за выпуск П. Б. Пигаль

Редактор Т. И. Андросюк
Корректор Ю. В. Цвикевич

Подписано в печать 07.07.2020 г. Формат 60×84/8.
Бумага офсетная. Гарнитура «Таймс». Ризография.
Усл. печ. л. 19,06. Уч.-изд. л. 7,02.
Тираж 78 экз. Заказ № 143.

Отпечатано в редакционно-издательском отделе
Полесского государственного университета.
225710, г. Пинск, ул. Днепровской флотилии, 23.