Методы программирования

Кафедра «Вычислительные системы и информационная безопасность»

Лекционный курс

Автор
Айдинян А.Р.

Аннотация

Лекционный курс предназначен для студентов второго курса специальности 10.05.02 «Информационная безопасность телекоммуникационных систем» очной формы обучения. Раскрывает лекционный материал тем рабочей программы дисциплины и соответствует ФГОС3+.

Автор

Айдинян Андрей Размикович –
доцент, к.т.н.,

ОГЛАВЛЕНИЕ

ГЛАВА 1 МЕТОДЫ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

ЛЕКЦИЯ №1

1.1 Обзор парадигм программирования

Императивное программирование

Императивное программирование — это исторически первая методология программирования. Она ориентирована на классическую фон Неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой. Методология императивного программирования характеризуется принципом *последовательного изменения состояния вычислителя пошаговым образом*. При этом управление изменениями полностью определено концепцией *алгоритма* и полностью контролируемо. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех ячеек памяти, состояние процессора (в том числе — указатель текущей команды) и всех сопряженных устройств. Единственная структура данных — последовательность ячеек (пар «адрес» — «значение») с линейно упорядоченными адресами. В качестве *математической модели* императивное программирование использует машину Тьюринга-Поста — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов. Языки, поддерживающие данную вычислительную модель, являются как бы средством описания функции переходов между состояниями вычислителя. Основным их синтаксическим понятием является *оператор*. Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным.

Структурное программирование

Структурное программирование — это такой подход к написанию программ, при котором они становятся более ясными для понимания, более корректными и лучше подходящими для дальнейших модификаций. Структурное программирование возникло как вариант решения проблемы уменьшения сложности разработки программного обеспечения. Структурное программирование дает определенные рекомендации по разбиению программы на отдельные блоки и основывается на принципе разработки программы сверху вниз с постепенной детализацией.

В основу структурного программирования положены следующие достаточно простые положения:

– алгоритм и программа должны составляться поэтапно (по шагам);

– сложная задача должна разбиваться на достаточно простые части, каждая из которых имеет один вход и один выход;

– логика алгоритма и программы должна опираться на минимальное число достаточно простых базовых управляющих структур.

Структурное программирование иногда называют еще «программированием без GO TO». Рекомендуется избегать употребления оператора перехода всюду,

где это возможно, но, чтобы это не приводило к слишком громоздким структурированным программам.

К полезным случаям использования оператора перехода можно отнести выход из цикла или процедуры по особому условию, «досрочно» прекращающего работу данного цикла или данной процедуры, т.е. завершающего работу некоторой структурной единицы (обобщенного оператора) и тем самым лишь локально нарушающего структурированность программы.

Фундаментом структурного программирования является *теорема о структурировании:* любую сложную задачу всегда можно представить с использованием ограниченного числа элементарных управляющих структур: следования, ветвления и повторения (цикла), любой алгоритм может быть реализован в виде композиции этих трех конструкций (рис. 1.1).

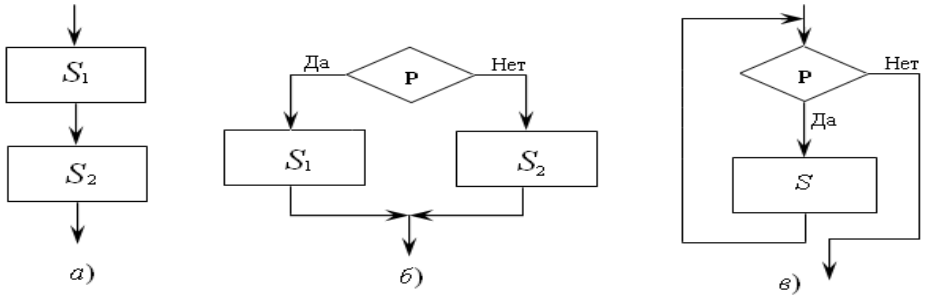


Рис. 1.1

(*а*) последовательность (или просто последовательность), (*б*) — структура выбора (ветвление), (*в*) — структура цикла с предусловием.

При словесной записи алгоритма указанные структуры имеют соответственно следующий смысл:

если P, то выполнить S_1, иначе выполнить S_2»,

«до тех пор, пока P, выполнять S»,

где P — условие; S, S_1, S_2 — действия.

Модульное программирование

Модульное программирование — это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой. Модуль — это совокупность команд (функций и процедур), к которым можно обратиться по имени, независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы.

Используется восходящее проектирование (снизу вверх) — подход к проектированию, при котором в основной задаче определяются минимальные подзадачи, решение которых возможно без использования вызовов дополнительных вновь создаваемых модулей. После того, как все подзадачи будут представлены в виде программных модулей, формируются вызывающие модули, решающие более крупные задачи. Процедура повторяется до тех пор, пока не будет получен главный модуль, решающий основную задачу. Это способ программирования, при котором вся программа разбивается на группу

компонентов, называемых модулями. Каждый со своими контролируемыми размерами, четким значением и хорошо определенным интерфейсом с внешней средой.

Единственная альтернатива модульности — монолитная программа, что, конечно, неудобно.

В основе модульного программирования лежат три основных концепции:

1) Принцип утаивания информации Парнаса — никакие данные, используемые внутри модуля не могут быть доступны извне, кроме тех, которые передаются через интерфейс ввода/ вывода, определяемых списком параметров

2) Аксиома модульности Коуэна. Модуль должен удовлетворять следующим условиям:

— блочность организации, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;

— синтаксическая обособленность, т. е. выделение модуля в тексте синтаксическими элементами;

— семантическая независимость, т. е. независимость от места, где программная единица вызвана;

— общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;

— полнота определения, т. е. самостоятельность программной единицы.

3) Сборочное программирование Цейтина. Модули — это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:

— стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;

— потребность организационного расчленения крупных разработок;

— возможность параллельного исполнения модулей (в контексте параллельного программирования).

Функциональная спецификация модуля должна включать:

— синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;

— описание семантики функций, выполняемых модулем по каждому из его входов.

Набор характеристик модуля состоит из следующих конструктивных характеристик:

1) размера модуля. В модуле должно быть 7 (+/-2) конструкций (например, операторов для функций или функций для пакета). Это число берется на основе представлений психологов о среднем оперативном буфере памяти человека. Символьные образы в человеческом мозгу объединяются в "чанки" — наборы фактов и связей между ними, запоминаемые и извлекаемые как единое целое. В каждый момент времени человек может обрабатывать не более 7 чанков.

Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора.

2) прочности (связности) модуля. *Связность (прочность)* модуля (cohesion) — мера независимости его частей. Чем выше связность модуля — тем лучше, тем больше связей по отношению к оставшейся части программы он упрятывает в себе. В связном модуле обращение идет к локальным, а не глобальным данным. Глобальные данные вредны и опасны. Локальность данных дает возможность легко читать и понимать модули, а также легко удалять их из программы.

Объектно-ориентированное программирование

Метод структурного программирования оказался эффективен при написании программ «ограниченной сложности». Однако с возрастанием сложности реализуемых программных проектов и, соответственно, объема кода создаваемых программ, возможности метода структурного программирования оказались недостаточными.

Основной причиной возникших проблем можно считать то, что в программе не отражалась непосредственно структура явлений и понятий реального мира и связей межу ними. При попытке анализа и модификации текста программы программист вынужден был оперировать искусственными категориями.

Чтобы писать все более сложные программы, необходим был новый подход к программированию. В итоге были разработаны принципы объектно-ориентированного программирования (ООП), который аккумулирует лучшие идеи, воплощённые в структурном программировании, и сочетает их с мощными новыми концепциями.

ООП — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Необходимо обратить внимание на следующие важные части этого определения: 1) объектно-ориентированное программирование использует в качестве основных логических конструктивных элементов объекты, а не алгоритмы; 2) каждый объект является экземпляром определенного класса; 3) классы образуют иерархии. Программа считается объектно-ориентированной, только если выполнены все три указанных требования. В частности, программирование, не использующее наследование, называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных.

Метод *объектно-ориентированной декомпозиции* — заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма.

Метод *абстрактных типов данных* — метод, лежащий в основе инкапсуляции. Поддерживается концепцией абстрактных типов данных.

Метод *пересылки сообщений* – заключается в описании поведения системы в терминах обмена сообщениями между объектами. Поддерживается концепцией сообщения.

Методы программирования

Вычислительная модель чистого ООП поддерживает только одну операцию — *посылку сообщения объекту*. Сообщения могут иметь параметры, являющиеся объектами.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля — персональные переменные данного объекта, значениями которых являются ссылки на другие объекты. Об одном из полей объекта хранится ссылка на объект-предок, которому переадресуются все сообщения, не обработанные данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обычно выделяются в отдельный объект, называемый классом данного объекта. Сам объект называется экземпляром указанного класса.

В объектно-ориентированном программировании определяют три основных принципа:

Инкапсуляция — это сокрытие информации и комбинирование данных и функций (методов) внутри объекта.

Наследование — построение иерархии порожденных объектов с возможностью для каждого такого объекта-наследника доступа к коду и данным всех порождающих объектов-предков. Построение иерархий является достаточно сложным делом, так как при этом приходится выполнять классифицирование.

Большинство окружающих нас объектов относится к категориям:

— реальные объекты — абстракции предметов, существующих в физическом мире;

— роли — абстракции цели или назначения человека, части оборудования или организации;

— инциденты — абстракции чего-то произошедшего или случившегося;

— взаимодействия — объекты, получающиеся из отношения между другими объектами.

Полиморфизм (полиморфизм включения) — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного сообщения. Сообщение может быть передано вверх по иерархии наследования, если сам класс не располагает методом для его обработки. Если обработчик для сообщения выбирается динамически, то методы принято называть виртуальными.

Естественным средством структурирования в данной методологии являются классы. Классы определяют, какие поля и методы экземпляра доступны извне, как обрабатывать отдельные сообщения и т. п. В чистых объектно-ориентированных языках извне доступны только методы, а доступ к данным объекта возможен только через его методы.

Взаимодействие задач в данной методологии осуществляется при помощи обмена сообщениями между объектами, реализующими данные задачи.

Класс, например, это проект дома. Он на бумаге определяет, как будет выглядеть дом, чётко описывает все взаимосвязи между его различными частями, даже если дом не существует в реальности. А объект — это реальный дом,

который построен в соответствии с проектом. Данные, которые хранятся в объекте похожи на дерево, провода и бетон, из которых построен дом: без сборки в соответствии с проектом, они будут всего лишь кучей материалов . Однако, собранные вместе они становятся отличным и удобным домом.

Классы формируют структуру данных и действий и используют эту информацию для строительства объектов. Из одного класса могут быть построен один и более объектов, каждый из которых будет независим от других. Продолжая аналогию со строительством, целый район может быть построен по одному проекту: 150 различных домов, которые имеют одинаковую структуру, но различные параметры.

Функциональное программирование

Функциональное программирование — программирование, основанное на идеях лямбда-исчисления и теории рекурсивных функций. Программы представляют собой неупорядоченный набор уравнений, определяющих функции и их значения рекурсивно через функцию и значения, которые задаются функцией от других значений.

1.2 Модели жизненного цикла программного обеспечения

Жизненный цикл (ЖЦ) ПО — это период «жизни» программной системы, начиная с момента возникновения потребности в ней и заканчивая моментом полного ее выхода из эксплуатации. ЖЦ ПО представляют в виде последовательности стадий (этапов) и выполняемых на них процессов.

Модель жизненного цикла — структура, содержащая процессы, действия и задачи, которые реализуются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

Исторически в ходе развития теории проектирования программного обеспечения и по мере его усложнения утвердились четыре основные модели ЖЦ.

1. Каскадная модель (также называют водопадной либо последовательной) ЖЦ была предложена Уинстон Ройсом в 1970 г. и является старейшей парадигмой процесса разработки программного обеспечения. Модель названа каскадной, чтобы подчеркнуть, что разработка информационной системы рассматривается как последовательное выполнение этапов, причем переход на следующий иерархически нижний этап происходит только после полного завершения работ на текущем этапе. Особенностями является то, что последующий этап не начнется, пока не завершится предыдущий, отсутствие возврата к предыдущим этапам, наличие результата только в конце разработки (рис. 1.2).

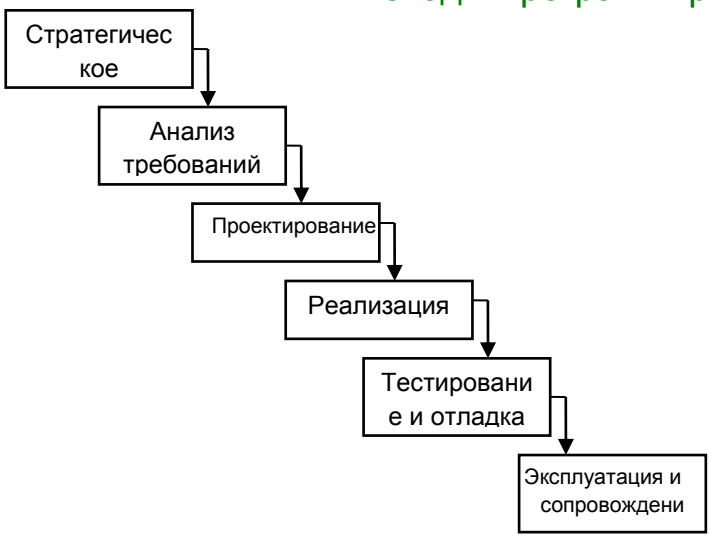


Рис. 1.2. Каскадная модель ЖЦ ПО

Каскадная модель характеризуется следующими основными особенностями:

– последовательным выполнением входящих в ее состав этапов;

– окончанием каждого предыдущего этапа до начала последующего;

– отсутствием временного перекрытия этапов (последующий этап не начнется, пока не завершится предыдущий);

– отсутствием (или определенным ограничением) возврата к предыдущим этапам;

– наличием результата только в конце разработки.

Выявление и устранение ошибок в каскадной модели производится только на стадии тестирования, которая может растянуться во времени или вообще никогда не завершиться.

2. Следующей стадией развития теории проектирования ПО стала итерационная модель ЖЦ или так называемая поэтапная модель с промежуточным контролем (рис. 1.3). Основной ее особенностью является наличие обратных связей между этапами, вследствие этого появляется возможность проведения проверок и корректировок проектируемой информационной системы на каждой стадии разработки. В результате трудоемкость отладки по сравнению с каскадной моделью существенно снижается. Итерационность модели проявляется в обработке ошибок, выявленных промежуточным контролем. Если на каком-либо этапе в ходе промежуточной проверки обнаружена ошибка, допущенная на более ранней стадии разработки, необходимо повторить весь цикл работ этой стадии. При этом анализируются причины ошибки и корректируются в случае необходимости исходные данные этапа или его содержание (последовательность действий).

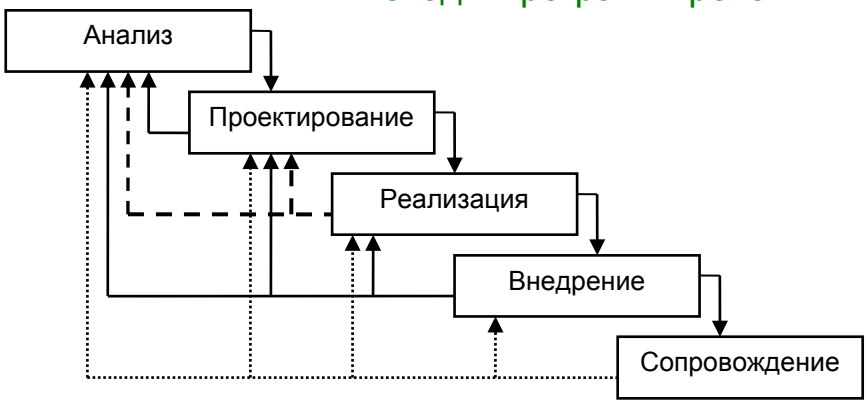


Рис. 1.3. Итерационная модель жизненного цикла ПО

К сожалению, в процессе разработки системы могут измениться начальные требования, и в этом случае итерационная модель может оказаться неэффективной.

3. Третья модель ЖЦ ПО — спиральная (spiral) модель (рис. 1.4) — поддерживает итерации поэтапной модели, но особое внимание уделяется начальным этапам проектирования: анализу требований, проектированию спецификаций, предварительному проектированию и детальному проектированию. Автор Барри Боэм (1988 г.). Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии ПО, уточняются цели и требования к программному обеспечению, оценивается качество разработанного фрагмента или версии и планируются работы следующей стадии разработки. Таким образом, углубляются и конкретизируются все детали проектируемого ПО, в результате получается продукт, который удовлетворяет всем требованиям заказчика.

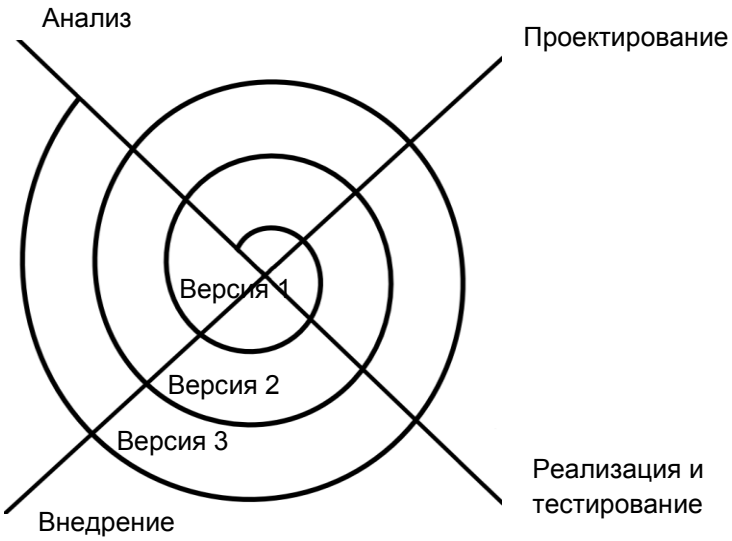


Рис. 1.4. Спиральная модель жизненного цикла ПО

4. Модель быстрой разработки приложений (Rapid application development) — пример применения инкрементной стратегии конструирования (рис. 1.5), когда в начале процесса определяются все пользовательские и системные требования, а

оставшаяся часть конструирования выполняется в виде последовательности версий (первая версия реализует часть запланированных возможностей и т.д., пока не будет получена полная система).

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD — высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60-90 дней). RAD-подход ориентирован на работку информационных систем и выделяет следующие этапы:

– **бизнес-моделирование**. Строят модели информационных потоков между бизнес-функциями, показывающие руководителей бизнес-процессов, генерируемую информацию в процессе выполнения функции, кем генерируется, где информация применяется, кто обрабатывает ее;

– **моделирование данных**. Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются свойства (атрибуты) каждого объекта, определяются отношения между объектами;

– **моделирование обработки**. Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов-данных;

– **генерация приложения**. Предполагается использование методов, ориентированных на проблемно-ориентированные языки, оперирующие конкретными понятиями узкой области. RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;

— **тестирование и объединение**. Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть *протестированы).*

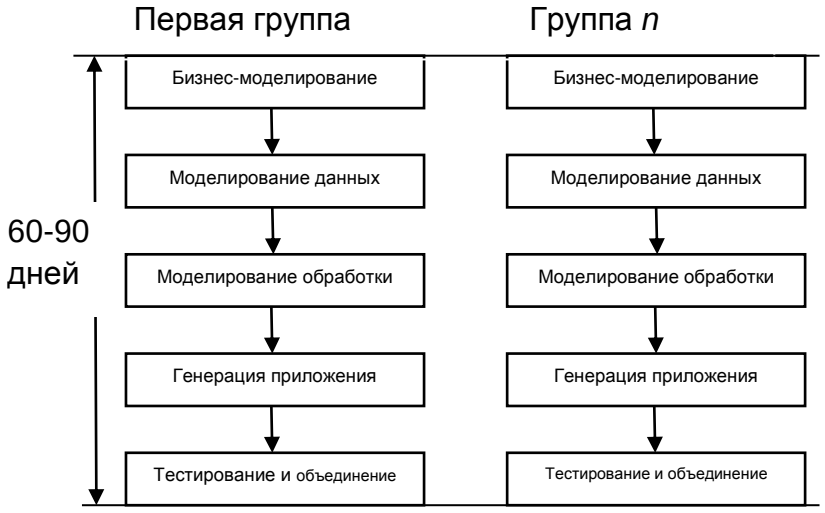


Рис.1.5. Модель быстрой разработки приложений

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет ряд недостатков и ограничений:

1) для больших проектов требуются существенные трудовые ресурсы (необходимо создать достаточное количество групп);

2) RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной;

3) RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

1.3 Экстремальное программирование

Экстремальное программирование (eXtreme Programming, XP) — методология быстрой разработки программного обеспечения. Состоит из набора методик и принципов, позволяющих как по отдельности, так и в комплексе оптимизировать процесс разработки. Главный автор — Кент Бек, 1999 г.

XP-процесс ориентирован на группы малого и среднего размера, строящие ПО в условиях неопределенных или быстро изменяющихся требований. XP-группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP — устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов (решение типичной проблемы в определенном контексте) и реляционных баз данных. Поэтому XP-процесс должен быть высокодинамичным процессом. XP-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций.

Четырьмя базовыми действиями в XP-цикле являются: кодирование, тестирование, выслушивание заказчика и проектирование. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

В экстремальном программировании четко описаны роли участников проекта. Каждая роль предусматривает характерный набор прав и обязанностей. Здесь существуют две ключевые роли: заказчик и разработчик.

Заказчик — человек или группа людей, заинтересованных в создании конкретного программного продукта. Он имеет следующие права и обязанности: зафиксировать сроки выпуска версий продукта; принимать решения относительно запланированных составляющих программы; знать ориентировочную стоимость каждой функциональной составляющей; принимать важные бизнес-решения; знать текущее состояние системы; изменять требования к системе, когда это действительно важно.

Разработчик — один или группа от двух до десяти человек, занимающихся непосредственно программированием и сопутствующими инженерными вопросами. Разработчик наделен следующими правами и обязанностями: получать достаточные сведения о реализуемых функциях; иметь возможность выяснения деталей в процессе разработки; предоставлять ориентировочные оценки трудозатрат на каждую функциональную часть или историю пользователя; корректировать оценки трудозатрат в пользу более точных в процессе разработки; предоставлять оценки рисков, связанных с использованием конкретных технологий.

Каждая из базовых ролей экстремального программирования может быть уточнена более мелкими ролями. В XP разрешено одному человеку совмещать несколько ролей.

Сторона заказчика имеет следующие подроли:

– *составитель историй* — специалист предметной области, обладающий способностями доступно изложить и описать требования к разрабатываемой системе. Этот человек или группа людей ответственны за написание историй пользователя и прояснения недопонимания со стороны программистов;

– *приемщик* — человек, контролирующий правильность функционирования системы. Хорошо владеет предметной областью. В обязанности входит написание приемочных тестов;

– *большой босс* — следит за работой всех звеньев, от разработчиков до конечных пользователей. Он контролирует внедрение системы и сопутствующие организационные моменты. Может быть также инвестором проекта.

Сторона разработчика имеет следующие подроли:

– *программист* — человек, занимающийся кодированием и проектированием на низком уровне. Он достаточно компетентен для решения текущих задач разработки и предоставления правдивых оценок запланированным задачам;

– *инструктор* — опытный разработчик, хорошо владеющий всем процессом разработки и его методиками. Несет ответственность за обучение команды аспектам процесса разработки. Внедряет и контролирует правильность выполнения методик используемого процесса. Обращает внимание команды на важные, но по каким-то причинам упущенные моменты разработки. Вместе с тем инструктор, как и любой другой человек, ограничен в знаниях, и поэтому со вниманием относится к идеям других членов команды;

– *наблюдатель* — член команды разработчиков, пользующийся доверием всей группы, который следит за прогрессом разработки. Он сравнивает предварительные оценки трудозатрат и реально потраченные, выводя количественные показатели работы команды. Часть этой информации тактично предоставляется разработчикам для знания состояния проекта, мест возникновения затруднений и планирования работ;

– *дипломат* — коммуникабельная личность, инициирующая общение между членами команды. Так как документооборот минимизирован, важно постоянное общение и передача опыта внутри команды, лучшее понимание требований к системе. Дипломат регулирует и упрощает общение между заказчиками и

разработчиками. Он препятствует недомолвкам и ненужным ссорам. Дипломат не может навязывать своего мнения дискуссирующим.

Также может присутствовать внешняя роль — *консультант* — специалист, обладающий конкретными техническими навыками, для помощи разработчикам в трудно разрешимых задачах. Обычно привлекается со стороны.

Экстремальное программирование включает двенадцать основных приемов, которые могут быть объединены в четыре группы:

1) короткий цикл обратной связи (fine scale feedback): разработка через тестирование (test driven development), игра в планирование (planning game), заказчик всегда рядом (whole team, иногда этот прием называют «локальный заказчик» — onsite customer), парное программирование (pair programming);

2) непрерывный, а не пакетный процесс: непрерывная интеграция (continuous integration), реорганизация (refactoring, также называют рефакторингом), частые небольшие релизы (small releases, также называют частой сменой версий);

3) понимание, разделяемое всеми: простое проектирование (simple design), метафора системы (system metaphor), коллективное владение кодом (collective code ownership) или выбранными шаблонами проектирования (collective patterns ownership), стандарт кодирования (coding standard or coding conventions);

4) социальная защищенность программиста (programmer welfare) выражена в 40-часовой рабочей недели (sustainable pace, forty hour week).

Рассмотрим используемые в экстремальном программировании приемы более подробно.

1. **Разработка через тестирование** — непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций.

Суть разработки через тестирование состоит в следующем. Сначала пишется тест, который изначально не проходит (т.к. логики, которую он должен проверять, еще просто не существует), затем реализуется логика, необходимая для того, чтобы тест прошел. Такой подход позволяет писать код более удобный в использовании, потому что при написании теста, когда логики еще нет, проще всего позаботиться об удобстве будущей системы.

В XP особое внимание уделяется двум разновидностям тестирования: тестирование модулей (unit testing) и приемочное тестирование (acceptance testing).

Разработчик не может быть уверен в правильности написанного им кода до тех пор, пока не сработают абсолютно все тесты модулей разрабатываемой им системы. Тесты модулей позволяют разработчикам убедиться в том, что их код работает корректно. Они также помогают другим разработчикам понять, зачем нужен тот или иной фрагмент кода и как он функционирует. Тесты модулей также позволяют разработчику без каких-либо опасений выполнять реорганизацию.

Приемочные тесты позволяют убедиться в том, что система действительно обладает заявленными возможностями. Кроме того, приемочные тесты позволяют проверить корректность функционирования разрабатываемого продукта.

Если обнаруживается ошибка, то создается тест, чтобы предотвратить ее повторное появление. Ошибка, произошедшая в рабочей системе (уже

установленной), требует написания функционального теста. Создание функционального теста непосредственно перед диагностикой ошибки позволяет заказчикам четко описать проблему и довести эту проблему до разработчиков. Невыполнившийся функциональный тест требует создания unit-теста. Это помогает сфокусировать усилия по отладке и четко показывает, когда ошибка исправлена.

2. **Игра в планирование —** быстрое определение плана работ над следующей реализацией путем объединения деловых приоритетов и технических оценок. Заказчик формирует пожелания, приоритетность и сроки выпуска одной или нескольких версий программного продукта, а разработчики ответственны за принятие технических решений и оценку (прослеживание) продвижения (прогресса). План работ обновляется его по мере того, как условия задачи становятся все более четкими.

3. **Заказчик всегда рядом** (**локальный заказчик**). В группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков. В данном случае, заказчик — конечный пользователь программного продукта, эксперт предметной области, член команды разработчиков, а не просто помощник.

4. **Парное программирование**. Весь код пишется двумя программистами, работающими на одном компьютере. Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура и манипулятор свободно передаются от одного программиста к другому. Если по какой-то причине один из пары пропустил что-то (например, болел), он обязан просмотреть все изменения, сделанные другим. В течение работы над проектом пары не фиксированы: рекомендуется их перемешивать, чтобы каждый программист в команде имел хорошее представление о всей системе. Это можно отнести к **коллективному владению кодом**. Может показаться, что парное программирование удваивает ресурсы, но исследования доказали: парное программирование приводит к повышению качества и уменьшению времени цикла. Таким образом, парное программирование усиливает взаимодействие внутри команды, пары обычно находят более оптимальные решения, существенно увеличивается качество кода, снижается число ошибок и затраты на сопровождение, ускоряется обмен знаниями между разработчиками.

5. **Непрерывная интеграция** — система интегрируется и строится много раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности. При выполнении интеграции разрабатываемой системы достаточно часто, можно избежать большей части связанных с этим проблем. В традиционных методиках интеграция, как правило, выполняется в самом конце работы над продуктом, когда считается, что все составные части разрабатываемой системы полностью готовы. В XP интеграция кода всей системы выполняется несколько раз в день, после того, как разработчики убедились в том, что все тесты модулей срабатывают корректно. Разработчики, по возможности, должны выпускать свой код каждые

несколько часов и интегрировать его. В любом случае никогда нельзя держать изменения дольше одного дня. Частая интеграция позволяет избежать отчуждения и фрагментирования в разработке, когда разработчики не могут общаться в смысле обмена идеями или повторного использования кода. Каждый должен работать с самой последней версией.

6. **Реорганизация** — способ улучшения программного кода, без изменения его функциональности; цель — устранить дублирование, избыточность, неиспользуемый код, улучшить взаимодействие, упростить систему или добавить в нее гибкость. XP-процесс подразумевает, что однажды написанный код в процессе работы над проектом почти наверняка будет неоднократно переделан. Ясный и понятный код легче модифицировать и расширять.

7. **Частые небольшие релизы** (**частая смена версий**) — быстрый запуск в производство простой версии системы (release). Новые версии реализуются в очень коротком (двухнедельном или как можно раньше) цикле. При этом каждая версия должна быть достаточно осмысленной с точки зрения полезности для бизнеса. Игра в планирование и частая смена версий зависят от заказчика, обеспечивающего набор «историй» (коротких описаний), характеризующих работу, которая будет выполняться для каждой версии системы. Версии генерируются каждые две недели, поэтому разработчики и заказчик должны прийти к соглашению о том, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчику, характеризует совокупность (пул) историй; но для следующей двухнедельной итерации из пула выбирается подмножество наиболее важных историй. Заказчик затем решает, какие задачи имеют более высокий приоритет.

8. **Простое проектирование**. Проектирование выполняется настолько просто, насколько это возможно в данный момент. В процессе работы условия задачи могут неоднократно измениться, а значит, разрабатываемый продукт не следует проектировать заблаговременно целиком и полностью. XP предполагает, что проектирование — это настолько важный процесс, что его необходимо выполнять постоянно в течение всего времени работы над программным продуктом. Проектирование должно выполняться небольшими этапами, с учетом постоянно изменяющихся требований.

9. **Метафора системы** — это аналог того, что в большинстве методик разработки ПО называется архитектурой, т.е. представлением о компонентах системы и их взаимосвязях. Поскольку XP-процесс предлагает непрерывное перепроектирование (с помощью реорганизации), при котором нет нужды в детализированной проектной документации, а для инженеров сопровождения единственным надежным источником информации является программный код, вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система. Метафора дает команде глобальное «видение» проекта, представление о том, каким образом система работает в настоящее время, в каких местах добавляются новые компоненты и какую форму они должны принять. Подбор хорошей метафоры облегчает для группы разработчиков понимание того, каким образом устроена система. Кроме того, метафора позволяет исключить дублирующийся код за счет введения имен объектов системы таким образом,

чтобы каждый член команды мог пользоваться ею без специальных знаний о системе.

10. **Коллективное владение кодом или выбранными шаблонами проектирования** — любой разработчик может улучшать любой код системы в любое время. Непрерывная интеграция, регрессионное тестирование и парное программирование обеспечивают защиту от проблем, возникающих при коллективном владении кодом. Важное преимущество рассматриваемого приема в том, что оно ускоряет процесс разработки, поскольку обнаруживаемые ошибки может устранять любой программист.

11. **Стандарты кодирования**. Должны выдерживаться правила (не важно какие), обеспечивающие одинаковое представление программного кода во всех частях программной системы (например, форматирование кода, именование классов, переменных, констант, стиль комментариев). Если в команде не используются единые стандарты кодирования, разработчикам становится сложнее выполнять реорганизацию; при смене партнеров в парах возникает больше затруднений; в целом, продвижение проекта затрудняется. В рамках XP необходимо добиться того, чтобы было сложно понять, кто является автором того или иного кода, — вся команда работает унифицировано как один человек. Команда должна сформировать набор правил, а затем каждый член команды должен следовать этим правилам в процессе кодирования. Перечень правил не должен быть исчерпывающим или слишком объемным. Стандарт кодирования вначале должен быть простым, затем он может постепенно усложняться по мере наработки опыта группой разработчиков.

12. **40-часовая рабочая неделя**. Как правило, над проектом работают не более 40 часов в неделю. Нельзя увеличивать рабочую неделю за счет сверхурочных работ. Этот принцип дает социальную защищенность команды.

Большинство принципов, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования и т. д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. В XP эти принципы, как показано в табл. 1.1, достигают «экстремальных значений».

Таблица 1.1.

Экстремумы в экстремальном программировании

Практика здравого смысла	XP-экстремум	XP-реализация
Проверки кода	Код проверяется все время	Парное программирование
Тестирование	Тестирование выполняется все время, даже с помощью заказчиков	Тестирование модуля, функциональное тестирование
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация

Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую функциональность	Самая простая вещь, которая могла бы работать
Архитектура	Каждый постоянно работает над уточнением архитектуры	Метафора
Тестирование интеграции	Интегрируется и тестируется несколько раз в день	Непрерывная интеграция
Короткие итерации	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Игра в планирование

1.4 Критерии качества программного обеспечения

Качество (*quality*) ПО — это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей. Качество ПО является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Поэтому при описании качества ПО, прежде всего, должны быть фиксированы *критерии* отбора требуемых свойств ПО. В настоящее время *критериями качества ПО* (criteria of software quality) принято считать: функциональность, надежность, легкость применения, эффективность, сопровождаемость, мобильность.

Функциональность — это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей.

Надежность (reliability) — это способность ПС безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под *отказом* в ПО понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок — важно лишь, чтобы эти ошибки при практическом применении проявлялись достаточно редко. Убедиться, что ПО является надежным можно при его тестировании, а также при практическом применении. Таким образом, можно разрабатывать лишь надежные, а не правильные ПО. Степень надежности можно характеризовать вероятностью работы ПО без отказа в течение определенного периода времени. При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

Методы программирования

Основным методом обеспечения надежности программного средства является борьба со сложностью. Известны два общих метода борьбы со сложностью систем:

- обеспечения независимости компонентов системы;
- использование в системах иерархических структур.

Обеспечение независимости компонент означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование. Использование иерархических структур допускает связи только между компонентами, принадлежащими смежным уровням иерархии. Этот метод означает разбиение большой системы на подсистемы путем абстрагирования.

При реализации ПО сначала необходимо обеспечить требуемую функциональность и надежность ПС, а затем уже доводить остальные критерии качества до приемлемого уровня.

Легкость применения — это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПО и оценке полученных результатов, а также вызывать положительные эмоции пользователя. Легкость применения определяется составом и качеством пользовательской документации, а также некоторыми свойствами, реализуемыми программным путем (например, *пользовательский интерфейс)*.

Пользовательский интерфейс представляет средство взаимодействия пользователя с ПС. При разработке пользовательского интерфейса следует учитывать потребности, опыт и способности пользователя. Поэтому потенциальные пользователи должны быть вовлечены в процесс разработки такого интерфейса.

При разработке интерфейса необходимо соблюдать следующие принципы: базирование на терминах и понятиях, знакомых пользователю; единообразность; возможность пользователю получать справочную информацию как по запросу, так и генерируемую ПО.

В настоящее время широко распространены командные и графические пользовательские интерфейсы.

Эффективность — это отношение уровня услуг, предоставляемых ПО пользователю при заданных условиях, к объему используемых ресурсов.

На эффективность ПС влияет выбор способа представления данных и выбор алгоритмов, а также особенности их реализации (включая выбор языка программирования. При этом постоянно приходится разрешать противоречие между *временно́й эффективностью* и *эффективностью по памяти* (*ресурсам*).

Для отыскания критических модулей с точки зрения временно́й эффективности ПО потребуется получить распределение по модулям времени работы ПО путем соответствующих измерений во время выполнения ПО. Это может быть сделано с помощью динамического анализатора (специального программного инструмента), который может определить частоту обращения к каждому модулю в процессе применения ПО.

Сопровождаемость — это характеристика ПО, которая позволяет минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей. Обеспечение сопровождаемости ПО сводится к обеспечению изучаемости ПО и к обеспечению его модифицируемости.

Мобильность — это способность ПО быть перенесенным из одной среды (окружения) в другую, в частности, с одного компьютера на другой.

Критерии качества являются противоречивыми: хорошее обеспечение одного какого-либо критерия качества может существенно затруднить или сделать невозможным обеспечение некоторых других. Поэтому существенная часть процесса обеспечения качества состоит из поиска приемлемых компромиссов.

ЛЕКЦИЯ №2

1.5 Объектно-ориентированное программирование

Объектно-ориентированное (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса во время исполнения программы.

Основные понятия объектно-ориентированного программирования:

Абстрагирование — выделение набора значимых характеристик объекта, исключая из рассмотрения незначимые.

Инкапсуляция — объединение данных и методов, работающих с ними, в классе с целью скрытия деталей реализации от пользователя класса (под пользователем класса понимается программист, который при написании программы использует возможности класса через методы класса (интерфейс) не зная деталей реализации).

Наследование — возможность создавать новые определения классов на основе существующих, расширяя и переопределяя их функциональность. Наследование используется для повторного использования кода. Класс, от которого производится наследование, называется базовым, родительским или суперклассом, новый класс — потомком, наследником или производным классом.

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм позволяет поддерживать выполнение нужного действия в зависимости от типа объекта и применяется для универсальной обработки схожих объектов разных типов.

1. Объявление класса на языке C#

Для реализации инкапсуляции в языке C# используются модификаторы доступа. Язык C# поддерживает следующие модификаторы доступа (модификаторы видимости):

— **public** — поля, свойства и методы являются общедоступными;

— **private** — поля, свойства и методы будут доступны только в классе, в котором они определены;

— **protected** — поля, свойства и методы будут доступны как в классе, в котором они определены, так и в любом производном класса;

— **internal** — поля, свойства и методы будут доступны во всех классах внутри сборки, в которой определен класс.

Для контроля доступа к полям класса в языке C# можно использовать свойства. Внутри класса свойство определяется в виде пары методов для

присвоения значения свойств и для чтения значения свойства. Для пользователей объектов класса свойство представляется как поле класса. Синтаксис определения свойства класса:

```
<тип>  <имя_свойства> {
get {
return <значение>;
}
set {
<имя_поля> = value;
}
}
```

Метод **get** используется для получения значения свойства. Метод **set** используется для изменения значения свойства. Внутри метода **set** новое значение свойства передается в виде переменной **value**. С помощью свойств можно проверить новое значение поля, прежде, чем изменить поле; выполнить дополнительные действия при изменении или чтении поля, представить для пользователя в виде поля информацию, которая на самом деле не является полем класса.

Язык C# не поддерживает множественное наследование.

Для реализации классического наследования в языке C# используется следующий синтаксис:

```
class <имя_класса_потомка> : <имя_класса_родителя> {

}
```

Для вызова в конструкторе класса-потомка конструктора класса-родителя используется ключевое слово **base**:

```
class <имя_класса_потомка> : <имя_класса_родителя> {
<имя_класса_потомка> (<аргументы>) : base(<аргументы>) {

}
}
```

Также ключевое слово **base** используется для вызовов методов класса-родителя из переопределенных методов в классе-потомке. Язык C# также поддерживает наследование включением.

Для реализации полиморфизма, основанного на виртуальных методах, в языке C# используются ключевые слова **virtual** и **override**:

```
class  <имя_класса_родителя> {
virtual <имя_метода1> (аргументы) {

}
}
class <имя_класса_потомка> : <имя_класса_родителя> {
override <имя_метода1> (<аргументы>) {

}
```

}

Кроме того, язык C# поддерживает объявление абстрактных методов с помощью ключевого слова **abstract**. Абстрактные методы не содержат реализации и должны быть переопределены с помощью ключевого слова **override** в классах-потомках. Класс, в котором объявлен хотя бы один абстрактный метод, является абстрактным и должен быть объявлен в помощью ключевого слова **abstract**. Объекты абстрактного класса не могут быть созданы.

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными.

```
class <имя_класса> {
// Объявление переменных.
<модификатор_доступа> <имя_типа> <имя_переменной1>;
<модификатор_доступа> <имя_типа> <имя_переменной2>;
//...
<модификатор_доступа> <имя_типа> <имя_переменнойN>;
// Объявление методов.
<модификатор_доступа> <возращаемый_тип> <имя_метода1> (<параметры>) {
<тело метода1>
}
<модификатор_доступа> <возращаемый_тип> <имя_метода2> (<параметры>) {
<тело метода2>
}
//...
<модификатор_доступа> <возращаемый_тип> <имя_методаN> (<параметры>) {
<тело методаN>
}
}
```

2. Свойства класса

Для удобства программирования в класс введено понятие свойства. Свойство представляет собой совокупность поля и одного или двух методов для считывания и/или записи данных в поле. Это позволяет использовать более предпочтительный синтаксис обращения к свойству как к полю, а не через синтаксис метода и одновременно позволяет скрыть данные в классе.

Так в языке Java свойств нет и обращение, например, к полю m_*Fam* осуществляется через методы следующим образом:

```
string s;
stud a = new stud();
s=a.getFam();
a.setFam(s).
```

Метод getFam служит для считывания информации, записанной в поле m_Fam, а метод setFam служит для записи нового значения в поле m_Fam.

В данном случае класс имеет вид:

```
class stud
```

```
{
  private string m_Fam;
  public string getFam (void) { return m_Fam; }
  public void setFam (string s);
}
```

В языке C# свойство позволяет использовать более удобный синтаксис обращения к свойству (как будто это поле):

```
string s;
stud a = new stud();
s = a.Fam;
a.Fam = s;
```

Синтаксис описания свойства следующий:

```
class stud
{
private string m_Fam;
public string Fam
{
        get { return m_Fam; }
        set { m_Fam = value; }
}
}
```

Как видно полю *m_Fam* сопоставлено свойство с двумя методами: метод **get** — для считывания данных из поля и метод **set** — для записи данных в поле. Свойство является удобным способом предотвратить некорректную работу с полем, например, запись информации в неверном формате или неверном диапазоне:

```
public string Fam
{
  get { return m_Fam; }
  set
{
        //поле реально изменится, если длина больше 1 буквы
//и первая буква русская строчная
if (value.length >1 & value[0]>='А' & value[0]<='Я')
m_Fam = value;
}
}
```

Начиная с версии C# 3.0 имеется возможность использовать короткий синтаксис объявления свойства:

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

В таком синтаксисе нет возможности объявить тела методов **get** и **set**. Они принимаются по умолчанию

get { **return** *m_Fam*; }
set { *m_Fam* = **value**; }

Это не позволяет добавить проверку значения при записи в поле. Поле также описывать не нужно.

Например, класс Grajdanin содержит 3 свойства типа **string**:

```
class Grajdanin
{
public string Fam { get; set; }  //фамилия
public string Nam { get; set; }  //имя
public string Country { get; set; }  //страна
}
```

Запись

```
public string Fam { get; set; }  //фамилия
```

вводит свойство *Fam* и автоматически создаваемое поле и является более компактной и удобной формой ранее изучаемой записи

3. Методы класса и методы объекта.

Функциональность класса реализуется в методах. Методы могут быть статическими (методами класса) и методами объекта. Методы могут обрабатывать данные: поля класса и аргументы метода. Аргументы могут быть входными, аргументами ссылками (**ref)** и выходными аргументами (**out**).

Входные аргументы используются для передачи методу переменной по значению — в переменную метода передается копия аргумента. При передаче значения в функцию можно передать константное выражение.

```
void func(int a)
{
a++;
}

func (4);     //можно так вызвать функцию func
int c=0;
func (c);
Console.Write(c);     //будет 0
```

Однако для объектов (экземпляров класса) передача по значению всегда означает передачу по ссылке (для экономии памяти):

```
void func(Grajdanin a)
{
a.Fam = ”Иванов”;
}

Stud c=new Grajdanin(“Козлов”);
Console.Write(c.Fam);     //будет “Иванов”
```

Аргументы-ссылки (**ref)** позволяют передать значение по ссылке, которое метод мог бы изменить. Перед передачей аргумента **ref** в метод переменную надо инициализировать. Переменная должна иметь тот же тип, что и аргумент. Передавать константное выражение недопустимо.

```
void func(ref int a)
{
a++;
}

int c=0;
func (ref c);     //будет c=1
func (ref 0);     //нельзя — ошибка компиляции
```

Выходные аргументы (**out)** предназначены для возврата значения из метода. Отличие от параметра **ref** в том, что предварительная инициализация переменной не требуется.

```
void Sq(float a, out float sq)
{
sq = a*a;
}

float s;   //не инициализировано
Sq(4, out s);    //будет s=16
```

Поскольку функция имеет только один результат, то выходные параметры (**out**) могут использоваться для возврата из функции большего количества значений.

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов.

Перегрузка методов. Перегрузка методов в C# означает, что в классе можно определить несколько методов с одним и тем же именем при условии, что эти методы получают разное число или типы параметров.

Перегрузка методов относится к одному из способов реализации полиморфизма в C#. В общем, для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: тип или число аргументов у каждого метода должны быть разными. Совершенно недостаточно, чтобы два метода отличались только типами результата. Они должны отличаться типами или числом своих аргументов. Когда вызывается перегружаемый метод, то выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам. Модификаторы параметров **ref** и **out** также

учитываются, когда принимается решение о перегрузке метода, но отличие между ними не столь существенно.

```
class UserInfo
{
// Перегружаем метод ui
public void ui()        //1
{
Console.WriteLine("Пустой метод\n");
}

public void ui(string Name)         //2
{
Console.WriteLine("Имя пользователя: {0}",Name);
}

        public void ui(string Name, string Family)     //3
{
}

public void ui(string Name, string Family, byte Age) //4
{
}
public void ui(string Name, string Family, ref byte Age)      //5
{
}
}

UserInfo user1 = new UserInfo();
// Разные реализации вызова перегружаемого метода
user1.ui();       //вызывается метод 1
user1.ui("Александр");       //вызывается метод 2
user1.ui("Александр", "Ерохин", 26);       //вызывается метод 4
byte y = 20;
user1.ui("Александр", "Ерохин", ref y);    //вызывается метод 5
```

4. Конструкторы

Конструктор — метод, вызываемый при создании объекта.

В классе автоматически создается конструктор по умолчанию. Это правило выполняется для любого класса, не имеющего явно описанного конструктора. В случае, если программист описал хотя бы один конструктор в классе, то конструктор по умолчанию автоматически не создается.

Перегрузка конструкторов. Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами.

Одна из самых распространенных причин для перегрузки конструкторов заключается в необходимости предоставить возможность одним объектам инициализировать другие разными способами.

Когда приходится работать с перегружаемыми конструкторами, то иногда очень полезно предоставить возможность одному конструктору вызывать другой. В C# это дается с помощью ключевого слова **this**. Ниже приведена общая форма такого вызова:

<имя_конструктора> (<параметры1>) **: this**(<параметры2>)
{
<тело конструктора, которое может быть пустым>
}

В исходном конструкторе сначала выполняется перегружаемый конструктор, список параметров которого соответствует критерию <параметры2>, а затем все остальные операторы, если таковые имеются в исходном конструкторе. Ниже приведен соответствующий пример:

```
class UserInfo
{
public string Name, Family;
public byte Age;

// Используем ключевое слово this для
// создания "цепочки" конструкторов
public UserInfo() : this("None","None",0)
{

}
public UserInfo(UserInfo obj) : this(obj.Name, obj.Family, obj.Age)
{

}
public UserInfo(string Name, string Family, byte Age)
{
this.Name = Name;
this.Family = Family;
this.Age = Age;
}
}
```

Вызывать перегружаемый конструктор с помощью ключевого слова **this** полезно, в частности, потому, что он позволяет исключить ненужное дублирование кода.

5. Создание списка

Для создания списка используется класс **List**< > — обобщенный класс.

Для создания списка для хранения граждан необходимо записать

List<*Grajdanin*> *list* = **new List**<*Grajdanin*>( ),

где List<*Grajdanin* > — класс, в котором может храниться только список объектов класса *Grajdanin* или его потомков.

При таком описании создается пустой список, в который можно добавлять объекты класса *Grajdanin*, используя метод Add класса **List**, и конструктор по умолчанию класса *Grajdanin*.

Добавление нового объекта класса *Grajdanin* в список записывается следующим образом:

list.Add(**new** *Grajdanin*());
list[0].Fam = “Иванов”;
list[0].Nam = “Иван”;
list[0].Country = “Россия”;
или инициализировать поля до добавления в список так:
Grajdanin g = **new** *Grajdanin*();
g.Fam = “Иванов”;
g.Nam = “Иван”;
g.Country = “Россия”;
list.Add(*g*);
или инициализированные конструктором:
Grajdanin g = **new** *Grajdanin*(“Иванов”, “Иван”, “Россия”);
list.Add(*g*);
list.Add(“Петров”, “Петр”, “Россия”);

6. Сокращенная инициализация

В языке C#, начиная с версии 2.5 (Visual Studio 2008), имеется возможность добавлять объекты в список и инициализировать поля с помощью более компактной и удобной записи, которая выполняет то же, что и предыдущий программный код.

```
List<Grajdanin> list = new List<Grajdanin>( )
{
new Grajdanin { Fam = "Иванов", Nam = "Иван", Country ="Russia"},
new Grajdanin { Fam = "Петров", Nam = "Петр", Country ="Russia"},
new Grajdanin { Fam = "Сидоров", Nam = "Семен", Country ="Germany"},
new Grajdanin { Fam = "Кизин", Nam = "Киз", Country ="Израиль"}
};
```

Обратите внимание на то, что новая коллекция заполняется непосредственно внутри фигурных скобок без использования метода Add класса **List**.

Данный синтаксис инициализации будет работать даже при отсутствии конструктора с тремя параметрами для класса **Grajdanin**, поскольку в этом случае не вызывается конструктор.

Класс **Grajdanin** с кострукторами и методом ToString() будет иметь, например, следующий вид:

class Grajdanin

```
{
public string Fam { get; set; }
public string Nam { get; set; }
public string Country { get; set; }
public override string ToString()
  {
return Fam + " " + Nam + " " + Country;
}

// конструктор по умолчанию (в классе может быть только один)
public Grajdanin()
{
Fam = "Иванов"; Nam = "Иван"; Country = "Russia";
}
// конструктор копирования (в классе может быть только один)
public Grajdanin(Grajdanin g)
{
Fam = g.Fam; Nam = g.Nam; Country = g.Country;
}
// конструкторы с параметрами
// конструкторов может быть сколько угодно,
// если различаются типы или количество аргументов
public Grajdanin(string fam, string nam)
{
Fam = fam; Nam = nam; Country = "Russia";
}
public Grajdanin(string fam, string nam, string contry)
{
Fam = fam; Nam = nam; Country = contry;
}
}
```

7. Перекрытие и сокрытие методов

Перекрытие методов позволяет в производном классе переопределить методы базового класса, если они определены как виртуальные (**virtual**). В производном классе можно создать метод с тем же именем, теми же аргументами и той же доступностью, дописав ключевое слово **override**.

Важной особенностью перекрытия является вызов метода производного класса при приведении производного класса к базовому классу.

```
class Rod
{
        public virtual int   f(int a)  {   }
}
```

```
class Reb: Rod
{
public override int   f (int a)  {   }
}

Reb r = new Reb( );
r.f(1);  //вызывается метод f ребенка
((Rod) r).f (1);          //и здесь вызывается метод f ребенка,
        //поскольку r в действительности объект класса Reb.
```

То есть решение о выборе вызываемого метода принимается во время работы программы, а не при компиляции. Это сказывается на быстродействии программы, но позволяет, например, одним программным кодом вызывать различные методы.

Преимущество перекрытия методов при использовании списков. В ссылку на родителя можно записать ссылку на родителя и на любого потомка (ребенка). То есть:

```
Rod r = new Rod();
Rod r1 = new Reb();
```

То есть в массив или список с элементами типа родителя можно поместить элементы — ссылки на ребенка.

```
List<Rod>list=new List();
List.Add(new Rod());
List.Add(new Rod());
List.Add(new Reb());
List.Add(new Reb());
for (int i = 0; i<list.Count; i++)
  list.Draw();
```

В этом случае два раза будет вызван метод Draw родителя и два раза — ребенка. При этом программный код в цикле не нужно модифицировать независимо от того, в какой последовательности будут находиться объекты в списке.

Сокрытие методов. Даже если в базовом классе метод не был объявлен виртуальным (**virtual**), в производном классе все равно можно объявить другой метод с такой же сигнатурой. Новый метод, однако, не перекроет метод базового класса, а скроет метод базового класса. При этом компилятор, решая, какой метод вызвать, всегда будет рассматривать тип данных, на который указывает переменная, как тип данных, заданный при ее объявлении. То есть решение о том, какой метод вызвать решает компилятор.

Для скрытия метода базового класса необходимо к его определению добавить ключевое слово **new**. Этот модификатор подскажет компилятору, что программист знает о факте сокрытия:

```
public class Rod
{
```

```
public string GetFunnyString()
{
return "Родитель!";
}
}

public class Reb : Rod
{
public new string GetFunnyString()
{
return "Ребенок!”;
}
}
.
Rod Rod1;
Reb Reb1;
Rod1 = new Rod();
Console.WriteLine(Rod1.GetFunnyString());   // будет Родитель!
Rod1 = new Reb();
Console.WriteLine(Rod1.GetFunnyString());    // будет Родитель!
Reb2 = new Reb();
Console.WriteLine(Reb2.GetFunnyString());   // будет Ребенок!
}
```

8. Абстрактные классы и абстрактные методы

Абстрактные классы — это классы, экземпляр которых создать невозможно. Абстрактные классы определяются с модификатором **abstract**. Они **могут** содержать абстрактные методы.

Абстрактные методы объявляются с модификатором **abstract**, не содержат тела метода. Абстрактный метод может быть перекрыт в производных классах как абстрактный или неабстрактный метод. Если хотя бы один метод в классе абстрактный, то и класс должен быть абстрактным.

```
public abstract class Rod
{
public abstract int MyAbstractMethod();          // тело отсуствует
public abstract int MyAbstractMethod1();                  // тело отсуствует
  ...

}
public abstract class Reb : Rod
{
public override int MyAbstractMethod()
{
return 0;
}
```

```
public abstract int MyAbstractMethod1();                    // тело отсуствует

}
```

Класс Reb остался абстрактным, так как один метод является абстрактным.

```
public class Reb2 : Rod
{
public override int MyAbstractMethod()
{
return 0;
}
public override int MyAbstractMethod1()
{
return 1;
}
}
```

9. Запечатанные классы

Запечатанные (**sealed**) классы и методы можно рассматривать как противоположность абстрактным классам и методам. Объявление класса или метода запечатанным означает, что произвести наследование и перекрытие невозможно.

```
sealed class FinalClass      // наследовать не возможно
{

}
public class myClass
{
// перекрытие в дальнейшем невозможно
public sealed override FinalMethod()
}
```

То есть у класса FinalClass не может быть наследников, а у класса myClass может быть наследник, но в нем нельзя перекрыть метод FinalMethod().

10. Интерфейсы

Интерфейс — это тип, определяющий набор методов и свойств без реализации, и используемый для определения классов со сходной функциональностью. Интерфейс определяется с помощью ключевого слова interface. Синтаксис определения интерфейса:

```
interface <имя_интерфейса> {
<имя_типа_метода>  <имя_метода> (<аргументы>);
<имя_типа_свойства>  <имя_свойства> { get; set;}
}
```

Особенности использования интерфейсов:

— все методы интерфейса по определению являются открытыми, при этом запрещено использовать в определении методов модификаторы доступа;

— тип интерфейса можно использовать в объявлении параметров методов и переменных, но создавать объекты типа интерфейс нельзя.

— вместо изменения уже используемого интерфейса следует воспользоваться наследованием интерфейса;

— интерфейсы реализуются с помощью классов. Под реализацией классом интерфейса понимается написание в классе программного кода для каждого из объявленных в интерфейсе методов и свойств. Для реализации интерфейса необходимо:

— после имени класса, реализующего интерфейс, поставить двоеточие и написать имя интерфейса (если в классе необходимо реализовать несколько интерфейсов, следует разделить их имена запятыми);

— включить в класс все методы и свойства, определенные в интерфейсе;

— для каждого реализованного метода и свойства указать модификатор доступа public.

Возможность реализации одним классом нескольких интерфейсов заменяет отсутствие множественного наследования. Для получения доступа к интерфейсу объекта применяются следующие способы:

— явное приведение типа (с помощью операции **()**) —
(<имя_интерфейса>**)** <имя_объекта>;

— с помощью ключевого слова **as** —
<имя объекта> **as** <имя_интерфейса>;

— с помощью ключевого слова **is** —
if (<имя_объекта> **is** <имя_интерфейса>).

Для реализации наследования интерфейсов в языке C# используется следующий синтаксис:

interface <имя_интерфейса >: <имя_интерфейса_родителя> {
  <тело интерфейса>
}

11. Вызов базовых версий методов

Ключевое слово **base** явно указывает компилятору, что происходит обращение к методу базового класса.

```
public class CustomerAccount
{
public virtual decimal CalculatePrice (CustomerAccount account)
{
return 2000M;
}
}
public class GoldAccount : CustomerAccount
{
public override decimal CalculatePrice(CustomerAccount account)
{
```

```
return base.CalculatePrice(account) * 0.9M;
}
}
```

Методы программирования

ЛЕКЦИЯ №3

1.6 Функциональное программирование

Функциональное программирование – это ветвь программирования, при котором программирование ведется с помощью определения функций. Вы можете сказать, что любой программист давно уже программирует с помощью функций, да и не только функций, а еще и процедур, циклов, модулей, объектов. Но в функциональном программировании нет ни процедур, ни циклов, нет даже переменных. Почти одни только функции. Функциональное программирование обладает рядом очень существенных преимуществ. Традиционное программирование родилось в 40-х годах 20 века, когда велась разработка первых ЭВМ. Его основой послужила концепция фон Неймана о хранимой программе автоматических вычислений по заданному алгоритму. Существенными чертами такой программы служили, во-первых, строгая последовательность в выполнении ее элементарных шагов и, во-вторых, возможность хранения и изменения программы наряду с данными для вычислений в общей с ними памяти. Исполнение программ в первых ЭВМ сводилось к выполнению арифметическим устройством (позже оно стало называться процессором) элементарных шагов, называемых командами, которые строго последовательно производили определенные действия над арифметическими значениями или другими командами, хранящимися в оперативной памяти компьютера.

Со временем принцип последовательного исполнения стал серьезным препятствием для развития компьютерной техники. Самым узким местом вычислительных систем уже долгое время остается процессор, который последовательно исполняет элементарные команды. Конечно, скорость работы современных процессоров не сравнить со скоростью работы арифметических устройств первых ЭВМ, однако, производительность компьютеров сейчас, как и раньше, ограничена, в основном, именно центральными процессором (ЦП). Именно скорость работы ЦП имеет решающее значение при определении общей производительности компьютера. Скорость работы ЦП стала зависеть уже не столько от его архитектуры и технологических элементов, сколько просто от его размеров, потому что на скорость работы решающее влияние стала оказывать скорость прохождения сигналов по цепям процессора, которая, как известно, не может превысить скорости света. Чем меньше процессор, тем быстрее смогут внутри него проходить сигналы, и тем больше оказывается конечная производительность процессора. Размеры процессора уменьшились многократно, однако, все труднее стало отводить от такого миниатюрного устройства вырабатываемое при работе его элементов тепло. Перед производителями вычислительной аппаратуры встал очень серьезный вопрос: дальнейшее повышение производительности стало почти невозможным без изменения основополагающего принципа всего современного программирования – последовательного исполнения команд. Конечно, можно так спроектировать вычислительную систему, чтобы в ней могли одновременно работать несколько процессоров, но, к сожалению, это почти не дает увеличения производительности,

потому что все программы, написанные на традиционных языках программирования, предполагают последовательное выполнение элементарных шагов алгоритма почти так же, как это было во времена фон Неймана. Время от времени предпринимаются попытки ввести в современные языки программирования конструкции для параллельного выполнения фрагментов кода, однако языки "сопротивляются".

Проблему можно решать различными способами. Во-первых, можно попробовать написать специальную программу, которая могла бы проанализировать имеющийся программный текст и автоматически выделить в ней фрагменты, которые можно выполнять параллельно. К сожалению, такой анализ произвольного программного кода очень труден. Последовательность выполнения шагов алгоритма очень трудно предсказать по внешнему виду программы, даже если программа «хорошо структурирована». Второй способ перейти к параллельным вычислениям – это создать такой язык программирования, в котором сам алгоритм имел бы не последовательную структуру, а допускал бы независимое исполнение отдельных частей алгоритма. Но против этого восстает весь накопленный программистами опыт написания программ. Тем не менее, оказалось, что опыт написания программ, не имеющих строго последовательной структуры, на самом деле есть.

Почти одновременно с первым "традиционным" языком программирования – Фортраном появился еще один совершенно непохожий на него язык программирования – Лисп, для которого последовательность выполнения отдельных частей написанной программы была несущественной. Ветвь программирования, начатая созданием Лиспа, понемногу развивалась с начала 60-х годов 20 века и привела к появлению целой плеяды очень своеобразных языков программирования, которые удовлетворяли всем требованиям, необходимым для исполнения программ несколькими параллельными процессорами. Во-первых, алгоритмы, записанные с помощью этих языков, допускают сравнительно простой анализ и формальные преобразования программ, а во-вторых, отдельные части программ могут исполняться независимо друг от друга. Это и есть языки функционального программирования. Описание алгоритмов в функциональном стиле сосредоточено не на том, как достичь нужного результата (в какой последовательности выполнять шаги алгоритма), а больше на том, что должен представлять собой этот результат. Пожалуй, единственный серьезный недостаток функционального стиля программирования состоит в том, что этот стиль не универсальный. Многие действительно последовательные процессы, такие как поведение программных моделей в реальном времени, игровые и другие программы, организующие взаимодействие компьютера с человеком, не выразимы в функциональном стиле. Тем не менее, функциональное программирование заслуживает изучения хотя бы еще и потому, что позволяет несколько по- иному взглянуть вообще на процесс программирования, а некоторые приемы программирования, которые, вообще говоря, предназначены для написания программ в чисто функциональном стиле, могут с успехом использоваться и в традиционном программировании.

Основные преимущества языков ФП.

- Краткость программы.

- Функциональные программы поддаются формальному анализу легче своих аналогов на алгоритмических языках за счет использования математической функции в качестве основной конструкции.

- Возможность реализации на ЭВМ с параллельной архитектурой.

Особенности функционального программирования.

1. Вызов функций является единственной разновидностью действий, выполняемых в функциональной программе,

2. В алгоритмических языках программа является последовательностью операторов, вызовов процедур в соответствии с алгоритмом. В функциональном программировании программа состоит из вызовов функций и описывает то, что нужно делать и что собой представляет результат решения, а не как нужно действовать для получения результата.

3. Каждая функция в программе выдает один и тот же результат на одном и том же наборе входных данных (аргументов функции), то есть результат работы функции является «повторяемым». Вычисление функции не может повлиять на результат работы других функций, то есть функции являются «чистыми».

4. Поскольку программа состоит только из «чистых» функций, то порядок вычисления аргументов этих функций будет несущественным.

5. Вообще, любые выражения, записанные в такой программе по отдельности, независимо друг от друга (вычисление значений отдельных элементов списка, полей кортежа и т.п.) могут вычисляться в любой последовательности или даже параллельно. При наличии нескольких независимых процессоров, работающих в общей памяти, вычисления, происходящие по программе, составленной только из вызовов чистых функций, легко распараллелить. Уже это свойство функционального стиля программирования привлекает к нему значительный интерес.

6. Основными методами программирования являются суперпозиция функций и рекурсия.

7. Функциональное программирование есть программирование, управляемое данными. В строго функциональном языке однажды созданные (введенные) данные не могут быть изменены.

8. В алгоритмических языках с именем переменной связана некоторая область памяти, соответствие строго сохраняется в течение всего времени выполнения программы. В функциональном программировании переменная обозначает только имя некоторой структуры, имена символов, переменных, списков, функций и других объектов не закреплены предварительно за какими-либо типами данных. В ФП одна и та же переменная в различные моменты времени может представлять различные объекты.

9. В языках функционального программирования программа и обрабатываемые ею данные имеют единую списочную форму представления.

10. Функциональное программирование предполагает наличие функционалов – функций, аргументы и результаты которых могут быть функциями.

Пример написания программы в функциональном стиле на традиционном языке программирования

Если программа представляет из себя набор «чистых» детерминированных функций, то она будет "функциональной" независимо от того, написана ли она на специальном языке функционального программирования или на традиционном.

Рассмотрим, например, задачу вычисления числа вещественных корней заданного квадратного уравнения. Функция, решающая эту задачу должна, получив в качестве исходных данных коэффициенты квадратного уравнения, вычислить его дискриминант, а затем сформировать нужный результат в зависимости от знака 10 вычисленного дискриминанта. На языке Java функция может выглядеть следующим образом (листинг 1.1).

Листинг 1.1. Функция вычисления числа корней квадратного уравнения имеет вид:

```
int roots(double a, double b, double c)
{
double discr = b * b - 4 * a * c;
if (discr < 0) return 0;
else if (discr == 0) return 1;
else return 2;
}
```

Эта функция, конечно же, детерминированная и «чистая», однако все же с точки зрения функционального стиля она имеет одну «неправильность». Дело в том, что в функции определяется и используется локальная переменная discr, которая используется для запоминания значения дискриминанта квадратного уравнения. В данном случае это никак не влияет на «чистоту» написанной функции, но если бы внутри функции были бы определены другие функции, то результат их работы мог бы быть различным в зависимости от того, когда они вызываются – до присваивания переменной discr нового значения или после него. Поэтому чисто функциональный стиль программирования предполагает полное отсутствие присваиваний в программах. Кроме того, в чисто функциональных программах нет понятия последовательного вычисления. Каждая функция должна представлять собой суперпозицию обращений к другим функциям. В нашем же примере порядок вычислений задается строго, в нем предписывается, что сначала требуется вычислить значение дискриминанта и присвоить вычисленное значение переменной discr, потом проверить, верно ли, что полученная переменная имеет значение, меньшее нуля, и т.д. Впрочем, в нашем случае исправить программу, приведя ее в соответствие с принципами функционального стиля программирования, довольно просто. Для этого определим две вспомогательные функции для вычисления дискриминанта квадратного уравнения и для вычисления знака вещественного числа (аналогичную стандартной

Math.signum, но выдающую целый результат). В результате получится следующая программа (листинг 1.2).

Листинг 1.2. Функция вычисления корней квадратного уравнения в функциональном стиле имеет вид:

```
int roots(double a, double b, double c) { return sign(discr(a, b, c)) + 1; }
int sign (double x) { return x < 0 ? -1 : x == 0 ? 0 : 1; }
double discr (double a, double b, double c) { return b * b - 4 * a * c; }
```

Эта программа уже действительно чисто функциональная. Обратите внимание также и на то, что вместо условных операторов мы в этой программе используем условные выражения, которые соединяют условиями не отдельные части последовательно выполняющейся программы, а отдельные подвыражения. Это тоже является характерной особенностью функционального стиля программирования.

ЛЕКЦИЯ 4

1.7 Обобщенное программирование

Начиная с версии 2.0 в .NET поддерживаются обобщения. С помощью обобщений можно создавать классы и методы, независимые от хранящихся в них типов. Вместо написания множества классов и методов с одинаковой функциональностью для разных типов можно создать только один обобщенный метод или обобщенный класс.

Обобщенное программирование – парадигма программирования, заключающаяся в таком описании данных, алгоритмов и классов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Однако в отличие от, например, языка C++, в котором обобщения реализованы в виде шаблонов, для создания экземпляра шаблона необходим исходный код шаблона.

Обобщенные методы

В объявлении обобщенного метода присутствует обобщенный тип. Обобщенные методы могут быть как внутри обобщенного, так и не обобщенного класса.

Обобщенные методы в качестве типов аргументов используют открытые типы.

Например, рассмотрим процедуру *Swap* перестановки двух значений. Независимо от типа переставляемых аргументов последовательность команд будет одинаковой. Поэтому программисту удобнее написать эту процедуру один раз и разрешить пользоваться ею для перестановки аргументов разных типов. Например, обобщённая процедура перестановки местами двух значений может иметь параметр-тип, определяющий тип значений, которые она меняет местами. Это делается через открытый тип. В процедуре *Swap* тип *T* является открытым типом.

```
void Swap<T>(ref T t1, ref T t2)
{
  T t = t1;
  t1 = t2;
  t2 = t;
}
```

В тех местах программы, где обобщённый тип или функция используется, программист должен явно указать фактический параметр-тип (закрытый тип), конкретизирующий описание открытого типа.

Когда программисту нужно поменять местами

- два целых значения, он вызывает процедуру с параметром-типом «целое число» и двумя параметрами — целыми числами,

- когда две строки — с параметром-типом «строка» и двумя параметрами — строками.

```
int a = 3, b = 6;
Swap<int>( ref a, ref b);
string s1= “Hello”, s2 = “Privet”;
Swap<string>( ref s1, ref s2);
```

Закрытый тип можно не указывать, если он может быть определен по типу параметров функции:

```
Swap (ref a, ref b);
```

Альтернативой обобщенных методов и классов является использование базового класса **Object** как типа данных с последующим выполнением надлежащего приведения типов.

```
void Swap(Object t1, Object t2)
{
  Object t = t1;
  t1 = t2;
  t2 = t;
}
```

Преимущества обобщений

1. Одним из основных преимуществ обобщений является производительность. Использование типов значений с необобщенными методами вызывает упаковку (boxing) и распаковку (unboxing) при преобразовании в ссылочный тип и обратно.

Классы в C# являются ссылочными типами, а структуры и стандартные типы – типами значений (типами-значениями). Тип **Object** является ссылочным. Однако вместо него можно подставить не ссылочный тип, например, типа **int**. При присваивании объекту переменной типа **int** происходит автоматическая упаковка (преобразование типа-значения к ссылочному типу). Распаковка осуществляется при преобразовании упакованного типа значений к простому типу значений. При распаковке требуется операция приведения.

```
int i = 3;
Object o = i;  // упаковка
i = (int) o;  // распаковка с приведением типа.
```

При распаковке необходимо правильно выбрать тип, в который осуществляется распаковка.

При вызове обобщенного метода упаковка и распаковка не требуется, поскольку для разных типов создается отдельный экземпляр функции. Это повышает производительность.

2. Другим свойством обобщений является безопасность типов. В случае необобщенных методов в качестве аргументов могут быть подставлены данные любых типов, поскольку все типы наследуются от Object.

При вызове не обобщенного метода *Swap* могут возникнуть ошибки при попытке его вызвать с параметрами разных типов:

```
int i=4; double d = 3.2; Stud s = new Stud();
Swap (i, d);
```

Swap (*s*, *d*);

Обобщения также повышают повторное использование двоичного кода (в отличие от шаблонов C++), поскольку обобщения хранятся в двоичном коде и на их основе могут быть созданы экземпляры многих типов или методов.

Поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL. Однако, когда обобщенные классы компилируются JIT-компилятором в родной машинный код, для каждого конкретного типа значения создается новый класс. Ссылочные типы при этом разделяют общую реализацию одного родного класса. Причина в том, что в случае ссылочных типов каждый элемент представлен в памяти 4-байтным адресом (на 32-разрядных системах) и машинные реализации обобщенного класса с различными ссылочными типами-параметрами не отличаются друг от друга. В отличие от этого, типы значений содержатся в памяти целиком, и поскольку каждый из них требует разного объема памяти, то для каждого из них создаются свои экземпляры классов на основе обобщенного.

Если в программе используются обобщения, то полезно, когда переменные обобщенных типов легко отличить от необобщенных. Принято имена обобщенных типов начинать с буквы **T**. Если обобщенный тип метода или класса один, то имя **T** подойдет. Если обобщенных типов несколько, то имена надо давать более информативные, например, TInput, TOutput, TKey, TValue, TEvent, TAction.

Обобщенные методы с ограничениями на тип

В случае, если на тип обобщения накладываются ограничесния, то можно записат их с помощью ключевого слова **where** так:

```
public void SpeakTo<T>(T person) where T : IPerson
```

С помощью выражения

where T: IPerson

указывается, что используемый тип T обязательно должен быть классом, реализующим интерфейс *IPerson*, или самим интерфейсом *IPerson,* или его наследником.

В приведенном ниже примере в классе Speaker осуществляется обращение к методу GetFirstName() персоны. Таким образом, у типа T должен быть метод GetFirstName(). Интерфейс IPerson имеет такой метод и поэтому к типу T устанавливается требование, что он должен быть классом, унаследованным от интерфейса IPerson.

```
class Speaker
{
  public void SpeakTo<T>(T person) where T : IPerson
  {
        string fn = person. GetFamNam();
        this.say("Hello, " + fn);
```

```
  }
}

class Men
{
  public string Fam {get; set; }
  public string Nam {get; set; }
  string GetFamNam()
  {
        return Fam + " " + Nam;
  }
}
```

Обобщенные классы

Кроме обобщенных методов существуют обобщенные классы. Обобщенные классы являются открыто сконструированными типами и описываются с открытым типом.

```
class MyObj <T>  //открыто сконструированный тип MyObj и открытый тип T
{
  T t;
  MyObj(T y)
  {
        t =y;
  }
public T f()
  {
        return t;
}
public void Plus(T a)
{
        t = t + a; //Класс T должен обеспечивать операцию сложения
}
}
```

Ведь, по существу, такой обобщенный тип, как MyObj<T>, является абстракцией. Конструкция, подобная MyObj<T>, называется в C# **открыто сконструированным типом**, поскольку в ней указывается параметр открытого типа T, но не такой конкретный тип, как int.

Когда для класса MyObj указывается аргумент типа, например, **int** или **string**, то создается так называемый в C# **закрыто сконструированный тип**.

```
MyObj <int> ob = new MyObj(3);
// закрыто сконструированный тип MyObj <int>
ob.Plus(5);
```

```
Console.WriteLine(ob.f());
MyObj <string> sob = new MyObj(“Hello”);
// закрыто сконструированный тип MyObj <string>
sob.Plus(“Viktor”);
Console.WriteLine(sob.f());
```

В случае с данными программист может, например, описать обобщённый тип «список» с параметром-типом, определяющим тип хранимых в списке значений. Тогда при описании реальных списков программист должен указать обобщённый тип и параметр-тип, получая, таким образом, любой желаемый список с помощью одного и того же описания.

```
class Spisok<T>
{
  List<T> list;
  void Add (T t) { list.Add(t); }
}
```

Компилятор, встречая обращение к обобщённому типу или функции, выполняет необходимые процедуры статического контроля типов, оценивает возможность заданной конкретизации и при положительной оценке генерирует код, подставляя фактический параметр-тип на место формального параметра-типа в обобщённом описании. Естественно, что для успешного использования обобщённых описаний фактические типы-параметры должны удовлетворять определённым условиям. Если обобщённая функция сравнивает значения типа-параметра, любой конкретный тип, использованный в ней, должен поддерживать операции сравнения, если присваивает значения типа-параметра переменным — конкретный тип должен обеспечивать корректное присваивание.

Сконструированным типом считается такой обобщенный тип, для которого предоставлены все аргументы типов. Если все эти аргументы относятся к закрытым типам, то такой тип считается закрыто сконструированным. А если один или несколько аргументов типа относятся к открытым типам, то такой тип считается открыто сконструированным.

Ограничения на обобощенный тип

Ограничения задаются после ключевого слова **where** например, так:

```
class Point<T> where T: struct, P2
{

}
class Point<T> where T: class, new()
{

}
```

struct – тип T должен быть типом значений.

class – тип T должен быть ссылочным типом.

<Имя класса или интерфейса> — тип *T* должен быть наследником класса или интерфейса.

new() – класс *T* должен иметь конструктор по умолчанию.

Также можно создавать обобщенные интерфейсы.

ГЛАВА 2. Методы программирования, реализованные в ОС Windows

ЛЕКЦИЯ №5

2.1 Коллекции

Коллекция в C# – совокупность объектов. В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций. Коллекции упрощают решение различных задач программирования, требующих сложных структур данных. К ним относятся списки, стеки, динамические массивы, очереди, хеш-таблицы.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. В среде .NET Framework поддерживаются четыре типа коллекций: необобщенные, специальные, с поразрядной организацией и обобщенные.

Первоначально существовали только классы необобщенных коллекций. Затем появились обобщенные версии классов и интерфейсов.

Необобщенные коллекции оперируют данными типа **object** (базовый класс для всех классов). Таким образом, они служат для хранения объектов любого типа, причем в одной коллекции допускается наличие объектов разных типов. Это их преимущество и недостаток. Это преимущество, если нужно оперировать объектами разных типов в одном списке или если типы объектов заранее неизвестны. Но эти коллекции не обеспечивают типовую безопасность. Пространство имен **System.Collections**.

Специальные коллекции оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеется специальные коллекции для символьных строк. Пространство имен System.Collections.Specialized.

Пространство имен **System**.**Collections**.**Specialized**.

Коллекции с поразрядной организацией поддерживают поразрядные операции над двоичными разрядами (И, ИЛИ). К ним относится только класс **BitArray**. Пространство имен **System.Collections**.

Обобщенные коллекции обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связанные списки, стеки, очереди и словари. Такие коллекции являются типизированными. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Это исключает случайное несовпадение типов. Пространство имен **System**.**Collections**.**Generic**.

Также имеются классы, поддерживающие создание собственных обобщенных коллекций. Пространство имен **System**.**Collections**.**ObjectModel**.

Основополагающим для всех коллекций является понятие перечислителя, который обеспечивает стандартный способ поочередного доступа к элементам коллекций. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса **IEnumarable**, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в

интерфейсе **IEnumerator** или **IEnumerator**<**T**>. Для поочередного обращения к элементам коллекции в цикле **foreach** используется перечислитель.

С перечислителем связано другое средство — итератор.

К обобщенным коллекциям относятся:

— Dictionary<**TKey**, **TValue**> — обобщенная коллекция ключей и значений.

— LinkedList<*T*> — двусвязный список.

— List<*T*> — последовательный список элементов с динамически изменяемым размером.

— Queue<*T*> — очередь.

— SortedDictionary<*TKey*, *TValue*> — словарь, поддерживаемый в отсортированном порядке пары «ключ/значение».

— SortedSet<*TKey*, *TValue*> — коллекция объектов, поддерживаемых в отсортированном порядке без дублирования.

— Stack<*T*> — стек.

Инициализация коллекций осуществляется следующим образом:

List<**int**> *myList* = **new List**<**int**>{0,1,2,3,4,5,6,7,8};

Если контейнер управляет коллекцией классов и структур, то можно смешивать синтаксис инициализации объектов с инициализации объектов с синтаксисом инициализации коллекций, создавая некоторый функциональный код:

```
List<Point> m = new List<int>
{
  new Point {X=2, Y=3},
  new Point {X=6, Y=4},
  new Point (Color.Red) {X=2, Y=3}
};
```

Класс List<T>

Класс List<T> имеет следующие методы:

1. Метод AddRange() – добавление множества элементов в коллекцию за один прием. Метод AddRange принимает один аргумент типа **IEnumerable**<*T*>

points.AddRange(**new** Point[] { **new** Point {X=5, Y=6}, **new** Point(5,8) });

2. Метод **Insert**() – вставка элемента в определенную позицию.

points.Insert(3, **new** Point(3,5));

Если указывается индекс, превышающий количество элементов в коллекции, то генерируется исключение типа **ArgumentOutOfRangeException**.

3. Метод **InsertRange**() – вставка элементов в определенную позицию за один прием. Аналогично AddRange.

4. Доступ к элементам осуществляется с помощью индексатора []. Первый элемент доступен по индексу 0.

Point p = points[0];

4. Проход по элементам коллекции возможен с помощью оператора **foreach** блягодаря реализации интерфейса IEnumerable:

```
foreach(Point p in points)
{
```

```
  Console.WriteLine(p);
}
```

5. Метод ForEach(), который может использоваться вместо оператора **foreach** также, объявленный с параметром Action<T>. **Action** должен быть объявлен как метод с параметром того же типа, что и элемент коллекции и возвращающий **void**.

Например,

```
points.ForEach(Console.WriteLine);
points.ForEach(p=>Console.WriteLine(p));
```

6. Метод **RemoveAt**() – удаление элемента по индексу.

7. Метод **Remove**() – удаление элемента по ссылке.

Удаление элемента по ссылке работает быстрее, поскольку при этом не надо выполнять поиск удаляемого элемента по всей коллекции. Метод Remove() сначала ищет удаляемый элемент с помощью метода IndexOf(), а затем использует индекс для удаления элемента.

8. Метод **RemoveRange**() удаляет множество элементов из коллекции. Первый параметр определяет индекс, начиная с которого располагаются удаляемые элементы, а второй параметр задает количество удаляемых элементов.

```
points.RemoveRange(3,6);
```

9. Метод **RemoveAll**() – удаление всех элементов с некоторыми характеристиками, указанных предикатом.

10. Метод **Clear**() – удаление всех элементов коллекции.

11. Метод IndexOf(), LastIndexOf(), FindIndex(), FindLastIndex(), Find(), FindLast() – поиск элемента коллекции.

12. Метод **IndexOf**() – возвращает индекс элемента, указанного в качестве параметра. Также можно указать индекс первого элемента и количество элементов среди которых необходимо осуществлять поиск. Возвращает -1, если элемент не найден.

13. Метод **FindIndex**() – возвращает индекс элемента с определенными свойствами, заданными предикатом. Также можно указать индекс первого элемента и количество элементов среди которых необходимо осуществлять поиск.

```
Point points.FindIndex(p=>p.X == 5);
```

14. Метод **Find**() – возвращает элемент (а не индекс) с определенными свойствами, заданными предикатом.

15. Метод **FindAll**() – возвращает все элементы с определенными свойствами, заданными предикатом.

```
List<Point> p = points.FindIndex(p=>p.X > 5);
```

16. Метод **Exists**() – проверка существования элемента.

17. Метод **Sort**() осуществляет сортировку элементов. Метод Sort без параметров позволяет сортировать только объекты, для которых реализован интерфейс IComparable. Причем сортировка осуществляется тем способом, который предусмотрен по умолчанию типом элементов.

Сортировка элементов коллекции способом, который не поддерживается по умолчанию

Для сортировки элементов другим способом, не тем, который поддерживается по умолчанию типом элементов, необходимо использовать интерфейс IComparer<T> для типа элемента коллекции. Интерфейс IComparer<T> определяет метод Compare(), который необходим для сортировки. В реализации этого метода используется метод CompareTo() типов string и int.

```
public class PointComparer: IComparer<Point>
{
  public enum CompareType
  {
        X,
        Y,
        Name
  }
  private CompareType compareType;

  public PointComparer(CompareType compareType)
  {
        this.compareType = compareType;
  }

  public int Compare(Point a, Point b)
  {
        if(a==null) throw new ArgumentNullException("a");
        if(b==null) throw new ArgumentNullException("b");
        int result;
        switch(compareType)
        {
              case CompareType.X: return a.X.CompareTo(b.X);
              case CompareType.Y:
                    result = a.Y.CompareTo(b.Y);
                    if (result==0) return a.X.CompareTo(b.X);
                    else return result;
              case CompareType.Name: return a.Name.CompareTo(b.Name);
              default: throw new ArgumentException("Недопустимое поле для сравнения");
        }
  }
```

Теперь коллекцию точек можно отсортировать по X, Y или Name.

```
points.Sort(new PointComparer(PointComparer.CompareType.X));
points.Sort(new PointComparer(PointComparer.CompareType.Y));
points.Sort(new PointComparer(PointComparer.CompareType.Name));
```

Класс Stack<T>

Класс Stack<T> представляет коллекцию элементов, работающую по алгоритму «последний вошел – первый вышел» (LIFO) и имеет следующие методы:

Метод **Push**() — вставка элемента в стек. Вставка элементов осуществляется в вершину стека, так же как и извлечение.

Метод **Pop**() — извлечение элемента из стека.

Метод **Peek**() — возвращает элемент из вершины стека без его удаления.

При попытке извлечения элемента из пустого стека генерируется исключение **InvalidOperationException**.

Класс Queue<T>

Класс Queue<T> представляет коллекцию элементов, работающую по алгоритму «первый вошел – первый вышел» (FIFO) и имеет следующие методы:

Метод **Enqueue** () — вставка элемента в очередь. Вставка осуществляется в конец очереди.

Метод **Dequeue**() — извлечение элемента из очереди. Извлечение осуществляетися из начала очереди.

Метод **Peek**() – просмотреть элемент из начала очереди без его удаления.

При попытке извлечения элемента из пустой очереди генерируется исключение **InvalidOperationException**.

Класс SortedSet<T>

Класс SortedSet<T> удобен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. При создании объекта SortedSet, его конструктору необходимо передать объект, реализующий интерфейс IComparer<T>, который будет информировать о том, как будут сортироваться объекты.

```
SortedSet<Point> points =
new SortedSet<Point> ( new PointComparer(PointComparer.CompareType.X) );
```

Класс ObservableCollection

Класс **ObservableCollection**<T> — коллекция аналогичная List<T>, но генерирует событие при изменении содержимого коллекции. То есть она информирует внешние объекты о том, что произошло изменение коллекции. Пространство имен **System.Collections.ObjectModel**.

2.2 Исключения

В результате работы программы могут произойти непредвиденные ситуации, например, деление на 0 или переполнение разрядной сетки переменной.

Далеко не всегда ошибки случаются по вине того, кто кодирует приложение. Иногда приложение генерирует ошибку из-за действий конечного пользователя, или же ошибка вызвана контекстом среды, в

которой выполняется код. В любом случае программист всегда должен ожидать возникновения ошибок в своих приложениях и проводить кодирование в соответствии с этими ожиданиями.

В языке C# ошибки в программе во время выполнения передаются через программу посредством механизма, называемого исключениями. Исключения создаются кодом, который встречает ошибку и перехватываются кодом, который может исправить ее. Исключения могут создаваться средой CLR платформы .NET Framework или кодом в программе.

В .NET Framework предусмотрена развитая система обработки ошибок. Механизм обработки ошибок C# позволяет закодировать пользовательскую обработку для каждого типа ошибочных ситуаций, а также отделить код, потенциально порождающий ошибки, от кода, обрабатывающего их.

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов **try** и **catch**. Эти ключевые слова действуют совместно и не могут быть использованы порознь. Ниже приведена общая форма определения блоков **try/catch** для обработки исключительных ситуаций:

```
try {
// Блок кода, проверяемый на наличие ошибок.
}

catch (<тип_исключения> exOb) {
// Обработчик исключения типа ExcepType1.
}
catch (<тип_исключения> exOb) {
// Обработчик исключения типа ExcepType2.
}
...
```

где <тип_исключения> — это тип возникающей исключительной ситуации. Когда исключение генерируется оператором try, оно перехватывается составляющим ему пару оператором catch, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор catch. Так, если типы генерируемого исключения и того, что указывается в операторе catch, совпадают, то выполняется именно этот оператор, а все остальные пропускаются. Когда исключение перехватывается,

переменная исключения exOb получает свое значение. На самом деле указывать переменную exOb необязательно. Так, ее необязательно указывать, если обработчику исключений не требуется доступ к объекту исключения, что бывает довольно часто. Для обработки исключения достаточно и его типа.

Для включения проверки переполнения в локальном месте программы можно использовать ключевое слово **checked**, а для отлавливания исключений **try … catch**.

```
private void button1_Click(object sender, EventArgs e)
{
int a = 20;
int s=1;
try
{
for (int i = 1; i <= a; i++)
checked { s *= i; }
}
catch (System.OverflowException)
{
//этот код будет выполнен при возникновения исключения
//System.OverflowException - переполнения
MessageBox.Show("Слишком большое число");
}
catch ( )
{
//этот код будет выполнен при возникновения любого
//другого исключения, не отловленного выше
MessageBox.Show("Какое-то исключение");
}
Text = s.ToString();
}
```

Внутри оператора **try** помещается код, в котором могут генерироваться исключения, а в блоке **catch** они обрабатываются.

Проверка двух исключнений

```
private void button1_Click(object sender, EventArgs e)
{
int a = 2000000;
int s=2;
```

```
try
{
a = a / s;
checked { a = a * 100000000; }
}
catch (System.OverflowException)
{
MessageBox.Show("Переполнение");
}
catch (System.DivideByZeroException)
{
MessageBox.Show("Деление на ноль");
}
Text = a.ToString();
}
```

Если s=2, то произойдет исключение переполнения, если s=0 – то исключение деления на 0. Как только происходит исключение – прерывается выполнение блока try и управление передается в блок **catch**. То есть при делении на 0, умножение выполняться не будет и a останется равной 20.

Стандартные исключения

Тип исключения	Описание
Exception	Базовый класс для всех объектов исключений
SystemException	Базовый класс для всех ошибок, генерируемых во время выполнения
IndexOutOfRangeException	Генерируется во время выполнения при выходе индекса массива за границы
NullReferenceException	Генерируется во время выполнения при ссылке на объект null
InvalidOperationException	Генерируется определенными методами, если вызов метода недопустим для текущего состояния объекта
ArgumentException	Базовый класс для всех исключений аргументов
ArgumentNullException	Генерируется методами, если аргумент равен null, что запрещено
ArgumentOutOfRangeException	Генерируется методами, если аргументы выходит из заданного диапазона

Программная генерация исключений.

Программист может прописать генерацию исключения с помощью ключевого слова **throw**, указав определенное исключение

throw new ArgumentOutOfRangeException(“Аргумент должен быть больше 5”);

```
class Stol{
private float dl;
public float dlina
{
set
{
if (value<=0)
throw new
ArgumentOutOfRangeException(“Аргумент должен быть больше 0”);
dl=value;
}
get {return dl; }
}
}
```

```
class Program
{
Stol s = new Stol( );
void main()
{
try{
s.dlina = -100;

}
catch(ArgumentOutOfRangeException)
{
                    MessageBox(“Длина не допустимая”);
                    s.dlina = 1;
}
}
}
```

Оператор **throw** можно включить в блок **catch**, чтобы заново вызвать исключение, перехваченное блоком **catch**. В следующем примере извлекаются сведения об источнике из исключения **IOException**, а затем это исключение вызывается для родительского метода:

```
catch (FileNotFoundException e)
{
// FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
// Извлекаем информацию об исключении и вызываем родительский метод
// т.к. IOException родитель FileNotFoundException
if (e.Source != null)
Console.WriteLine("IOException source: {0}", e.Source);
throw;
}
```

Вы можете перехватывать одно исключение и сгенерировать другое исключение, как показано в следующем примере:

```
catch (InvalidCastException e)
```

```
{
// Выполняем некоторый код здесь и затем генерируем новое исключение
throw new YourCustomException ("Put your error message here.", e);
}
```

Также можно повторно вызывать исключение при выполнении указанного условия, как показано в следующем примере:

```
catch (InvalidCastException e)
{
if (e.Data = = null)
{
throw;
}
else
{
// Выполняем некоторые действия здесь
}
}
```

В блоке **try** инициализируйте только те переменные, которые в нем объявлены. В противном случае до завершения выполнения блока может возникнуть исключение. Например, в следующем примере кода переменная *n* инициализируется внутри блока **try**. Попытка использовать данную переменную вне этого блока **try** в инструкции Write(*n*) приведет к ошибке компилятора.

```
static void Main()
{
int n;
try
{
// Переменная n не описана в try, поэтому
// не надо инициализировать в try …
n = 123;
}
catch
{

}
// … так как в случае возникновения исключения здесь
//переменная n не будет инициализирована.
```

Console.Write(*n*);

Общие сведения об исключениях

Исключения имеют следующие свойства.

Исключения имеют типы, в конечном счете являющиеся производными от System.Exception.

Следует использовать блок try для заключения в него инструкций, которые могут выдать исключения.

При возникновении исключения в блоке try поток управления немедленно переходит к первому соответствующему обработчику исключений, присутствующему в стеке вызовов. В языке C# ключевое слово catch используется для определения обработчика исключений.

Если обработчик для определенного исключения не существует, выполнение программы завершается с сообщением об ошибке.

Не перехватывайте исключение, если его нельзя обработать, и оставьте приложение в известном состоянии. При перехвате System.Exception вновь инициируйте это исключение с использованием ключевого слова throw в конце блока catch.

Если в блоке catch определяется переменная исключения, ее можно использовать для получения дополнительной информации о типе произошедшего исключения.

Исключения могут явно генерироваться программной с помощью ключевого слова throw.

Объекты исключения содержат подробные сведения об ошибке, такие как состояние стека вызовов и текстовое описание ошибки.

Код в блоке finally выполняется, даже при возникновении исключения. Блок finallyиспользуется для освобождения ресурсов, например для закрытия потоков или файлов, открытых в блоке try.

Управляемые исключения в платформе .NET Framework реализуются в начале механизма структурированной обработки исключений Win32. Дополнительные сведения см. в разделе Структурированная обработка исключений и в документе Подробное руководство по структурированной обработке исключений Win32.

При генерации исключений следует избегать следующего:

Исключения не рекомендуется использовать для изменения потока программы в рамках обычного выполнения. Исключения используются только для сообщения о состояниях ошибки и их обработки.

Исключения не должны возвращаться в качестве возвращаемого значения или параметра вместо генерации.

Не рекомендуется специально генерировать System.Exception, System.SystemException, System.NullReferenceException или System.IndexOutOfRangeException из собственного исходного кода.

Не рекомендуется создавать исключения, которые могут быть сгенерированы в режиме отладки, а не в режиме выпуска. Чтобы определить ошибки времени выполнения на этапе разработки, используйте Debug Assert.

Программы могут генерировать предопределенный класс исключений в пространстве имен System (если специально не обозначено иное), или создавать собственные классы исключений путем наследования от Exception. Производные классы должны определять, по меньшей мере, четыре конструктора: один конструктор по умолчанию, один конструктор, задающий свойство сообщения, и еще один, задающий свойства Message и InnerException. Четвертый конструктор служит для сериализации исключения. Новые классы исключений должны быть сериализуемыми.

```
[Serializable()]
public class InvalidDepartmentException : System.Exception
{
   public InvalidDepartmentException() : base() { }
   public InvalidDepartmentException(string message) : base(message) { }
   public InvalidDepartmentException(string message, System.Exception
inner) : base(message, inner) { }

   // A constructor is needed for serialization when an
   // exception propagates from a remoting server to the client.
   protected InvalidDepartmentException
            (System.Runtime.Serialization.SerializationInfo info,
     System.Runtime.Serialization.StreamingContext context) { }
}
```

Методы программирования

ЛЕКЦИЯ №6

2.3 Технология LINQ

Введение в LINQ. Актуальность и примеры

По мере становления платформы .NET Framework и поддерживаемых ею языков C# и VB, стало ясно, что одной из наиболее проблемных областей для разработчиков остается доступ к данным из разных источников.

Проблемы, связанные с базами данных, многочисленны. Первая сложность в том, что взаимодействие с базой данных происходит с помощью строковой команды на языке SQL, а не программно на уровне языка, на котором пишется программа. Это приводит к синтаксическим ошибкам, которые не проявляются вплоть до момента запуска. Неправильные ссылки на поля базы данных тоже не обнаруживаются.

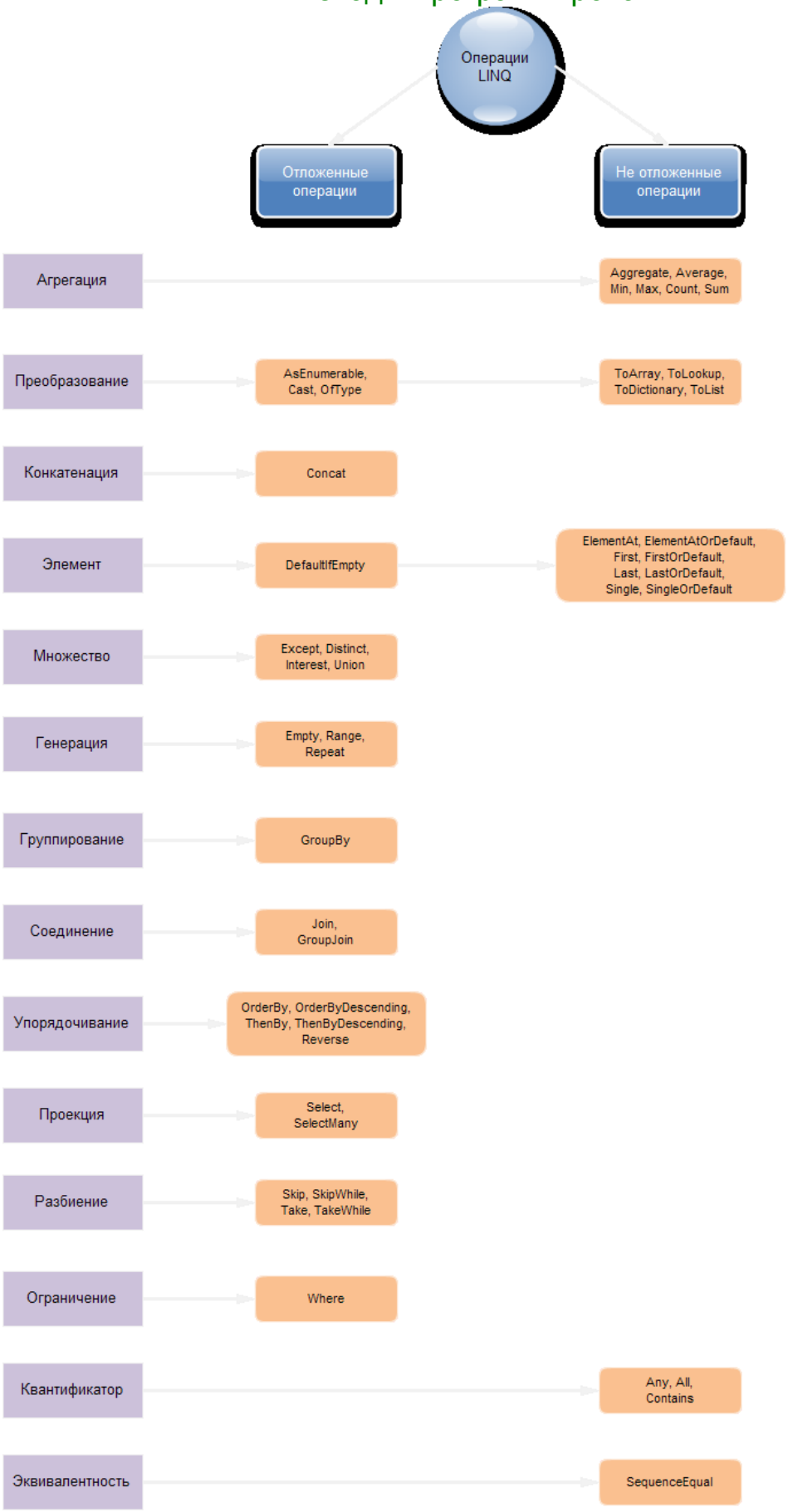
Вместо того чтобы просто добавить больше классов и методов для постепенного восполнения этих недостатков, в Microsoft решили разработать универсальную технологию LINQ (язык встроенных запросов). LINQ — технология Microsoft, предназначенная для поддержки запросов к данным всех типов на уровне языка. Запросы могут выполняться над массивами (в том числе строками) и коллекциями в памяти, базами данных, документами XML и другими данными.

LINQ включает в себя около 50 стандартных операций запросов, разделяемых на 2 большие группы - отложенные операции (выполняются не во время инициализации, а только при их вызове) и не отложенные операции (выполняются сразу).

К отложенным операциям относятся: AsEnumerable, Cast, OfType, Concat, DefaultEmpty, Except, Distinct, Intersect, Union, Empty, Range, Repeat, GroupBy, Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse, Select, SelectMany, Skip, SkipWhile, Take, TakeWhile, Where.

К не отложенным операциям относятся: Aggregate, Average, Min, Max, Count, Sum, ToArray, ToLookup, ToDictionary, ToList, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, Any, All, Contains, SequenceEqual.

На рисунке ниже наглядно показана "градация" операций LINQ:



Запросы LINQ возвращают набор объектов, единственный объект или подмножество полей из объекта или набора объектов. В LINQ этот возвращенный

набор называется последовательностью (sequence). Большинство последовательностей LINQ имеют тип IEnumerable<T>, где T — тип объектов, находящихся в последовательности. Например, если есть последовательность целых чисел, они должны храниться в переменной типа IEnumerable<int>.

Пример 1. Сортировка массива строк таким образом, чтобы числа, хранящиеся в строках, были упорядочены по возрастанию.

```
string[] numbers = { "40", "2012", "176", "5" };
// Преобразуем массив строк в массив типа int и сортируем по возрастанию,
// используя LINQ
int[] nums = numbers.Select(s => Int32.Parse(s)).OrderBy(s => s).ToArray();
foreach (int n in nums)
Console.Write(n + " ");
```

Вывод: 5, 40, 176, 2012

Если бы это были по-прежнему строки, то 5 оказалось бы в конце списка, а 176 — в начале.

Пример 2. Сортировка массива строк, таким образом, чтобы строки были упорядочены по возрастанию длины строки. В результате получим массив строк.

```
var nums = numbers.Select(s => s.Length()).OrderBy(s => s).ToArray();
foreach (var n in nums)
Console.Write(n + " ");
```

Вывод: 5, 40, 176, 2012

Пример 3. Сортировка массива строк, таким образом, чтобы строки были упорядочены по возрастанию длины строки. В результате получим последовательность IEnumerable<string>.

```
var nums = numbers.Select(s => s.Length()).OrderBy(s => s);
foreach (var n in nums)
Console.Write(n + " ");
```

Вывод: 5, 40, 176, 2012

В LINQ-запросах можно использовать SQL-подобный синтаксис и стандартную точечную нотацию C#.

При записи запроса в стандартной точечной нотации не происходит никакой трансформации при компиляции.

Пример 4. Получение названий марок автомобилей длиной менее 6 символов с использованием точечной нотации и синтаксиса запросов.

```
string[] names = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford",  "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",
    "Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
// Использование точечной нотации
IEnumerable<string> sequence = names
  .Where(n => n.Length < 6)
  .Select(n => n);

// Использование синтаксиса запросов
IEnumerable<string> sequence = from n in names
                               where n.Length < 6
                               select n;

foreach (string name in sequence)
{
  Console.WriteLine("{0}", name);
}
```

Синтаксис выражений запросов поддерживается только для наиболее распространенных операций запросов: Where, Select, SelectMany, Join, GroupJoin, GroupBy, OrderBy, ThenBy, OrderByDescending и ThenByDescending.

Интерфейс IEnumerable<T>

Функциональность LINQ to Objects (LINQ для работы с данными в памяти) обеспечивается интерфейсом IEnumerable<T>.

Если есть переменная типа IEnumerabie<T>, то можно сказать, что имеется последовательность элементов типа T. Например, IEnumerable<string> означает последовательность строк. Любая переменная, объявленная как IEnumerable<T> для типа T, рассматривается как последовательность типа T. Последовательность — это термин для обозначения коллекции, реализующей интерфейс IEnumerable<T>.

Большинство стандартных операций запросов представляют собой расширяющие методы в статическом классе System.Linq.Enumerable и прототипированы с IEnumerable<T> в качестве первого аргумента. Поскольку они являются расширяющими методами, предпочтительно вызывать их на переменной типа IEnumerable<T>, что позволяет синтаксис расширяющих методов, а не передавать переменную типа IEnumerable<T> в первом аргументе.

Пример 5. Имеется метод **Where**, который возвращает последовательность, удовлетворяющую условию. Например, необходимо получить последовательность из элементов массива, в которой останутся только отрицательные числа.

```
int [] m = {10, -5, -6, -2, 4, -4, 1, 0, -8, 12};
IEnumerable<int> m2 = m.Where(x=>x<0);
```

Метод Where вызывается на объекте m (массиве), однако не является методом массива, а представляет собой расширяющий метод – статический метод, который может вызываться на объекте, и описан в другом классе.

В качестве аргумента функции указана анонимная функция «X переходит в X<0», которая возвращает величину *true*, если очередной элемент последовательности меньше нуля и *false* – в противном случае и представляет собой более короткую запись функции

```
bool f(int x)
{
   return x<0;
}
```

или

```
bool f(int x)
{
  if (x<0)
    return true;
  else
    return false;
}
```

Важно помнить, что хотя многие из стандартных операций запросов прототипированы на возврат **IEnumerable**<T>, и **IEnumerable**<T> воспринимается как последовательность, на самом деле операции не возвращают последовательность в момент их вызова. То есть выполнение запроса откладывается до перечисления. Вместо этого операции возвращают объект, который не является последовательностью, но при перечислении выдает очередной элемент последовательности. Только во время перечисления возвращенного объекта (например, при выводе на экран или подсчете суммы) запрос выполняется, и выданный элемент помещается в выходную последовательность.

Для облегчения написания перечислителей в язык C# введено специальное ключевое слово **yield**.

Пример 6. Демонстрация места возникновения исключения при выходе за границы массива.

Вдобавок, поскольку такого рода запросы, возвращающие IEnumerable<T>, являются отложенными, код определения запроса может быть вызван однажды и затем использован многократно, с перечислением его результатов несколько раз. В случае изменения данных при каждом перечислении результатов будут выдаваться разные результаты. Ниже показан пример отложенного запроса, где результат не кэшируется и может изменяться от одного перечисления к другому:

Давайте более подробно рассмотрим, что здесь происходит. Когда вызывается операция Select, возвращается объект, хранящийся в переменной ints типа, реализующего интерфейс IEnumerable<int>. В этой точке запрос в действительности еще не выполняется, но хранится в объекте по имени ints. Другими словами, поскольку запрос еще не выполнен, последовательность целых чисел пока не существует, но этот объект ints знает, как получить последовательность, выполнив присвоенный ему запрос, которым в этом случае является операция Select.

Когда оператор foreach выполняется на ints в первый раз, объект ints производит запрос и получает последовательность по одному элементу за раз.

После этого в исходном массиве целых чисел изменяется один элемент. Затем снова запускается оператор foreach. Это заставляет ints снова выполнить запрос. Поскольку элемент в исходном массиве был изменен, а запрос выполнен снова, т.к. заново запущено перечисление ints, на этот раз возвращается измененный элемент.

Обратите внимание, что несмотря на однократный вызов запроса, результаты двух перечислений отличаются. Это еще одно доказательство того, что запрос является отложенным. Если бы это было не так, то результаты двух перечислений совпали бы.

Если не хотите, чтобы в таких ситуациях результаты отличались, воспользуйтесь одной из операций преобразования, которые не возвращают IEnumerable<T>, так что запрос получается не отложенным. К таким операциям относятся ToArray, ToList, ToDictionary или ToLookup и любые другие не отложенные операции.

Ниже показан тот же код, что и в предыдущем примере, но запрос возвращает не IEnumerable<T>, а List<int> — за счет вызова операции ToList:

```
// Создать массив целых чисел.
int[] intArray = new int[] { 1, 2, 3 };

List<int> ints = intArray.Select(i => i).ToList();

foreach (int i in ints)
Console.WriteLine(i);

// Изменить элемент, в источнике данных
intArray[0] = 5;

Console.WriteLine("---------");

foreach (int i in ints)
Console.WriteLine(i);
```

Вывод: 1, 2, 3 в обоих случаях.

Обратите внимание, что результаты, полученные от двух перечислений, одинаковы. Причина в том, что метод ToList не является отложенным, и запрос на самом деле выполняется в тот момент, когда он был вызван.

Отличие между этим примером и предыдущим, где операция Select отложена, заключается в том, что операция ToList является не отложенной. Когда ToList вызывается в операторе запроса, она немедленно перечисляет объект, возвращенный оператором Select, в результате чего весь запрос перестает быть отложенным.

Операции LINQ

Операция Where

Операция Where используется для фильтрации элементов в последовательность. Операция Where имеет два прототипа.

Первый прототип Where

```
public static IEnumerable<T> Where<T>(
this IEnumerable<T> source,
Func<T, bool> predicate);
```

Этот прототип Where принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает true.

Благодаря расширяющим методам нет необходимости передавать первый аргумент в стандартную операцию запроса, первый аргумент которой помечен модификатором — ключевым словом this, при условии, что операция вызывается на объекте того же типа, что у первого аргумента.

При вызове Where передается делегат метода-предиката. Этот метод-предикат должен принимать тип *T* в качестве входного, где *T* — тип элементов, содержащихся во входной последовательности, и возвращать bool. Операция Where вызовет метод-предикат для каждого элемента входной последовательности и передаст ему этот элемент. Если метод-предикат вернет true, то Where выдаст этот элемент в выходную последовательность Where. Если метод-предикат вернет false, то Where этого не сделает.

Второй прототип Where:

```
public static IEnumerable<T> Where<T>(
this IEnumerable<T> source,
Func<T, int, bool> predicate);
```

Второй прототип Where идентичен первому, но с тем отличием, что он указывает на то, что делегат метода-предиката принимает дополнительный целочисленный аргумент. Этот аргумент будет индексом элемента во входной последовательности.

IEnumerable<string> *sequence* = *cars*.Where((*p*, *i*) => (*i* & 1) == 1);

foreach (string *s* in *sequence*)
Console.WriteLine(*s*);

Операция Select

Операция Select используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа. Эти типы не обязательно должны совпадать.

Существуют два прототипа этой операции, которые описаны ниже:
Первый прототип Select
public static IEnumerable<*S*> Select<T, S>(
this IEnumerable<T> source,
Func<*T*, *S*> *selector*);

Этот прототип Select принимает входную последовательность и делегат метода-селектора в качестве входных параметров, а возвращает объект, который при перечислении проходит по входной последовательности и выдает последовательность элементов типа S. Как упоминалось ранее, T и S могут быть как одного, так и разных типов.

При вызове Select делегат метода-селектора передается в аргументе selector. Метод-селектор должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа S. Операция Select вызовет метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Метод-селектор выберет интересующую часть входного элемента, создаст новый элемент — возможно, другого типа (даже анонимного) — и вернет его.

Второй прототип Select
public static IEnumerable<*S*> Select<*T*, *S*>(
this IEnumerable<*T*> *source*,
Func<*T*, int, *S*> *selector*);

В этом прототипе операции Select методу-селектору передается дополнительный целочисленный параметр. Это индекс, начинающийся с нуля, входного элемента во входной последовательности.

Пример вызова первого прототипа показан ниже:

string[] cars = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge", "BMW", "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

IEnumerable<int> *sequence* = *cars*.Select(p => p.Length);

```
foreach (int i in sequence)
Console.Write(i + " ");
```

Обратите внимание, что метод-селектор передается через лямбда-выражение. В данном случае лямбда-выражение вернет длину каждого элемента из входной последовательности. Также отметьте, что, хотя тип входных элементов — строка, тип выходных элементов — int.

Это простой пример, потому что никакого класса не генерируется. Ниже приведена более интересная демонстрация использования первого прототипа:

```
var sequence = cars.Select(p => new { p, p.Length });

foreach (var i in sequence)
Console.WriteLine(i);
```

Обратите внимание, что лямбда-выражение создает экземпляр нового анонимного типа. Компилятор динамически сгенерирует анонимный тип, который будет содержать string *p* и int *p*.Length, и метод-селектор вернет этот вновь созданный объект. Поскольку тип возвращаемого элемента является анонимным, выходная последовательность присваивается переменной, указанной с помощью ключевого слова var.

С этим кодом связана одна проблема: управлять именами членов динамически сгенерированного анонимного класса нельзя. Однако, благодаря средству инициализации объектов C#, можно написать лямбда-выражение и задать имена членов анонимного класса, как показано ниже:

```
var carObj = cars.Select(p => new { LastName = p, Length = p.Length });

foreach (var i in carObj)
Console.WriteLine("Автомобиль {0} имеет длину {1} символов", i.LastName, i.Length);
```

Для примера второго прототипа будет добавлен индекс, который передается методу-селектору, в тип элемента выходной последовательности:

```
var carObj = cars.Select((p, i) => new { Index = i + 1, LastName = p });

foreach (var i in carObj)
Console.WriteLine( i.Index + ". " + i.LastName);
```

Этот пример выводит номер индекса плюс единица, за которым следует имя. Код производит следующий результат:

Операция Take возвращает указанное количество элементов из входной последовательности, начиная с ее начала.

Ниже показан пример использования операции *Take*:

string[] *cars* = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge", "BMW", "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

IEnumerable<string> *auto* = *cars*.Take(5);

foreach (string *str* in *auto*)

Console.WriteLine(*str*);

Этот код вернет первые пять элементов из массива cars.

Операция TakeWhile

Операция TakeWhile возвращает элементы из входной последовательности, пока истинно некоторое условие, начиная с начала последовательности. Остальные входные элементы пропускаются.

Ниже приведен пример вызова первого прототипа:

string[] cars = { "Nissan", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge", "BMW", "Aston Martin", "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

IEnumerable<string> auto = cars.TakeWhile(s => s.Length < 12);

foreach (string str in auto)

Console.WriteLine(str);

В приведенном коде входные элементы извлекаются до тех пор, пока их длина не превышает 11 символов.

Элемент, который заставил операцию TakeWhile прекратить обработку входной последовательности — Aston Martin.

Рассмотрим пример второго прототипа операции TakeWhile:

IEnumerable<string> auto = cars.TakeWhile((s, i) => s.Length < 12 && i < 5);

foreach (string str in auto)

Console.WriteLine(str);

Код в этом примере прекращает выполнение, когда входной элемент превысит 11 символов в длину или когда будет достигнут шестой элемент — в зависимости от того, что произойдет раньше.

Операция Skip

Операция Skip пропускает указанное количество элементов из входной последовательности, начиная с ее начала, и выводит остальные.

Операция Skip получает входную последовательность и целое число count, задающее количество входных элементов, которое должно быть пропущено, и возвращает объект, который при перечислении пропускает первые count элементов и выводит все последующие элементы.

Ниже приведен пример вызова операции Skip:

string[] cars = { "Nissan", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge", "BMW", "Aston Martin", "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

```
IEnumerable<string> auto = cars.Skip(5);
foreach (string str in auto)
    Console.WriteLine(str);
```

В данном примере пропускаются первые 5 элементов. Обратите внимание, что в следующем выводе действительно пропущены первые пять элементов входной последовательности:

Операция SkipWhile

Операция SkipWhile обрабатывает входную последовательность, пропуская элементы до тех пор, пока условие истинно, а затем выводит остальные в выходную последовательность. У операции SkipWhile есть два прототипа, описанные ниже:

```
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Операция SkipWhile принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении пропускает элементы до тех пор, пока метод-предикат возвращает true. Как только метод-предикат вернет false, операция SkipWhile начинает вывод всех прочих элементов. Метод-предикат принимает элементы входной последовательности по одному и возвращает признак того, должен ли элемент быть пропущен из входной последовательности.

Этот прототип подобен первому во всем, за исключением дополнительного параметра — индекса элемента из входной последовательности, начинающегося с нуля.

Ниже приведен пример вызова первого прототипа операции SkipWhile:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};
IEnumerable<string> auto = cars.SkipWhile(s => s.StartsWith("A"));
foreach (string str in auto)
    Console.WriteLine(str);
```

В этом примере метод SkipWhile должен пропускать элементы до тех пор, пока они начинаются с буквы "A". Все остальные элементы выдаются в выходную последовательность.

Теперь рассмотрим пример использования второго прототипа SkipWhile:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};
IEnumerable<string> auto = cars.SkipWhile((s, i) => s.StartsWith("A") && i < 10);

foreach (string str in auto)
    Console.WriteLine(str);
```

В данном примере входные элементы пропускаются до тех пор, пока они начинаются с буквы "A" или пока не будет достигнут десятый элемент. Остальные элементы выдаются в выходную последовательность.

Пропуск элементов был прекращен, как только встретился элемент Nissan, поскольку он начинается с N, хотя его индексом является 3.

Concat

Операция Concat соединяет две входные последовательности и выдает одну выходную последовательность.

В этом прототипе две последовательности одного типа T — first и second — являются входными. Возвращается объект, который при перечислении проходит по первой последовательности, выдавая каждый ее элемент в выходную последовательности за которым начинается перечисление второй входной последовательности с выдачей каждого ее элемента в ту же выходную последовательность.

Ниже приведен пример использования операции Concat, а также операций Take и Skip:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};
IEnumerable<string> auto = cars.Take(5).Concat(cars.Skip(5));
foreach (string str in auto)
Console.WriteLine(str);
```

Этот код берет пять первых членов из входной последовательности cars и соединяет со всеми, кроме первых пяти входных элементов из последовательности cars.

Альтернативный подход к соединению предусматривает вызов операции SelectMany на массиве последовательностей, как показано ниже:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};
IEnumerable<string> auto = new[] {
cars.Take(5),
cars.Skip(5)}
.SelectMany(s => s);
```

В данном примере создается экземпляр массива, состоящего из двух последовательностей: одной, созданной вызовом операции Take на входной последовательности, и другой, созданной вызовом операции Skip на входной последовательности. Обратите внимание, что это подобно предыдущему примеру во всем, за исключением того, что на массиве последовательностей вызывается операция SelectMany. С учетом того, что операция Concat позволяет объединять только две последовательности, при наличии массива последовательностей продемонстрированный прием может оказаться удобнее.

Результат получается тем же, что и при использовании операции Concat.

OrderBy и OrderByDescending

Операции упорядочивания позволяют выстраивать входные последовательности в определенном порядке. Важно отметить, что и OrderBy, и OrderByDescending требуют входной последовательности типа IEnumerable<T> и возвращают последовательность типа IOrderedEnumerable<T>. Передавать операциям OrderBy и OrderByDescending в качестве входной последовательности IOrderedEnumerable<T> нельзя. Причина в том, что последующие вызовы операций OrderBy и OrderByDescending не принимают во внимание порядок, созданный предыдущими вызовами OrderBy и OrderByDescending. Это значит, что передавать последовательность, возвращенную из OrderBy либо OrderByDescending, в последующий вызов операции OrderBy или OrderByDescending не имеет смысла.

Если требуется большая степень упорядочивания, чем возможно достичь с помощью одиночного вызова операции OrderBy или OrderByDescending, необходимо последовательно вызывать операции ThenBy или ThenByDescending, речь о которых пойдет далее.

OrderBy

Операция OrderBy позволяет упорядочить входную последовательность на основе метода keySelector, который возвращает значение ключа для каждого входного элемента. Упорядоченная выходная последовательность IOrderedEnumerable<T> выдается в порядке возрастания на основе значений возвращенных ключей.

Сортировка, выполненная операцией OrderBy, определена как неустойчивая. Это значит, что она не сохраняет входной порядок элементов. Если два входных элемента поступают в операцию OrderBy в определенном порядке, и значения ключей этих двух элементов совпадают, их расположение в выходной последовательности может остаться прежним или поменяться, причем ни то, ни другое не гарантируется. Даже если все выглядит нормально, поскольку порядок определен как неустойчивый, всегда следует исходить из этого. Это значит, что никогда нельзя полагаться на порядок элементов, поступающих из операций OrderBy или OrderByDescending, для любого поля кроме указанного в вызове метода. Сохранение любого порядка, который существует в последовательности, передаваемой любой из этих операций, не может гарантироваться.

Операции ThenBy и ThenByDescending

Вызовы ThenBy и ThenByDescending могут соединяться в цепочку, т.к. они принимают в качестве входной последовательности IOrderedEnumerable<T> и возвращают в качестве выходной последовательности тоже IOrderedEnumerable<T>.

Например, следующая последовательность вызовов не разрешена:

inputSequence.OrderBy(s => s.LastName).OrderBy(s => s.FirstName)...

Вместо нее должна использоваться такая цепочка:

inputSequence.OrderBy(s => s.LastName).ThenBy(s => s.FirstName).

Операция ThenBy позволяет упорядочивать входную последовательность типа IOrderedEnumerable<T> на основе метода keySelector, который возвращает значение ключа. В результате выдается упорядоченная последовательность типа IOrderedEnumerable<T>.

В отличие от большинства операций отложенных запросов LINQ to Objects, операции ThenBy и ThenByDescending принимают другой тип входных последовательностей — IOrderedEnumerable<T>. Это значит, что сначала должна быть вызвана операция OrderBy или OrderByDescending для создания последовательности IOrderedEnumerable, на которой можно затем вызывать операции ThenBy и ThenByDescending.

Сортировка, выполняемая операцией ThenBy, является устойчивой. Другими словами, она сохраняет входной порядок элементов с эквивалентными ключами. Если два входных элемента поступили в операцию ThenBy в определенном порядке, и ключевое значение обоих элементов одинаково, то порядок тех же выходных элементов гарантированно сохранится. В отличие от OrderBy и OrderByDescending, операции ThenBy и ThenByDescending выполняют устойчивую сортировку.

Ниже показан пример использования первого прототипа:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

IEnumerable<string> auto = cars.OrderBy(s => s.Length).ThenBy(s => s);
foreach (string str in auto)
    Console.WriteLine(str);
```

Этот код сначала упорядочивает элементы по их длине, в данном случае — длине названия автомобиля. Затем упорядочивает по самому элементу. В результате получается список названий, отсортированный по длине от меньшей к большей (по возрастанию), а затем — по имени в алфавитном порядке:

Операции ToArray и ToList

Следующие операции преобразования предоставляют простой и удобный способ преобразования последовательностей в другие типы коллекций.

Операция ToArray создает массив типа T из входной последовательности типа T. Эта операция имеет один прототип, описанный ниже:

```
public static T[] ToArray<T>( this IEnumerable<T> source);
```

Эта операция берет входную последовательность source с элементами типа T и возвращает массив элементов типа T.

Операция ToList создает List типа T из входной последовательности типа T. Эта операция имеет один прототип, описанный ниже:

```
public static List<T> ToList<T>(
    this IEnumerable<T> source);
```

Эта операция часто полезна для кэширования последовательности, чтобы она не могла измениться перед ее перечислением. Также, поскольку эта операция

не является отложенной и выполняется немедленно, множество перечислений на созданном списке List<T> всегда видят одинаковые данные.

Операции Any, All и Contains

Операция Any возвращает true, если любой из элементов входной последовательности отвечает условию.

Операция All возвращает true, если каждый элемент входной последовательности отвечает условию.

```
all = cars.All(s => s.Length > 2);
Console.WriteLine("Правда ли, что все элементы коллекции Cars длинее 2х символов: " + all);
```

Операция Contains возвращает true, если любой элемент входной последовательности соответствует указанному значению.

Для демонстрации работы первого прототипа, рассмотрим следующий пример:

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW", "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};
Console.WriteLine("Операция Contains\n\n********\n");
bool contains = cars.Contains("Jaguar");
Console.WriteLine("Наличие \"Jaguar\" в массиве: " + contains);
contains = cars.Contains("BMW");
Console.WriteLine("Наличие \"BMW\" в массиве: " + contains);
```

В данном пример проверяется наличие слов "Jaguar" и "BMW" в исходном массиве Cars с помощью первого прототипа операции Cars.

Операции Count, LongCount и Sum

Операция Count возвращает количество элементов во входной последовательности.

Операция LongCount возвращает количество элементов входной последовательности как значение типа long.

Операция Sum возвращает сумму числовых значений, содержащихся в элементах последовательности. Эта операция имеет два прототипа, описанные ниже:

```
long optionsSum = options.Sum(o => o.optionsCount);
Console.WriteLine("Сумма опционов сотрудников: {0}", optionsSum);
```

Операции Min и Max

Операция Min возвращает минимальное значение входной последовательности. Эта операция имеет четыре прототипа, которые описаны ниже:

Первый прототип операции Min возвращает элемент с минимальным числовым значением из входной последовательности source.

Второй прототип операции Min ведет себя подобно первому, за исключением того, что он предназначен для нечисловых типов.

```
public static T Min<T>(
    this IEnumerable<T> source);
```

Операция Max возвращает максимальное значение из входной последовательности.

Операции Average и Aggregate

Операция Average возвращает среднее арифметическое числовых значений элементов входной последовательности.

Операция Aggregate выполняет указанную пользователем функцию на каждом элементе входной последовательности, передавая значение, возвращенное этой функцией для предыдущего элемента, и возвращая ее значение для последнего элемента.

```
int N = 5;
int agg = Enumerable
        .Range(1, N)
        .Aggregate((av, e) => av * e);
Console.WriteLine("{0}! = {1}",N,agg);
```

В этом коде генерируется последовательность, содержащая целые числа от 1 до 5, для чего используется операция Range. Затем вызывается операция Aggregate, которой передается лямбда-выражение, умножающее переданное агрегатное значение на сам переданный элемент.

Для деления вещественных чисел 12/6/2/1:

```
List<float> m = new List<float>(){6,2,1};
var agg = m.Aggregate(12.0, (av, ee) => av / ee);
```

обратите внимание, что 12.0 – для того, чтобы Аккумулятор был типа float..

Методы программирования

ЛЕКЦИЯ №7

2.4 Делегаты и события

Делегаты

Делегат — это частный случай класса для определения функционального объекта. В определение делегата входит синтаксис функций, который может быть представлен этим делегатом. Делегаты, как и интерфейсы не задают реализации. Экземплярами класса делегата являются методы, в которых задана реализация. Каждый метод, сигнатура которого совпадает с сигнатурой делегата, может рассматриваться как экземпляр этого делегата.

При объявлении делегата указывается прототип функции, тип которой совместим с типом делегата. Экземпляры делегата можно использовать для хранения функции с указанным прототипом и вызова этой функции.

Объявление делегата может располагаться внутри или вне какого-либо класса и похоже на объявление функции, но с добавлением в начале ключевого слова delegate. Для объявления делегата используется следующий синтаксис:

delegate <тип_делегата> <имя_делегата> (<аргументы>);

Возможность объявления делегата вне какого-либо класса связана с тем, что на основе строки объявления делегата компилятор фактические генерирует объявление класса, производного от MulticastDelegate.

Для создания экземпляра делегата необходимо объявить переменную типа делегата и с помощью ключевого слова new вызвать конструктор делегата, передав ему в качестве параметра метод, который будет храниться в делегате или ранее созданный экземпляр делегата.

<имя_делегата> <имя_переменной>=**new** <имя_делегата>(<имя_метода> или <имя_экземпляра_делегата>);

Для вызова делегата указывается имя переменной типа делегата, а затем в круглых скобках через запятую перечисляются параметры, которые надо передать методу, вызываемому через экземпляр делегата.

Если объявлен делегат:

delegate double Calc(double x);

то методы

double Math.Sin(double x);

double Math.Cos(double x);

double Kvadrat.Pl(double dlina);

являются экземплярами этого делегата (это связано только с тем, что у них одинаковый прототип).

Можно создать экземпляры делегата **Calc** различными способами (через метода класса, через метод объекта, через существующий делегат) и вызвать соответствующий метод через делегат:

```
Calc c1 = new Calc(Math.Sin);   //через метода класса
Calc c2 = new Calc(Kvadrat.Pl);    //через метода класса
Kvadrat k = new Kvadrat();
Calc c3 = new Calc(k.Pl);    //через метода объекта
```

Calc c4 = **new** Calc(c3); //через существующий делегат

Теперь при вызове делегата c1(2) — будет вычислен Sin(2); при вызове c2(3), c3(3) и c4(3) — будет вызван метод Pl(3) класса Kvadrat.

Обычно делегаты используются для реализации обратных вызовов. Обратный вызов - вызов из функции A функции B, переданной в качестве параметра функции A. Обратные вызовы часто применяются для передачи из функции информации о производимых ее действиях.

Синтаксис объявления делегата:

<модификаторы_доступа> **delegate** <тип_результата> <имя_делегата> (<спецификация_параметров>);

Модификатор является не обязательным и может принимать значения: **new**, **public**, **protected**, **internal**, **private**.

delegate – служебное слово, вводящее тип делегата;

<тип_результата> – обозначение типа значения, возвращаемого методами, которые будет представлять делегат;

<имя_делегата> – выбранное программистом имя типа делегата;

<спецификация_параметров> – список спецификаций параметров тех методов, которые будет представлять делегат.

В объявлении делегата-типа нет необходимости определять его конструктор. Конструктор встраивается автоматически.

Делегат может быть объявлен:

— как локальный тип класса или структуры (локализация делегата);

— в пространстве имен наряду с объявлениями других классов и структур (внешнее размещение делегата).

На основе делегата можно создать экземпляр делегата и присвоить ему ссылку на функцию, сигнатура которой совпадает с сигнатурой делегата.

Так как делегат — это тип ссылок, то, определив делегат, можно применять его для создания переменных ссылок с помощью операции **new**.

Синтаксис определения ссылок-делегатов традиционен:

<имя_делегата> <имя_ссылки>;

Синтаксис создания экземпляра делегата:

<имя_ссылки> = **new** <имя_делегата> (<имя_метода> или <имя_экземпляра_делегата>);

При создании экземпляра делегата при обращении к конструктору в операции **new** в качестве аргумента можно использовать:

— метод класса (имя метода, уточненное именем класса);

— метод объекта (имя метода, уточненное именем объекта);

— ссылку на уже существующий в программе экземпляр делегата.

Экземпляр делегата моет представлять как метод класса, так и метод объекта. Однако в обоих случаях метод должен соответствовать по типу возвращаемого значения и по спецификации параметров объявлению делегата.

Ковариантность и контравариантность

Делегаты становятся еще более гибкими средствами программирования благодаря двум свойствам: ковариантности и контравариантности. Как правило, метод, передаваемый делегату, должен иметь такой же возвращаемый тип и сигнатуру, как и делегат. Но в отношении производных типов это правило оказывается не таким строгим. В частности, ковариантность позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. А контравариантность позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

Функции высших порядков

Одно из наиболее важных применений делегатов связано с функциями высших порядков. Функцией высшего порядка называется метод класса, у которой один или несколько аргументов принадлежат к функциональному типу. Без этих функций в программировании обойтись довольно трудно. Классическим примером является функция вычисления интеграла, у которой один из аргументов задает подынтегральную функцию. Другим примером может служить функция, сортирующая объекты. Аргументом ее является функция Compare, сравнивающая два объекта. В зависимости от того, какая функция сравнения будет передана на вход функции сортировки, объекты будут сортироваться по-разному, например, по имени, или по ключу, или по нескольким полям. Вариантов может быть много, и они определяются классом, описывающим сортируемые объекты.

Пример. Вычислить определенный интеграл $\int_a^b f(x)dx$. Функция $f(x)$ может быть произвольной, поэтому ее будем передавать в метод вычисления определенного интеграла.

```
double Integral(double a, double b, delegate double f (double))
{
  return (f (a)+f (b))*(b-a) /2;
}
double f (double x)
{
  return Math.Sin(x)*x;
}
```

Массивы делегатов

Ссылки на экземпляры делегатов, можно объединять в массивы. Такая возможность позволяет программисту задавать наборы действий, которые затем будут автоматически выполнены в указанной последовательности.

Многоадресные групповые экземпляры делегатов

До сих пор рассматривались делегаты, каждый экземпляр которых представляет один конкретный метод. Однако делегат может содержать ссылки сразу на несколько методов, соответствующих типу делегата. При однократном обращении к такому делегату автоматически организуется последовательный вызов всех методов, которые он представляет.

К многоадресному делегату можно присоединить или отсоединить делегат с помощью операций += и -= соответственно.

Многоадресный_делегат += ссылка_на_делегат;

Многоадресный_делегат -= ссылка_на_делегат;

Анонимные методы

К экземпляру делегата можно прикрепить явно заданный код безымянного метода. Такой метод называют анонимным. Сигнатуру этого метода (спецификацию параметров и тип возвращаемого значения) специфицирует тип делегата. Применение анонимных методов рекомендуется в тех случаях, когда метод используется однократно.

События

Все объекты обладают одними и теми же методами и, следовательно, ведут себя одинаково, то есть методы задают врожденное поведение объектов. Одним из способов сделать поведение отличным для данного объекта — это наследование. Можно создать класс-наследник, у которого, наряду с унаследованным родительским поведением, будут и собственные методы.

Еще один механизм, позволяющий объектам одного класса вести себя по разному — события.

О событиях в мире программных объектов чаще всего говорят в связи с интерфейсными объектами, у которых события возникают по причине действий пользователя. Так, командная кнопка может быть нажата — событие Click, документ может быть закрыт — событие Close, в список может быть добавлен новый элемент — событие Changed. Интерфейсные и многие другие программные объекты обладают стандартным набором предопределенных событий.

Можно создавать события для самостоятельно разработанных классов.

Событие – средство, позволяющее объекту (или классу) послать сообщение о переходе в некоторое новое состояние или о получении сообщения из внешнего источника.

В основе механизма обработки событий лежат делегаты, поэтому прежде, чем объявить событие, необходимо объявить делегат. Объявление события похоже на объявление переменной типа делегата, но с добавлением в начале ключевого слова **event**. Для объявления события используется следующий синтаксис:

event <делегат-тип> <имя_события>;

Чтобы сгенерировать (возбудить) событие, указывается имя события, а затем в круглых скобках через запятую перечисляются параметры события. Прежде чем возбуждать событие, рекомендуется проверить, переменная, описывающая событие, не равна **null**.

Для подписки на событие для какого-либо объекта используется оператор +=, после которого с помощью ключевого слова **new** вызывается конструктор делегата, в качестве параметра которого указывается имя метода, представляющего из себя обработчик события.

объект.событие+=new делегат(метод);

Класс, решивший иметь события, должен уметь, по крайней мере:

— объявить событие в классе;

— зажечь в нужный момент событие, передав обработчику необходимые для обработки аргументы. (Под зажиганием или включением события понимается некоторый механизм, позволяющий объекту уведомить клиентов класса, что у него произошло событие.).

Заметьте, что, зажигая событие, класс посылает сообщение получателям события — объектам некоторых других классов. Будем называть класс, зажигающий событие, классом — отправителем сообщения (sender). Класс, чьи объекты получают сообщения, будем называть классом - получателем сообщения (receiver). Класс-отправитель сообщения, в принципе, не знает своих получателей. Он отправляет сообщение в межмодульное пространство. Одно и то же сообщение может быть получено и по-разному обработано произвольным числом объектов разных классов.

Класс sender. Как зажигаются события

Объявление события может размещаться в классе или в интерфейсе.

Если делегат определен, то в классе Sender, создающем события, достаточно объявить событие как экземпляр соответствующего делегата. Это делается точно так же, как и при объявлении функциональных экземпляров делегата. Исключением является добавление служебного слова **event**. Формальный синтаксис объявления:

<модификаторы> event <имя_делегата> <имя_события>;

модификатором может быть abstract, new, override, static, virtual, public, protected, private, internal;

event – служебное слово декларации события;

<имя_события> – идентификатор, выбираемый программистом в качестве названия конкретного члена, называемой переменной события. Принято начинать с префикса on_.

<имя_делегата> – имя делегата-типа. Он должен представлять событию те методы, которые будут вызываться в ответ на посылку сообщения о событии.

Таким образом, событие – член класса, имеющий тип делегата.

Причины возникновения события могут быть разными. Поэтому вполне вероятно, что одно и то же событие будет зажигаться в разных методах класса в

тот момент, когда возникнет одна из причин появления события. Поскольку действия по зажиганию могут повторяться, полезно в состав методов класса добавить защищенную процедуру, зажигающую событие. Этой процедуре обычно дается имя, начинающееся со слова On, после которого следует имя события. Процедура состоит из вызова объявленного события, если есть ли хоть один обработчик события, способный принять соответствующее сообщение. Если таковых нет, то событие не включается.

Посылка сообщения оформляется как оператор в теле некоторого метода. Оператор посылки выглядит так:

Имя_события(аргументы_для_делегата).

Событие, по существу, представляет собой автоматическое уведомление о том, что произошло некоторое действие.

Классы receiver. Как обрабатываются события

Объекты класса Sender создают события и уведомляют о них объекты, возможно, разных классов, названных нами классами Receiver, или клиентами. Класс receiver должен:

— иметь обработчик события - процедуру, согласованную по сигнатуре с функциональным типом делегата, который задает событие;

— иметь ссылку на объект, создающий событие, чтобы получить доступ к этому событию - event-объекту;

— уметь присоединить обработчик события к event-объекту. Это можно реализовать по-разному, но технологично это делать непосредственно в конструкторе класса, так что когда создается объект, получающий сообщение, он изначально готов принимать и обрабатывать сообщения о событиях.

События действуют по следующему принципу:

— объект, проявляющий интерес к событию создает экземпляр того делегата, на который настроено событие;

— регистрирует обработчик этого события (подключает экземпляр делегата к событию);

— когда событие происходит, вызываются все зарегистрированные обработчики этого события объектом, в котором произошло событие;

На одно событие могут быть подписаны несколько методов, для каждого из которых нужно использовать свой оператор приведенного вида.

Несмотря на всю свою простоту, данный пример кода содержит все основные элементы, необходимые для обработки событий. Он начинается с объявления типа делегата для обработчика событий, как показано ниже.

delegate void MyEventHandler();

Все события активизируются с помощью делегатов. Поэтому тип делегата события определяет возвращаемый тип и сигнатуру для события. В данном случае параметры события отсутствуют, но их разрешается указывать.

Далее создается класс события MyEvent. В этом классе объявляется событие SomeEvent в следующей строке кода.

public event MyEventHandler on_SomeEvent;

Обратите внимание на синтаксис этого объявления. Ключевое слово event уведомляет компилятор о том, что объявляется событие.

Кроме того, в классе MyEvent объявляется метод OnSomeEvent (), вызываемый для сигнализации о запуске события. Это означает, что он вызывается, когда происходит событие. В методе OnSomeEvent () вызывается обработчик событий с помощью делегата on_SomeEvent.

if (on_SomeEvent != **null**)

on_SomeEvent();

Как видите, обработчик вызывается лишь в том случае, если событие SomeEvent не является пустым. А поскольку интерес к событию должен быть зарегистрирован в других частях программы, чтобы получать уведомления о нем, то метод OnSomeEvent() может быть вызван до регистрации любого обработчика события. Но во избежание вызова по пустой ссылке делегат события должен быть проверен, чтобы убедиться в том, что он не является пустым.

В классе EventDemo создается обработчик событий Handler( ). В данном примере обработчик событий просто выводит сообщение, но другие обработчики могут выполнять более содержательные функции. Далее в методе Main () создается объект класса события MyEvent, а Handler () регистрируется как обработчик этого события, добавляемый в список.

MyEvent evt = new MyEvent();

// Добавить метод Handler() в список событий,

evt.SomeEvent += Handler;

Обратите внимание на то, что обработчик добавляется в список с помощью оператора +=. События поддерживают только операторы += и –=. В данном случае метод Handler() является статическим, но в качестве обработчиков событий могут также служить методы экземпляра.

И наконец, событие запускается, как показано ниже.

evt.OnSomeEvent();

Вызов метода OnSomeEvent() приводит к вызову всех событий, зарегистрированных обработчиком. В данном случае зарегистрирован только один такой обработчик, но их может быть больше, как поясняется в следующем разделе.

Пример 1. Класс Krug имеет свойство Радиус и генерирует событие при увеличении радиуса. Событие передает ссылку на круг, в котором произошло событие увеличения радиуса и старый радиус.

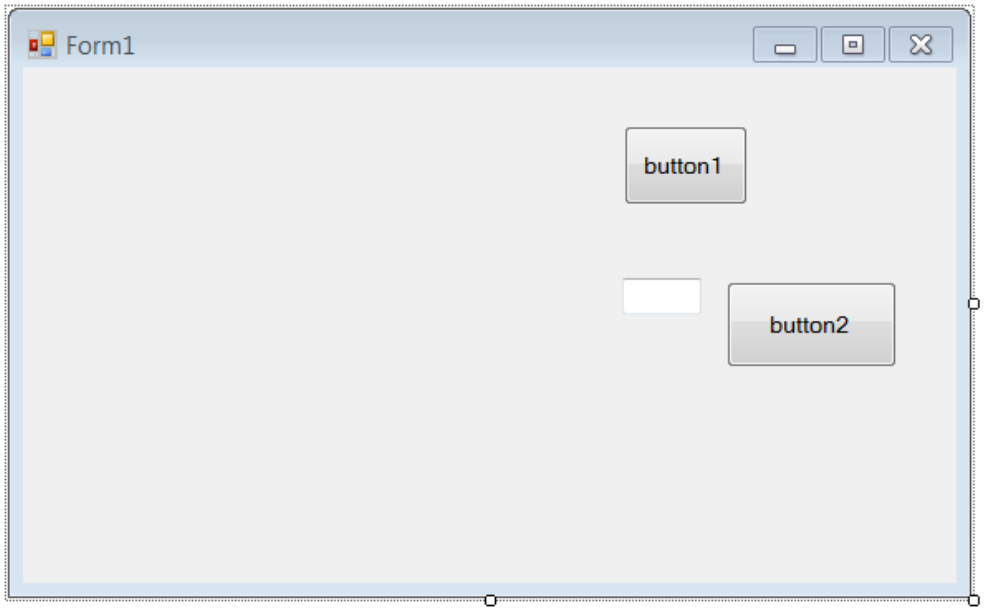
```
using System;

namespace WindowsFormsApplication2
{
   public delegate void delKrugR(Krug krug,  int oldR);

   public class Krug
```

```
    {
        public event delKrugR on_RUv;
        private int r;
        public int R
        {
            get { return r; }
            set {
                int oldR = r;
                r = value;
                if (r > oldR)
                    OnRUv(ref oldR); }
        }
        void OnRUv(ref int oldR)
        {
            if (on_RUv != null)
                on_RUv(this, oldR);
        }
    }
}
```

В форме имеется возможность изменить радиус круга двумя способами



Если происходит событие увеличения радиуса, то необходимо вывести информацию в заголовок формы.

```
using System;
using System.Windows.Forms;

namespace WindowsFormsApplication2
{
```

```
public partial class Form1 : Form
{
public Krug krug;
public Form1()
{
InitializeComponent();
krug = new Krug();
krug.on_RUv += UvelKrug;
}

void UvelKrug(Krug krug, int oldR)
{
Text = "Радиус круга увеличился: Старый радиус=" + oldR +
", Новый=" + krug.R;
}

private void button1_Click(object sender, EventArgs e)
{
krug.R += 5;
}

private void button2_Click(object sender, EventArgs e)
{
krug.R = int.Parse(textBox1.Text);
}
}
}
```

ЛЕКЦИЯ №8

2.5 Технология COM

Общие сведения о технологии COM

COM (Component Object Model) — это объектная модель компонентов. Данная технология является базовой для технологий ActiveX и OLE. Технологии OLE и ActiveX — всего лишь надстройки над данной технологией.

Технология COM предоставляет модель взаимодействия между компонентами и приложениями, написанными на разных языках и средах программирования, поскольку технология COM определяет двоичный стандарт.

Технология COM работает с COM-объектами, которые содержат свойства, методы и интерфейсы. Обычный COM-объект включает в себя один или несколько интерфейсов. Каждый из этих интерфейсов имеет собственный указатель. Ключевым аспектом технологии COM является возможность предоставления связи и взаимодействия между компонентами и приложениями, а также реализация клиент-серверных взаимодействий при помощи интерфейсов.

Технология COM реализуется с помощью COM-библиотек, которые содержат набор стандартных интерфейсов, которые обеспечивают функциональность COM-объекта, а также небольшой набор функций API, отвечающих за создание и управление COM-объектов.

Технология COM имеет два явных преимущества:

создание COM-объектов не зависит от языка программирования. Таким образом, COM-объекты могут быть написаны на различных языках;

COM-объекты могут быть использованы в любой среде программирования под Windows. В число этих сред входят Delphi, Visual C++, C++Builder, Visual Basic, C# и многие другие.

Хотя технология COM обладает явными плюсами, она имеет также и минусы, среди которых зависимость от платформы. То есть, данная технология применима только в операционной системе Windows и на платформе Intel.

Все COM-объекты обычно содержатся в файлах с расширением DLL или OCX. Один такой файл может содержать как одиночный COM-объект, так и несколько COM-объектов.

Развитие COM-технологий

Одной из важнейших задач, которые ставила перед собой фирма Microsoft, когда продвигала операционную систему Windows, была задача по обеспечению эффективного взаимодействия между различными программами, работающими в Windows.

Самыми первыми попытками решить эту непростую задачу были буфер обмена, разделяемые файлы и технология динамического обмена данными (Dynamic Data Exchange, DDE). После этого была разработана технология связывания и внедрения объектов (Object Linking and Embedding, OLE). Первоначальная версия OLE 1 предназначалась для создания составных документов. Эта версия была признана несовершенной и на смену ей пришла версия OLE 2. Новая версия позволяла решить вопросы предоставления друг

другу различными программами собственных функций. Данная технология активно внедрялась до 1996 года, после чего ей на смену пришла технология ActiveX, которая включает в себя автоматизацию (OLE-автоматизацию), контейнеры, управляющие элементы, Web-технологию и т.д.

Расширения COM

Технология COM изначально разрабатывалась как ядро для осуществления межпрограммного взаимодействия. Уже на этапе разработки предполагалось расширять возможности технологии при помощи так называемых расширений COM. COM расширяет собственную функциональность, благодаря созданию специализированных наборов интерфейсов для решения конкретных задач.

Технология ActiveX — это технология, которая использует компоненты COM, особенно элементы управления. Она была создана для того, чтобы работа с элементами управления была более эффективной. Это особенно необходимо при работе с приложениями Internet/Intranet, в которых элементы управления должны быть загружены на компьютер клиента, прежде чем они будут использоваться.

В табл. 2.1 и рис. 2.1 представлены некоторые из используемых в настоящее время расширений COM.

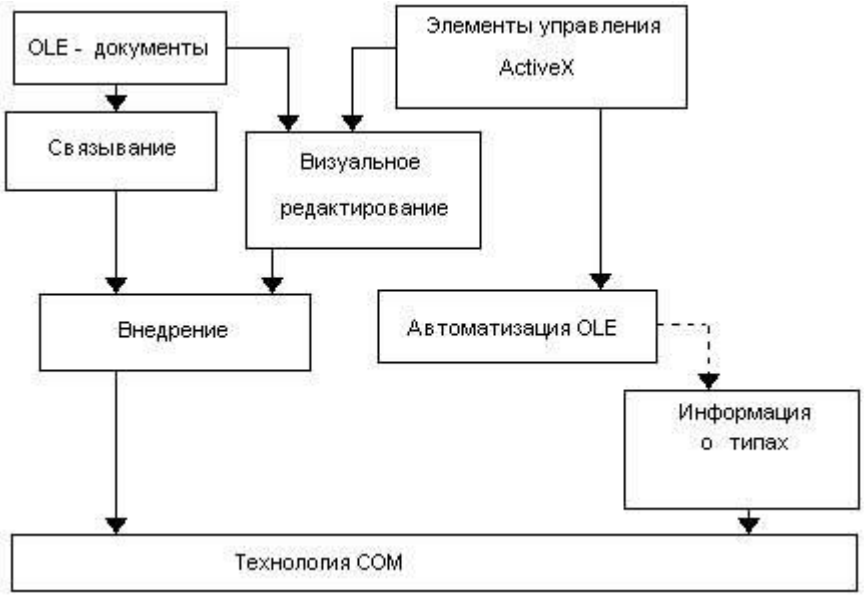


Рисунок 2.1 - Технологии, основанные на COM

Таблица 2.1 - Особенности объектов COM

COM-объект	Визуальность	Процесс	Связь	Библиотека типов
Активный документ (Active Document)	Обычно визуальный	Внутренний или локальный	OLE	Нет
Автоматизация (Automation)	Может быть как визуальным,	Внутренний, локальный	Автоматический маршалинг при помощи интерфейса	Требуется для автоматического

	так и невизуальным	или удаленный	IDispatch	маршалмнга
Элемент управления ActiveX (ActiveX Control)	Обычно визуальный	Внутренний	Автоматический маршалинг при помощи интерфейса IDispatch	Требуется
Произвольный объект интерфейса	По выбору	Внутренний	Не требуется маршалинг	Рекомендуется
Произвольный объект интерфейса	По выбору	Внутренний, локальный или удаленный	Автоматический маршалинг в зависимости от библиотеки типов, в противном случае-ручной маршалинг	Рекомендуется

Терминология COM

COM-объект — двоичный код, который выполняет какую-либо функцию и имеет один или более интерфейсов, в которые входят методы, позволяющие приложению пользоваться COM-объектом. Клиенту достаточно знать несколько базовых интерфейсов COM-объекта, чтобы получить полную информацию о перечне свойств и методов объекта.

Пользователь COM-объекта — приложение или часть приложения, которое использует COM-объект и его интерфейсы в своих собственных целях. Как правило, COM-объект находится в другом приложении.

COM со-классы (coclass) — это классы, которые содержат один или более COM-интерфейс. Вы можете не обращаться к COM-интерфейсу непосредственно, а получать доступ к COM-интерфейсу через со-класс. Со-классы идентифицируются при помощи идентификаторов класса (CLSID).

Библиотека типов — это специальный файл, который содержит информацию о COM-объекте. Данная информация содержит список свойств, методов, интерфейсов, структур и других элементов, которые содержатся в COM-объекте. Библиотека типов содержит также информацию о типах данных каждого свойства и типах данных, возвращаемых методами COM-объекта. Файлы библиотеки типов имеют расширение TLB.

COM-интерфейс — это группы логически или семантически связанных процедур и функций, которые обеспечивают связь между поставщиком услуги (COM-объектом) и его клиентом (пользователем COM). *COM-интерфейсы* применяются для объединения методов COM-объекта. COM-интерфейс позволяет клиенту правильно обратиться к COM-объекту, а объекту — правильно ответить клиенту. COM-объект может иметь несколько интерфейсов, каждый из которых обеспечивает какую-либо функциональность. Названия COM-интерфейсов начинаются с буквы I. Клиент может не знать, какие интерфейсы

имеются у COM-объекта. Каждый COM-объект всегда поддерживает основной COM-интерфейс IUnknown, который применяется для передачи клиенту сведений о поддерживаемых интерфейсах.

По правилам обозначения COM-объектов, интерфейсы COM-объекта обозначаются кружками справа или слева от COM-объекта. Базовый интерфейс IUnknown рисуется кружком сверху от COM-объекта.

На рис. 2.2 схематично изображен стандартный COM-интерфейс.

Рис. 2.2. COM-объект с COM-интерфейсом

Имеются следующие правила для интерфейсов:

Однажды определенные интерфейсы не могут быть изменены. Дополнительную функциональность можно реализовать с помощью дополнительных интерфейсов.

По взаимному соглашению, все имена интерфейсов начинаются с буквы *I,* например IPersist, IMalloc.

Каждый интерфейс гарантированно имеет свой уникальный идентификатор, который называется *глобальный уникальный идентификатор* (Globally Unique Identifier, GUID). Уникальные идентификаторы интерфейсов называют *идентификаторами интерфейсов* (Interface Identifiers, IIDs). Данные идентификаторы обеспечивают устранение конфликтов имен различных версий приложения или разных приложений.

Интерфейсы не зависят от языка программирования. Вы можете воспользоваться любым языком программирования для реализации COM-интерфейса. Язык программирования должен поддерживать структуру указателей, а также иметь возможность вызова функции при помощи указателя явно или неявно.

Интерфейсы обеспечивают доступ к объектам. Таким образом, клиенты не могут напрямую обращаться к данным, доступ осуществляется при помощи указателей интерфейсов.

Все интерфейсы всегда являются потомками базового интерфейса IUnknown.

COM-Интерфейс IUnknown обеспечивает механизм учета ссылок (счетчик ссылок на COM-объект). При передаче указателя на интерфейс выполняется метод интерфейса IUnknown *AddRef*. По завершении работы с интерфейсом приложение-клиент вызывает метод Release, который уменьшает счетчик ссылок.

При вызове метода **QueryInterface** интерфейса **IUnknown** в метод передается параметр IID, имеющий тип TGUID, т. е. идентификатор интерфейса. Параметр метода **out** возвращает либо ссылку на запрашиваемый интерфейс, либо значение **null**. Результатом вызова метода может быть одно из значений, перечисленных в табл. 2.2.

Таблица 2.2. Значения, возвращаемые методом Queryinterface

Значение	Описание
S_OK	Интерфейс поддерживается
E_NOINTERFACE	Интерфейс не поддерживается
E_UNEXPECTED	Неизвестная ошибка

Счетчик ссылок — содержит число процессов (т. е. пользователей COM-объекта), которые в текущий момент времени используют COM-объект. Счетчик ссылок на COM-объект нужен для того, чтобы высвобождать процессорное время и оперативную *память, занимаемую COM-объектом, в том* случае, когда он не используется. Всякий раз, когда новое приложение подключается к COM-объекту — счетчик ссылок этого COM-объекта увеличивается на единицу. Когда процесс отключается от COM-объекта — счетчик ссылок уменьшается на единицу. При достижении счетчиком нуля память, занимаемая COM-объектом, высвобождается.

OLE-объект — часть данных, использующаяся совместно несколькими приложениями. Те приложения, которые могут содержать в себе OLE-объекты, называются *OLE-контейнерами* (OLE container). Приложения, имеющие возможность содержать свои данные в OLE-контейнерах, называются *OLE-серверами* (OLE server).

Указатель COM-интерфейса — указатель на экземпляр объекта, который является, в свою очередь, указателем на реализацию каждого метода интерфейса. Реализация методов доступна через массив указателей на эти методы, который называется **vtable**. Использование массива **vtable** похоже на механизм поддержки виртуальных функций (рис. 2.3).

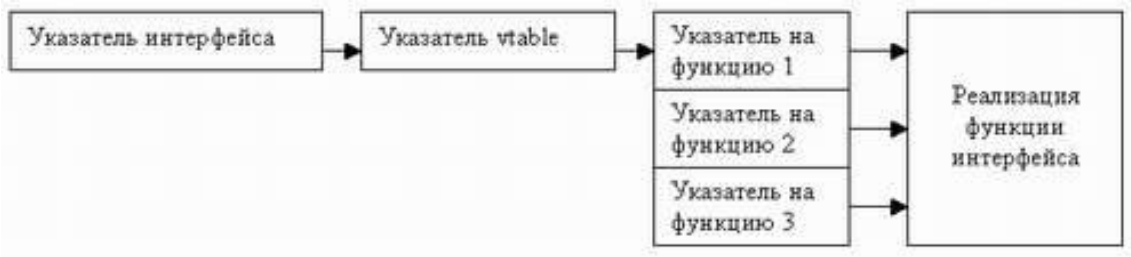


Рис. 2.3. Схема работы указателя COM-интерфейса

COM-сервер представляет собой приложение или библиотеку, которая предоставляет услуги приложению-клиенту или библиотеке. Клиенты не знают, как COM-объект выполняет свои действия. COM-объект предоставляет свои услуги при помощи интерфейсов. В дополнение, приложению-клиенту не нужно знать, где находится COM-объект. Технология COM обеспечивает прозрачный доступ независимо от местонахождения COM-объекта. Когда клиент запрашивает услугу от COM-объекта, он передает COM-объекту идентификатор класса (CLSID). CLSID - всего лишь GUID, который применяется при обращении к COM-объекту. После передачи CLSID, COM-сервер должен обеспечить так называемую *фабрику класса*, которая создает экземпляры COM-объектов.

В общих чертах, COM-сервер должен выполнять следующее:

регистрировать данные в системном реестре Windows для связывания модуля сервера с идентификатором класса (CLSID);

предоставлять фабрику COM-класса, создающую экземпляры COM-объектов;

обеспечивать механизм, который выгружает из памяти серверы COM, которые в текущий момент времени не предоставляют услуг клиентам.

Фабрика класса — это специальный COM-объект, который поддерживает интерфейс IClassFactory и отвечает за создание экземпляров того класса, с которым ассоциирована данная фабрика класса.

Интерфейс IClassFactory имеет два метода: CreateInstance и LockServer. Метод CreateInstance создает экземпляр COM-объекта ассоциированной фабрики класса. Если параметр fLock метода LockServer имеет значение true, то счетчик ссылок сервера увеличивается, иначе — уменьшается. Когда счетчик достигает значения 0, сервер выгружается из памяти. Каждый coclass должен иметь фабрику класса и идентификатор класса CLSID. Использование CLSID для coclass подразумевает, что они могут быть откорректированы всякий раз, когда в класс вводятся новые интерфейсы. Таким образом, в отличие от DLL, новые интерфейсы могут изменять или добавлять методы, не влияя на старые версии.

Составной документ — документ, включающий в себя один или несколько OLE-объектов.

Технология DCOM (Distributed COM) — распределенная COM-технология. Она применяется для предоставления средств доступа к COM-объектам, расположенным на других компьютерах в сети (в том числе и сети Internet).

С использованием COM клиент не должен беспокоиться о том, где располагается объект, он просто делает вызов интерфейса данного объекта. Технология COM обеспечивает все необходимые шаги для того, чтобы сделать этот вызов. Шаги могут отличаться, в зависимости от местонахождения объекта. Объект может находиться в том же процессе, где и клиент, в другом процессе на том же компьютере, где расположен клиент, или на другом компьютере в сети. В зависимости от этого применяются разные типы серверов:

внутренний сервер (In-process server);

локальный сервер или сервер вне процесса (Local server, Out-of-process server);

удаленный сервер (Remote server).

Внутренний сервер — это библиотека DLL, которая запущена в одном процессе вместе с клиентом. Например, элемент управления ActiveX, который внедрен на Web-страницу и просматривается при помощи Internet Explorer. Приложение-клиент связывается с сервером внутри процесса при помощи прямых вызовов COM-интерфейса. На рис. 2.4 представлена схема взаимодействия клиента с внутренним сервером.

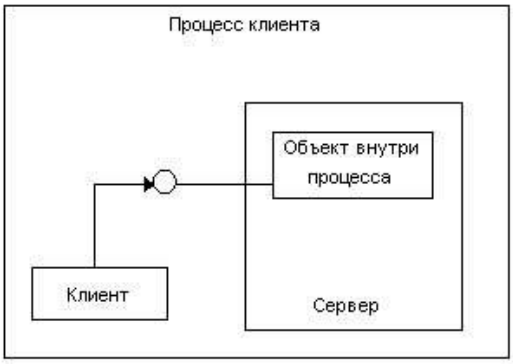


Рисунок 2.4. Схема взаимодействия клиента с внутренним сервером

Внутренний COM-сервер должен экспортировать четыре функции:

function DllRegisterServer: HResult; stdcall;

function DllUnregisterServer: HResult; stdcall;

function DllGetClassObject (const CLSID, IID: TGUID; var Obj): HResult; stdcall;

function DllCanUnloadNow: HResult; stdcall;

Все вышеперечисленные функции уже реализованы в модуле comserv, их нужно только добавить в описания exports вашего проекта.

Рассмотрим данные функции более подробно:

DllRegisterServer - применяется для регистрации DLL COM-сервера в системном реестре Windows. При регистрации COM-класса в системном реестре создается раздел в

HKEY_CZASSES_ROOTCLSID

{XXXXXXXX-XXXX-XXXX-xxxx-xxxxxxxx},

где число, записанное вместо символов x, представляет собой CLSID данного COM-класса. Для внутреннего сервера в данном разделе создается дополнительный подраздел inProcserver32. В этом подразделе указывается путь к DLL внутреннего сервера (рис. 2.5).

DllUnregisterServer — применяется для удаления всех разделов, подразделов и параметров, которые были созданы в системном реестре функцией DllRegisterServer при регистрации DLL COM-сервера.

DllGetclassObject — возвращает фабрику класса для конкретного COM-класса.

DllcanUnloadNow — применяется для определения, можно ли в настоящий момент времени выгрузить DLL COM-сервера из памяти. Функция проверяет, есть ли указатели на любой COM-объект данной DLL, если есть, то возвращает значение S_FALSE, т. е. DLL выгрузить нельзя. Если ни один COM-объект данной DLL не используется, то функция возвращает значение S_TRUE.

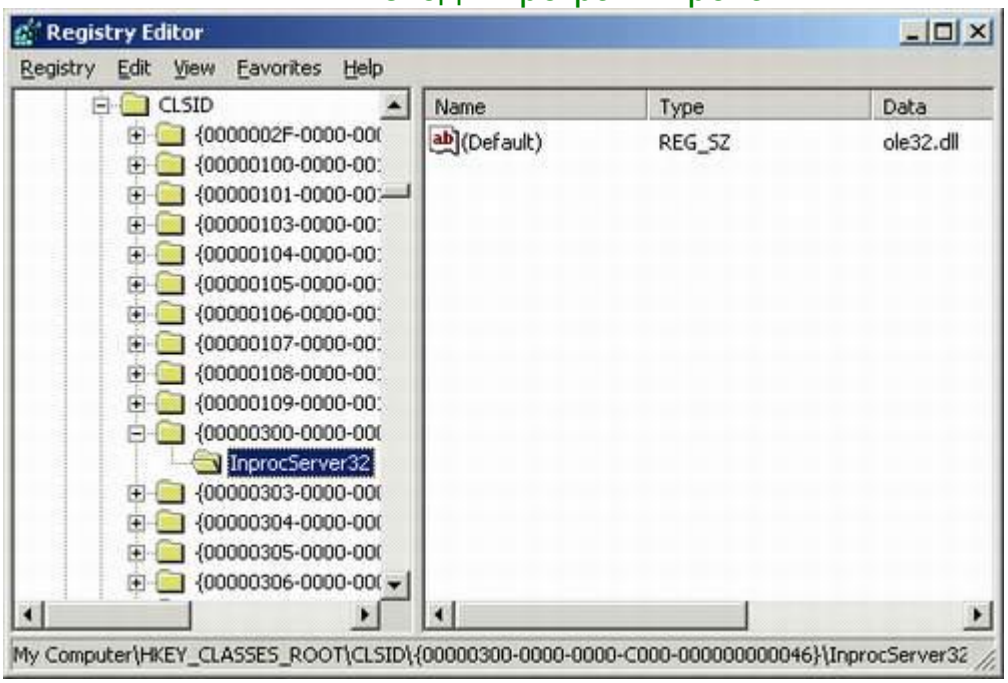


Рисунок 2.5 - Путь к локальному COM-серверу в окне редактора системного реестра

Локальный сервер — это приложение EXE, которое запущено в другом процессе, но на одном компьютере вместе с клиентом. Например, лист электронной таблицы Microsoft Excel связан с документом Microsoft Word. При этом два разных приложения работают на одном компьютере. Локальные серверы используют COM для соединения с клиентом.

Когда клиент и сервер находятся в различных приложениях, а также, когда они находятся на разных компьютерах в сети, COM использует *внутренний (внутрипроцессный) прокси* (In-process proxy) для реализации процедуры удаленного вызова. Прокси располагается в одном процессе вместе с клиентом, поэтому, с точки зрения клиента, вызов интерфейсов осуществляется так же, как и в случае, когда клиент и сер'вер находятся внутри одного процесса. Задача прокси заключается в том, чтобы перехватывать вызовы клиента и перенаправлять их туда, где запущен сервер. Механизм, который позволяет клиенту получать доступ к объектам, расположенным в другом адресном пространстве или на другом компьютере, называется *маршалинг* (marshaling).

Функции маршалинга:

принимать указатель интерфейса из процесса сервера и делать указатель прокси в процессе клиента доступным;

передавать аргументы вызовов интерфейса таким образом, как будто они произошли от клиента и размещать аргументы в процесс удаленного объекта.

Для любого вызова интерфейса клиент помещает аргументы в стек, вызывает необходимую функцию COM-объекта через указатель интерфейса. Если вызов объекта произошел не внутри процесса, вызов проходит через прокси. Прокси упаковывает аргументы в *пакет маршалинга* и передает получившуюся структуру удаленному объекту. *Заглушка* (stub) объекта распаковывает пакет маршалинга, выбирает аргументы из стека и вызывает необходимую функцию COM-объекта.

Таким образом, маршалинг — это процесс упаковки информации, а демаршалинг — процесс распаковки информации.

Тип маршалинга зависит от объектной принадлежности COM. Объекты могут использовать стандартный механизм маршалинга, предоставляемый интерфейсом IDispatch. *Стандартный маршалинг* позволяет устанавливать связь при помощи стандартного системного *удаленного вызова процедуры* (Remote Procedure Call, RFC).

На рис. 2.6 изображена схема, показывающая методику взаимодействия клиента и сервера в случае, когда приложения работают на одном компьютере, но в разных приложениях.

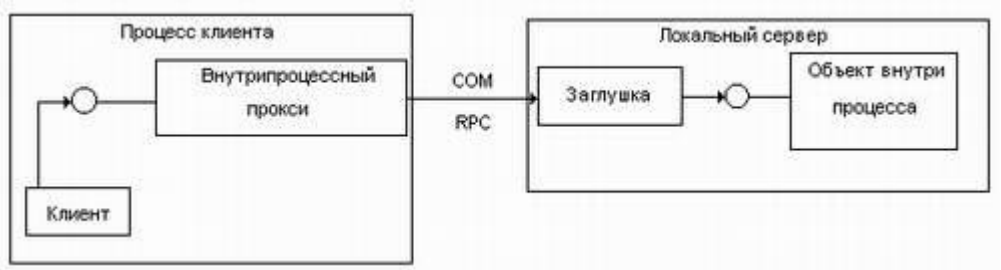


Рисунок 2.6. Схема взаимодействия клиента с сервером в разных процессах на одном компьютере

Локальный COM-сервер регистрируется в системном реестре Windows так же, как и внутренний COM-сервер.

Удаленный сервер — это библиотека DLL или иное приложение, запущенное на другом компьютере. То есть клиент и сервер работают на разных компьютерах в сети. Например, приложение базы данных, написанное с помощью Delphi, соединяется с сервером на другом компьютере в сети. Удаленный сервер использует *распределенные COM-интерфейсы* (Distributed COM, DCOM) для связи с клиентом.

Удаленный сервер работает также с помощью прокси. Различие в работе между локальным и удаленным сервером заключается в типе используемой межпроцессной связи. В случае локального сервера - это COM, а в случае удаленного сервера — DCOM. Схема взаимодействия клиента и удаленного сервера показана на рис. 2.7.

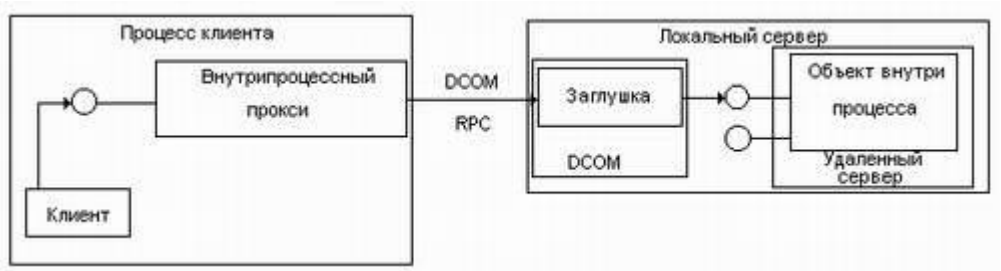


Рисунок 2.7. Схема взаимодействия клиента с сервером на разных компьютерах

Сравнительный анализ технологий CORBA и COM

Технология создавалась консорциумом OMG как универсальная технология создания распределенных систем в гетерогенных средах. Первая спецификация CORBA появилась в 1991 г. Новые возможности официально считаются

добавленными в CORBA в момент утверждения соответствующей спецификации. Как правило, в разработке спецификации участвуют крупнейшие специалисты в данной области. Разработка реализации - задача конкретной фирмы.

Несмотря на внешнюю похожесть, что вызвано общностью решаемых задач, между COM и CORBA, пожалуй, больше различий, чем сходства. В большинстве случаев либо нецелесообразно использовать CORBA (для небольших и простых проектов под Windows просто по причине относительно высоких затрат на приобретение программного обеспечения, лицензий и пр.), либо практически невозможно использовать COM (для сложных, масштабируемых, высоконадежных проектов или просто при работе в гетерогенных средах, а не только в Windows). Windows-приложения, ориентированные на взаимодействие с Microsoft Office, всегда будут использовать COM; проекты с использованием Java и любых Java-технологий (кроме Microsoft J++), будут строить на основе CORBA.

ЛЕКЦИЯ №9

2.6 Регулярные выражения

Введение в использование регулярных выражений в .NET

Регулярное выражение (regular expression) - это не новый язык, а стандарт для поиска и замены текста в строках. Существует два стандарта: основные регулярные выражения (BRE - basic regular expressions) и расширенные регулярные выражения (ERE - extended regular expressions). ERE включает все функциональные возможности BRE.

Наиболее развита поддержка регулярных выражений, до появления .Net, была в Perl (выполнялась на уровне интерпретатора). В настоящее время регулярные выражения поддерживаются практически всеми новыми языками программирования. Обработка регулярных выражений в .Net, хотя и не является встроенной, сами регулярные выражения почти полностью аналогичны регулярным выражениям в Perl и имеют ряд дополнительных возможностей.

Существует два типа механизмов выполнения регулярных выражений: механизм детерминированных конечных автоматов и механизм не детерминированных конечных автоматов. Первый работает быстрее, но не поддерживает сохранение, позиционные проверки и минимальные квантификаторы. Net полностью поддерживает традиционный механизм недетерминированных конечных автоматов (не Posix совместимый). Его принцип действия заключается в последовательном сравнении каждого элемента регулярного выражения с входной строкой и запоминании найденных позиций совпадений. При неудачном дальнейшем поиске выполняется возврат к сохраненным позициям. В заключении, перебираются альтернативные варианты совпадений и выбираются ближайшие к левой границе точки начала поиска (в отличие от Posix совместимого, выбирающего максимально возможное).

Поддержка регулярных выражений в .Net выполняется классами пространства имен:

```
using System.Text.RegularExpressions;
```

Основные классы:

Regex - постоянное регулярное выражение.

Match - предоставляет результаты очередного применения всего регулярного выражения к исходной строке.

MatchCollection - предоставляет набор успешных сопоставлений, при итеративном применении образца регулярного выражения к строке.

Capture - предоставляет результаты отдельного захвата подвыражения.

Group - предоставляет результаты для одной регулярной группы.

GroupCollection - предоставляет коллекцию найденных групп и возвращает набор групп как одно соответствие.

CaptureCollection - предоставляет последовательность найденных подстрок и возвращает наборы соответствий отдельно для каждой группы.

Более подробно на использовании классов мы остановимся ниже, после рассмотрения синтаксиса регулярных выражений.

Таблица 2.3. Символы, используемые в регулярных выражениях

Символ	Интерпретация
	Категория: escape-последовательности
\b	При использовании его в квадратных скобках соответствует символу "обратная косая черта" с кодом - \u0008
\t	Соответствует символу табуляции \u0009
\r	Соответствует символу возврата каретки \u000D
\n	Соответствует символу новой строки \u000A
\e	Соответствует символу escape \u001B
\040	Соответствует символу ASCII, заданному кодом до трех цифр в восьмеричной системе
\x20	Соответствует символу ASCII, заданному кодом из двух цифр в шестнадцатиричной системе
\u0020	Соответствует символу Unicode, заданному кодом из четырех цифр в шестнадцатиричной системе
	Категория: подмножества (классы) символов
.	Соответствует любому символу, за исключением символа конца строки
[aeiou]	Соответствует любому символу из множества, заданного в квадратных скобках
[^aeiou]	Отрицание. Соответствует любому символу, за исключением символов, заданных в квадратных скобках
[0-9a-fA-F]	Задание диапазона символов, упорядоченных по коду. Так, 0-9 задает любую цифру
\p{name}	Соответствует любому символу, заданному множеству с именем name, например, имя Ll задает множество букв латиницы в нижнем регистре. Поскольку все символы разбиты на подмножества, задаваемые категорией Unicode, то в качестве имени можно задавать имя категории
\P{name}	Отрицание. Большая буква всегда задает отрицание множества, заданного малой буквой
\w	Множество символов, используемых при задании идентификаторов - большие и малые символы латиницы, цифры и знак подчеркивания
\s	Соответствует символам белого пробела
\d	Соответствует любому символу из множества цифр
	Категория: Операции (модификаторы)
*	Итерация. Задает ноль или более соответствий; например, \w*

(abc)*.	Аналогично, {0,}
+	Положительная итерация. Задает одно или более соответствий; например, \w+ или (abc)+. Аналогично, {1,}
?	Задает ноль или одно соответствие; например, \w? или (abc)?. Аналогично, {0,1}
{n}	Задает в точности n соответствий; например, \w{2}
{n,}	Задает, по меньшей мере, n соответствий; например, (abc){2,}
{n,m}	Задает, по меньшей мере, n, но не более m соответствий; например, (abc){2,5}
	Категория: Группирование
(?<Name>)	При обнаружении соответствия выражению, заданному в круглых скобках, создается именованная группа, которой дается имя Name. Например, (?<tel> \d{7}). При обнаружении последовательности из семи цифр будет создана группа с именем tel
()	Круглые скобки разбивают регулярное выражение на группы. Для каждого подвыражения, заключенного в круглые скобки, создается группа, автоматически получающая номер. Номера следуют в обратном порядке, поэтому полному регулярному выражению соответствует группа с номером 0
(?imnsx)	Включает или выключает в группе любую из пяти возможных опций. Для выключения опции перед ней ставится знак минус. Например, (?i-s:) включает опцию i, задающую нечувствительность к регистру, и выключает опцию s - статус single-line

Основные элементы синтаксиса регулярных выражений

Элементарные примеры в таблице даны для подстановки строки и регулярного выражения в две функции, в которых одновременно демонстрируется и использование классов Match и MatchCollection.

Пример 1. Функция для однократного (первого вхождения) поиска соответствия регулярного выражения, заданного объектом класса Regex в строке s.

```
string s = "Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22";
Regex regex = new Regex("дя|я");
//Код 1
Match match = regex.Match(s);
if (match.Success)
{
Console.WriteLine(match.Value.ToString());
```

```
}
```

Будет выведено: я
(первая буква я в строке
"Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22"
)

Пример 2. Для поиска всех вхождений необходимо использовать код

```
Regex regex = new Regex("дя|я");
string s = "Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22";
//Код 2
MatchCollection matchcollection = regex.Matches(s);
  Console.WriteLine();
for (int i = 0; i < matchcollection.Count; i++)
{
Console.WriteLine(matchcollection[i].Value);
}
```

Будет выведено:
я
дя
я
я
"Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22"

Пример 3. Поиск всех групп вхождений. Группы задаются круглыми скобками.

```
string s = "a1ba2ba3ba4b";
Regex regex = new Regex(".(a.b)(a.b)");
//Код 3
Match match = regex.Match(s);
if (match.Success)
{
 for (int i = 0; i < match.Groups.Count; i++)
 {
   Console.WriteLine(match.Groups[i].Value.ToString());
 }
}
```

ba2ba3b
a2b
a3b

Пример 4. Поиск всех вхождений.

```
string s = "a1ba2ba3ba4b";
          Regex regex = new Regex(".(a.b)(a.b)");
```

```
//Код 2
        MatchCollection matchcollection = regex.Matches(s);
           for (int i = 0; i < matchcollection.Count; i++)
            {
              Console.WriteLine(matchcollection[i].Value);
            }
```

Будет выведено
ba2ba3b

Пример 5. Поиск всех вхождений.

```
string s = "a1ba2ba3ba4b";
        Regex regex = new Regex("(a.b)(a.b)");
//Код 2
        MatchCollection matchcollection = regex.Matches(s);
        for (int i = 0; i < matchcollection.Count; i++)
        {
Console.WriteLine(matchcollection[i].Value);
        }
```

```
string s = "a1ba2ba3ba4b";
Regex regex = new Regex("(a.b)(a.b)");
```

Поиск всех вхождений
a1ba2b
a3ba4b

Для

```
        string s = "999Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22";
        Regex regex = new Regex("[.]*[^0-9]+");
```

Код 1 находит
Дядя: (
Код 2 находит все не цифровые строки:
Дядя: (
)
Тетя:
-
-
Петя: (
)
-
-

Таблица 2.4

Символ	Значение	Примеры применения
.	Любой одиночный символ, кроме символа новой строки \n.	Regex("a.b"); Код 1 Строка s = "a1ba2ba3ba4b"; Результат: a1b Код 2 Строка s = "a1ba2ba3ba4b"; Результат: a1b a2b a3b a4b
\	Определение метасимвола	Символ "." соответствует любому символу, а "\." точке . Regex(@"a\.b"); в предыдущем примере ничего не найдет Код 1 Строка s = "a1ba2ba3ba.b"; Результат: a.b Код 2 Строка s = "a1ba.ba3ba.b"; Результат: a.b a.b
|	Разделение шаблона (или)	Regex("a|b"); Код 1 Строка s = "abcabcabcabc"; Результат: a Код 2 Строка s = "abcabcabcabc"; Результат: a b a b a b a b
[мн-во сим-в]	Соответствует любому символу из данного множества (одному и только одному)	Regex("[abc]"); Код 1 Строка s = "abcdabcdabcdabcd"; Результат: a Код 2 Строка s = "abcdabcdabcdabcd"; Результат: a b c a b c a b c a b c
[^мн-во сим-в]	Отрицание множества символов	Regex("[^abc]"); Код 1 Строка s = "abcdabcdabcdabcd"; Результат: d Код 2 Строка s = "abcdabcdabcdabcd"; Результат: d d d d
^	Соответствует началу строки Отрицание - если первый в квадратной скобке	Regex("^abc"); Код 1 Строка s = "abcdefabcdef"; Результат: abc Код 2 Строка s = "abcdefabcdef"; Результат: abc Regex("^bc"); в обоих примерах ничего не найдет
$	Соответствует концу строки. Это конец строки или позиция перед символом начала новой строки.	Regex("def$"); Код 1 Строка s = "abcdefabcdef"; Результат: def Код 2 Строка s = "abcdefabcdef"; Результат: def

(....)	Группировка элементов.	Regex("(ab)"); Код 1 Строка s = "abcdefabcdef"; Результат: ab ab Код 2 Строка s = "abcdefabcdef"; Результат: ab ab
?	Повторение 0 или 1 раз, стоящего перед	Regex("ab?"); Код 1 Строка s = "aabbbcde"; Результат: a Код 2 Строка s = "aabbbcde"; Результат: a ab Regex("a?b?"); Код 1 Строка s = "aabbbcde"; Результат: a Код 2 Строка s = "aabbbcde"; Результат: a ab b b Regex("a?b"); Код 1 Строка s = "aabbbcde"; Результат: ab Код 2 Строка s = "aabbbcde"; Результат: ab b b
*****	Повторение 0 или более раз стоящего перед	Regex("ab*"); Код 1 Строка s = "aabbbcde"; Результат: a Код 2 Строка s = "aabbbcde"; Результат: a abbb --- Regex("a*b*"); Код 1 Строка s = "aabbbcde"; Результат: aabbb Код 2 Строка s = "aabbbcde"; Результат: aabbb --- Regex("a*b"); Код 1 Строка s = "aabbbcde"; Результат: aab Код 2 Строка s = "aabbbcde"; Результат: aab b b
+	Повторение 1 или более раз стоящего перед ним	Regex("ab+"); Код 1 Строка s = "aabbbcde"; Результат: abbb Код 2 Строка s = "aabbbcde"; Результат: abbb --- Regex("a+b+");

		Код 1 Строка s = "aabbbcde"; Результат: aabbb Код 2 Строка s = "aabbbcde"; Результат: aabbb --- Regex("a+b"); Код 1 Строка s = "aabbbcde"; Результат: aab Код 2 Строка s = "aabbbcde"; Результат: aab
{n}	Повторение точно n раз	Regex("ab{2}"); Код 1 Строка s = "aabbbcde"; Результат: abb Код 2 Строка s = "aabbbcde"; Результат: abb
{n, m}	повторение от n до m раз	Regex("ab{2,4}"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbb Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbb abbbb
{n,}	повторение n и более раз	Regex("ab{2,}"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abb Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbb abbbbb
***?**	повторение 0 или более раз, минимально возможное количество	Regex("ab*?c"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc abbbbbc
+?	повторение 1 или более раз, минимально возможное количество	Regex("ab+?c"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc abbbbbc
??	повторение 0 или 1 раз, минимально возможное количество	Regex("ab??c"); Код 1 Строка s = "acdeabcdeabbcde"; Результат: ac Код 2 Строка s = "acdeabcdeabbcde"; Результат: abc

\w	Слово (цифра или буква или знак подчеркивания)	Regex(@"\w"); Код 1 Строка s = "abcd_~`!@#$%^&*()-=+\|:;\",.<>?/12345"; Результат: A Код 2 Строка s = "abcd_~`!@#$%^&*()-=+|:;\",.<>?/12345"; Результат: A Ф b c d _ 1 2 3 4 5
\W	Не слово (не цифра и не буква)	Regex(@"\W"); Код 1 Строка s = "abcd_~`!@#$%^&*()-=+\|:;\",.<>?/12345"; Результат: ~ Код 2 Строка s = "abcd_~`!@#$%^&*()-=+|:;\",.<>?/12345"; Результат: ~ ` ! @ # $ % ^ & * ( ) - = + | : ; " , . < > ? /
\d	Десятичная цифра	Regex(@"\d"); Код 1 Строка s = "abcd12345"; Результат: 1 Код 2 Строка s = "abcd12345"; Результат: 1 1 2 3 4 5
\D	Не десятичная цифра	Regex(@"\D"); Код 1 Строка s = "abcd12345"; Результат: a Код 2 Строка s = "abcd12345"; Результат: a b c d _ ~ ` ! @ # $ % ^ & * ( ) - = + | : ; " , . < > ? /
\s	Пустое место (пробел, \f, \n, \r, \t, \v)	Regex(@"\s"); Код 1 Строка s = "ab cd"; Результат: b c Код 2 Строка s = "ab cd"; Результат: b c
\S	Не пустое место (не пробел, не \f, не \n, не \r, не \t, не \v)	Regex(@"\S"); Код 1 Строка s = "ab c\td"; Результат: a Код 2 Строка s = "ab c\td"; Результат: a b c d

\b	Граница слова (Символы для поиска пишутся:
	для начала слова - справа;
	для конца - слева.)
		Regex(@"\bqw"); //ничего не будет найдено при Regex(@"\bwe");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: qw
		Код 2
		Строка s = "abcde"; Результат: qw qw
		--- 	
		Regex(@"ty\b"); //ничего не будет найдено при Regex(@"rt\b");

		Код 1 Строка s = "abcde qwerty qwerty"; Результат: ty Код 2 Строка s = "abcde qwerty qwerty"; Результат: ty ty

\B	Не граница слова (Символы для поиска пишутся: для начала слова - справа;
	для конца - слева.)
		Regex(@"\Bwe"); //ничего не будет найдено при Regex(@"\qw");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: we
		Код 2
		Строка s = "abcde"; Результат: we we
		--- 	
		Regex(@"rt\B"); //ничего не будет найдено при Regex(@"ty\B");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: rt
		Код 2
		Строка s = "abcde qwerty qwerty"; Результат: rt rt

\A	"Истинное" начало строки	Regex(@"ab\A"); //ничего не будет найдено при Regex(@"qw\A");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: ab
		Код 2
		Строка s = "abcde qwerty qwerty"; Результат: ab

\Z	Конец строки, позиция перед символом начала новой строки.	Regex(@"ty\Z"); Код 1 Строка s = "abcde qwerty qwerty\n"; Результат: ty Код 2 Строка s = "abcde qwerty qwerty"; Результат: ty
\z	"Истинный" конец строки, позиция перед символом начала новой строки.	Regex(@"ty\n\z"); Код 1 Строка s = "abcde qwerty qwerty\n"; Результат: ty + нечитаемый символ Код 2 Строка s = "abcde qwerty qwerty"; Результат: ty + нечитаемый символ

В регулярных выражениях допускаются метасимволы: \t - горизонтальная табуляция, \r - возврат курсора, \n - перевод строки, \v - вертикальная табуляция, \e - Escape, \f - подача страница, \0AA - символ представленный двумя цифрами (AA) восьмеричного кода, \xAA - символ представленный двумя цифрами (AA) шестнадцатеричного кода, \xAAAA- символ представленный четырьмя цифрами (AAAA) шестнадцатеричного кода, \a - звуковой сигнал.

Знакомство с классами пространства RegularExpressions

В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.

Класс Regex

Это основной класс, всегда создаваемый при работе с регулярными выражениями. Объекты этого класса определяют регулярные выражения. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая регулярное выражение. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению RegexOptions и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать регулярное выражение. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Рассмотрим четыре основных метода класса Regex.

Метод Match запускает поиск соответствия. В качестве параметра методу передается строка поиска, где разыскивается первая подстрока, которая удовлетворяет образцу, заданному регулярным выражением.В качестве результата метод возвращает объект класса Match, описывающий результат поиска. При успешном поиске свойства объекта будут содержать информацию о найденной подстроке.

Метод Matches позволяет разыскать все вхождения, то есть все подстроки, удовлетворяющие образцу. У алгоритма поиска есть важная особенность - разыскиваются непересекающиеся вхождения подстрок. Можно считать, что метод Matches многократно запускает метод Match, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск. В качестве результата возвращается объект MatchCollection, представляющий коллекцию объектов Match.

Метод NextMatch запускает новый поиск, начиная с того места, на котором остановился предыдущий поиск.

Метод Split является обобщением метода Split класса String. Он позволяет, используя образец, разделить искомую строку на элементы. Поскольку образец может быть устроен сложнее, чем простое множество разделителей, то метод Split класса Regex эффективнее, чем его аналог класса String.

Классы Match и MatchCollection

Как уже говорилось, объекты этих классов создаются автоматически при вызове методов Match и Matches. Коллекция MatchCollection, как и все коллекции, позволяет получить доступ к каждому ее элементу - объекту Match. Можно, конечно, организовать цикл foreach для последовательного доступа ко всем элементам коллекции.

Класс Match является непосредственным наследником класса Group, который, в свою очередь, является наследником класса Capture. При работе с объектами класса Match наибольший интерес представляют не столько методы класса, сколько его свойства, большая часть которых унаследована от родительских классов. Рассмотрим основные свойства:

свойства Index, Length и Value наследованы от прародителя Capture. Они описывают найденную подстроку- индекс начала подстроки в искомой строке, длину подстроки и ее значение;

свойство Groups класса Match возвращает коллекцию групп - объект GroupCollection, который позволяет работать с группами, созданными в процессе поиска соответствия;

свойство Captures, наследованное от объекта Group, возвращает коллекцию CaptureCollection. Как видите, при работе с регулярными выражениями реально приходится создавать один объект класса Regex, объекты других классов автоматически появляются в процессе работы с объектами Regex.

Классы Group и GroupCollection

Коллекция GroupCollection возвращается при вызове свойства Group объекта Match. Имея эту коллекцию, можно добраться до каждого объекта Group, в нее входящего. Класс Group является наследником классаCapture и, одновременно, родителем класса Match. От своего родителя он наследует свойства Index, Length и Value, которые и передает своему потомку.

Давайте рассмотрим чуть более подробно, когда и как создаются группы в процессе поиска соответствия. Если внимательно проанализировать предыдущую таблицу, которая описывает символы, используемые в регулярных выражениях, в частности символы группирования, то можно понять несколько важных фактов, связанных с группами:

Факты, связанные с группами:

при обнаружении одной подстроки, удовлетворяющей условию поиска, создается не одна группа, а коллекция групп;

группа с индексом 0 содержит информацию о найденном соответствии;

число групп в коллекции зависит от числа круглых скобок в записи регулярного выражения. Каждая пара круглых скобок создает дополнительную группу, которая описывает ту часть подстроки, которая соответствует шаблону, заданному в круглых скобках;

группы могут быть индексированы, но могут быть и именованными, поскольку в круглых скобках разрешается указывать имя группы.

В заключение отмечу, что создание именованных групп крайне полезно при разборе строк, содержащих разнородную информацию. Примеры разбора подобных текстов будут даны.

Классы Capture и CaptureCollection

Коллекция CaptureCollection возвращается при вызове свойства Captures объектов класса Group и Match. Класс Match наследует это свойство у своего родителя - класса Group. Каждый объект Capture, входящий в коллекцию, характеризует соответствие, захваченное в процессе поиска, - соответствующую подстроку. Но поскольку свойства объекта Capture передаются по наследству его потомкам, то можно избежать непосредственной работы с объектами Capture. По крайней мере, в моих примерах не встретится работа с этим объектом, хотя "за кулисами" он непременно присутствует.

Перечисление RegexOptions

Объекты этого перечисления описывают опции, влияющие на то, как устанавливается соответствие. Обычно такой объект создается первым и передается конструктору объекта класса Regex. В вышеприведенной таблице, в разделе, посвященном символам группирования, говорится о том, что опции можно включать и выключать, распространяя, тем самым, их действие на участок шаблона, заданный соответствующей группой. Об одной из этих опций - Compiled, влияющей на эффективность работы регулярных выражений, уже упоминалось.

Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска регулярные выражения. В этом случае необходимо будет создать объект класса RegexCompilationInfo и передать ему информацию о регулярных выражениях, подлежащих компиляции, и о том, куда поместить оттранслированную программу. Дополнительно в таких ситуациях следует включить опцию Compiled.

Примеры работы с регулярными выражениями

Полагаю, что примеры дополнят краткое описание возможностей регулярных выражений и позволят лучше понять, как с ними работать. Начну с функции FindMatch, которая производит поиск первого вхождения подстроки, соответствующей образцу:

Пример 1. Поиск первого вхождения подстроки, соответствующей образцу:

```
string FindMatch(string str, string strpat)
{
  Regex pat = new Regex(strpat);
  Match match =pat.Match(str);
  string found = "";
  if (match.Success)
  {
          found =match.Value;
          Console.WriteLine("Строка ={0}\tОбразец={1}\
                tНайдено={2}", str,strpat,found);
  }
  return(found);
}//FindMatch
```

В качестве входных аргументов функции передается строка str, в которой ищется вхождение, и строка strpat, задающая образец - регулярное выражение. Функция возвращает найденную в результате поиска подстроку. Если соответствия нет, то возвращается пустая строка. Функция начинает свою работу с создания объекта pat класса Regex, конструктору которого передается образец поиска. Затем вызывается метод Match этого объекта, создающий

объект match класса Match. Далее анализируются свойства этого объекта. Если соответствие обнаружено, то найденная подстрока возвращается в качестве результата, а соответствующая информация выводится на печать._(Чтобы спокойно работать с классами регулярных выражений, я не забыл добавить в начало проекта предложение: using System.Text.RegularExpressions.)

Примеры:

```
public void TestSinglePat()
{
  //поиск по образцу первого вхождения
  string str,strpat,found;
  Console.WriteLine("Поиск по образцу");
  //образец задает подстроку, начинающуюся с символа a,
  //далее идут буквы или цифры.
  str ="start"; strpat =@"a\w+";
  found = FindMatch(str,strpat);
  str ="fab77cd efg";
  found = FindMatch(str,strpat);
  //образец задает подстроку,начинающуюся с символа a,
  //заканчивающуюся f с возможными символами b и d в середине
  strpat = "a(b|d)*f";  str = "fabadddbdf";
  found = FindMatch(str,strpat);
  //диапазоны и escape-символы
  strpat = "[X-Z]+"; str = "aXYb";
  found = FindMatch(str,strpat);
  strpat = @"\u0058Y\x5A"; str = "aXYZb";
  found = FindMatch(str,strpat);
}//TestSinglePat
```

Некоторые комментарии к этой процедуре.

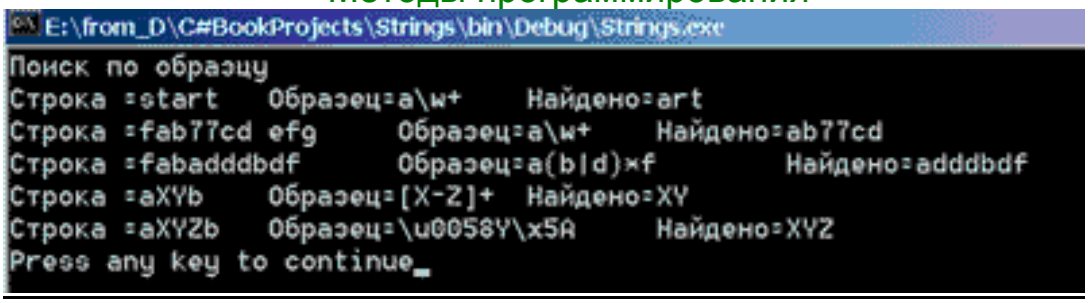
Регулярные выражения задаются @ -константами.

В первом образце используется последовательность символов \w+, обозначающая, как следует из таблицы 15.1, непустую последовательность латиницы и цифр. В совокупности образец задает подстроку, начинающуюся символом a, за которым следуют буквы или цифры (хотя бы одна). Этот образец применяется к двум различным строкам.

В следующем образце используется символ * для обозначения итерации. В целом регулярное выражение задает строки, начинающиеся с символа a и заканчивающиеся символом f, между которыми находится, возможно, пустая последовательность символов из b и d.

Последующие два образца демонстрируют использование диапазонов и escape-последовательностей для представления символов, заданных кодами (в Unicode и шестнадцатиричной кодировке).

Результаты, полученные при работе этой процедуры.



Приоритет групповых регулярных выражений:

Круглые скобки ( )(?:...)
Множители ?, +, *, {m,n}
Последовательность и фиксация abc,\A, \Z
Дизъюнкция |

Примеры использования групповых регулярных выражений

Извлечь адрес электронной почты из строки

```
string s = "Почта: wladm@narod.ru Сайт: http://wladm.narod.ru ";
Regex regex = new Regex
 (@"\b\w+([\.\w]+)*\w@\w((\.\w)*\w+)*\.\w{2,3}\b");
```

В регулярном выражении \b определяет начало слова, далее \w+ слово (цифра или буква) повторенные 1 или более раз. Следующую часть строки ([\.\w]+)*\w@ для данного примера можно было заменить на @, но адрес может иметь еще слова через точку до @. Поэтому точка или слово (в квадратных скобках) повторенные 1 или более раз и все это может быть повторено 0 или более раз. Следующий символ \w@ тоже можно сократить \w, но символ @ обязателен. Он якорный для распознавания почтового адреса. Далее идет практически повторение того, что было до символа @ - повторение слов с точкой и в конце слово от двух до трех символов (ru, com....).

```
string s = "Почта: wladm@narod.ru Сайт: http://wladm.narod.ru ";
Regex regex = new Regex(@"\b\w+([\.\w]+)*\w@\w((\.\w)*\w+)*\.\w{2,3}\b");

Match match = regex.Match(s);
MatchCollection matchcollection = regex.Matches(s);

if (match.Success)
{
        Console.WriteLine(match.Groups[i].Value.ToString());
}


string s = "Почта: wladm@narod.ru Сайт: http://wladm.narod.ru ";
        Regex regex = new Regex(@"((П)о(чт))(а)");
      string s = "Почта: wladm@narod.ru Сайт: http://wladm.narod.ru ";
        Regex regex = new Regex(@"((П)о(чт))(а)");

 Match match = regex.Match(s);
```

```
MatchCollection matchcollection = regex.Matches(s);

if (match.Success)
{
 for (int i = 0; i < match.Groups.Count; i++)
 {
  Console.WriteLine(match.Groups[i].Value.ToString());
 }
}
```

По порядку открывающихся скобок

Почта

Почт

П

чт

а

Извлечь адрес сайта из строки

Для примера взяты 2 адреса, короткий и длинный - реальный адрес MSDN с описанием класса Match. Первый адрес также показывает, что классы успешно работают с национальными кодировками.

```
string s = "http://msdn.microsoft.com/иванов.html Сайт: ";
s+="http://msdn.microsoft.com/library/rus/default.asp?url=";
s+="/library/RUS/cpref/html/frlrfSystemTextRegularExpressionsMatchClassTopic.asp";
Regex regex =
new Regex(@"(\b\w+:\/\/\w+((\.\w)*\w+)*\.\w{2,3}(\/\w*|\.\w*|\?\w*\=\w*)*)");
```

Пример похож на предыдущий, отличие в начале:

\b\w+:\/\/ выделяет http://

и в конце: (\/\w*|\.\w*|\?\w*\=\w*)* - перечислено с какими знаками могут повторяться буквы ("/",".","?","=") и все они объединены операцией или "|".

Результат:

```
http://msdn.microsoft.com/иванов.html
http://msdn.microsoft.com/library/rus/default.asp?url=
  /library/RUS/cpref/html/frlrfSystemTextRegularExpressionsMatchClassTopic.asp
```

Выделение цифр из строки

Пример:

```
string s = "Приход = +225 Расход: 211 Остаток: 335 Выручка 2222 Долги -357.8 ";
Regex regex = new Regex(@"[-+]?\d*\.?\d*");
```

Здесь [-+]? - знак перед цифрой повторенный 0 или 1 раз, \. точка повторенная 0 или 1 раз и цифры, которые могут быть повторены 0 и более раз.

Результат:

```
225        211        335        2222        -357.8
```

Удаление пробелов из текста

```
string
```

```
s = "   Это мой текст -  был с     пробелами: в начале, в конце и в середине?    ";
Regex regex = new Regex(@"\b(\w+)\,?\??\!?(\ \-\ )?\:?\ ?");
.....
//При формировании строки из слов, к каждому слову не надо
//добавлять пробел (его находит \ ?).
Text += matchcollection[i].Value;
```

Результат:

Это мой текст - был с пробелами: в начале, в конце и в середине?

Пример:

```
string s = "Дядя: 812-555-55-55 Тетя: 555-55-55 Петя: 848-222-22-22";
Regex regex = new Regex
 (@"\b(\d{3}){0,1}(\-){0,1}(\d{3}\-)(\d{2}\-)(\d{2})\b");
```

В примере все знакомые конструкции. В начале слова код города с черточкой, которого может и не быть. Результат:

812-555-55-55 555-55-55 848-222-22-22

Более усложненный пример, понимающий код города в скобках и номер телефона с (и) без:

```
string s = "Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22";
Regex regex = new Regex
 (@"((\(\b\d{3}\))|(\b(\d{3}){0,1}(\-){0,1}))(\d{7}|((\d{3}\-)(\d{2}\-)(\d{2})\b))");
```

Особенность примера в том, что начало слова не может быть отнесено к скобке и, поэтому, скобка фиксируется как отдельный элемент, а начало слова относится к тому, что в слове.

Результат:

(812)5555555 555-55-55 (848)222-22-22

Разбор сложного текстового файла содержащего символы табуляции

В данном примере используется реальный файл ведомости платежей, рассылаемый Мегафон в организации, сотрудники которой пользуются в служебных целях данной связью. Фрагмент данного файла содержит семизначный номер телефона, четыре колонки начислений за пользование различными видами связи и колонку суммарных значений платежа.

```
1111111     412.86              288.67                    701.53
2222222     626.63                                626.63
3333333
4444444     732.96              285.34      54.40         1072.70
5555555 288.23      135.00      156.74                    579.97
```

Пример кажется простым до того момента, пока мы не посмотрим его строковое представление. Символы табуляции в файле обязательно идут только до номера телефона. Далее они "почти непредсказуемы" и их количество зависит от того, есть ли цифра в следующей колонке. При наличии цифры это два tab, при отсутствии один. Задача - превратить эту "мешанину" tab в нули там, где это необходимо.

\t1111111\t412.86\t\t288.67\t\t\t701.53
\t2222222\t626.63\t\t\t\t\t626.63
\t3333333\t\t\t\t\t\t\t
\t4444444\t732.96\t\t285.34\t54.40\t1072.70
\t5555555\t288.23\t135.00\t156.74\t\t\t579.97

Выполнить задачу можно только поэтапно:

Этап 1 - извлекаем номер телефона:

```
string sS=СОДЕРЖИМОЕ СТРОКИ С ЗАПИСЬЮ О СЧЕТЕ;
Regex r = new Regex(@"\d{7}", RegexOptions.IgnoreCase);
Match mymatch = r.Match(sS);
if(mymatch.Success)
{
 string sTel = mymatch.Groups[0].Value.ToString();
```

Этап 2 - сокращаем строку, убирая из нее номер телефона:

```
sS = sS.Substring(sS.IndexOf(sTel) + 8, sS.Length - sS.IndexOf(sTel) - 8);
```

Этап 3 - заменяем в строке все \t перед которыми есть цифра пробелами:

```
Regex r1 = new Regex(@"(?<=\d)\t",RegexOptions.IgnoreCase);
sS = r1.Replace(sS," ");
```

Этап 4 - заменяем оставшиеся \t нулями:

```
Regex r2 = new Regex(@"\t",RegexOptions.IgnoreCase);
sS = r2.Replace(sS1, " 00.00 ");
```

Этап 5 - Извлекаем цифры - теперь они на своих местах:

```
Regex r3 = new Regex(@"(\d+\.\d+)",RegexOptions.IgnoreCase);
```

Далее можно использовать извлеченные цифры:

```
MatchCollection matchcollection = r3.Matches(sS);
```

Жадность квантификаторов

Квантификаторы или повторители шаблонов (+ и *) обладают т.н. жадностью - т.е. возвращают самый длинный фрагмент текста, соответствующий шаблону.

```
string s =
"Это мой текст был с пробелами в начале, в конце и в середине?";
Regex regex = new Regex(@"\bЭ(.*)о");
//или
..Regex regex = new Regex(@"\bЭ(.+)о");
```

Результат:

Это мой текст был с пробелами в начале, в ко

Конструктор класса Regex и его основные методы

Конструктор класса Regex имеет два перегружаемых метода, для второго из которых options - поразрядная комбинация OR значений перечисления RegexOption.

```
public Regex
(
 string pattern,
```

RegexOptions options

);

Основные опции определяются, как логическое "И" RegexOptions опций.

Compiled - регулярное выражение компилируется в сборку, что значительно увеличивает скорость их выполнения, но снижает скорость загрузки. Кроме того, скомпилированные регулярные выражения выгружаются только при завершении работы приложения, даже когда сам объект Regex освобождён и уничтожен сборщиком мусора.

CultureInvariant - игнорировать культурные различия в языках.

ECMAScript - включить поведение для выражения, соответствующее ECMAScript. Используется только в соединении с флагами IgnoreCase, Multiline и Compiled, с другими дает ошибку.

ExplicitCapture - Указывает, что только верные явно названные или пронумерованные (с помощью конструкции (?)) группы сохраняются. Это позволяет избежать излишнего использования конструкции (?::).

IgnoreCase - игнорировать регистр.

IgnorePatternWhitespace - устраняет пробелы из шаблонов и позволяет использовать комментарии, отмеченные знаком "#".

Multiline - значения ^ и $ обозначают начало и конец каждой строчки текста.

RightToLeft - поиск справа налево.

SingleLine - весь текст как одна строка. Символ точки в данном режиме включает и \n.

Основные методы класса - это:

Match - поиск соответствия. Метод возвращает объект класса Match, содержащий результаты найденного соответствия.

Matches - поиск всех непересекающихся соответствий и возвращение объекта MatchCollection.

NextMatch - продолжение поиска соответствия для Match с позиции последнего найденного.

Эти методы мы уже многократно использовали при рассмотрении синтаксиса регулярных выражений. Далее остановимся на других методах класса.

ГЛАВА 3. Изучение и разработка алгоритмов

ЛЕКЦИЯ №10

3.1 Рекурсивные алгоритмы

Рекурсивный алгоритм — алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. В технике процедурного программирования данное понятие распространяется на функцию, которая реализует решение отдельного блока задачи посредством вызова из своего тела других функций, в том числе и себя самой. Если при этом на очередном этапе работы функция организует обращение к самой себе, то такая функция является рекурсивной.

Рекурсивный вызов процедуры мало чем отличается от вызова другой функции. Выполняется тот же код, но с другими значениями параметров и локальных переменных.

Важным для понимания идеи рекурсии является то, что в подобных подпрограммах можно выделить две серии шагов. Первая серия – это шаги рекурсивного погружения подпрограммы в себя до тех пор, пока выбранный параметр не достигнет граничного значения. Это важное требование должно выполняться, чтобы функция не создала бесконечную последовательность вызовов самой себя. Вторая серия – это шаги рекурсивного выхода до тех пор, пока выбранный параметр не достигнет конечного значения.

Рассмотрим примеры реализации рекурсий.

Пример реализации прямой рекурсии для вычисления факториала может выглядеть так:

```
  Int64 Factorial(Int64 num)
{
        return num * Factorial (num – 1);
  }
```

Здесь в теле функции Factorial осуществляется вывоз самой функции Factorial. Такая рекурсия будет осуществляться бесконечно, поскольку отсутствует условие выхода из рекурсии.

Из этого примера видно, что очевидная опасность рекурсии заключается в бесконечной рекурсии. Если неправильно построить алгоритм, то функция может пропустить условие остановки рекурсии и выполняться бесконечно.

Для нахождения условия остановки необходимо выделить базу рекурсии. Выделение базы рекурсии предполагает нахождение в решаемой задаче тривиальных случаев, результат для которых очевиден и не требует проведения расчетов. Верно найденная база рекурсии обеспечивает завершенность рекурсивных обращений, которые в конечном итоге сводятся к базовому случаю.

Функция также может вызывать себя бесконечно, если условие остановки не прекращает все возможные пути рекурсии. В следующей ошибочной версии функции Factorial2, функция будет бесконечно вызывать себя, если входное значение — не целое число, или если оно меньше 0.

```
Int64 Factorial2 (Int64 num)
```

```
{
  Int64 fact2;
  if (num == 0)
        fact2 = 1;
  else
        fact2 = num * Factorial2 (num-1);
  return fact2;
}
```

Для устранения бесконечной рекурсии необходимо сделать следующее. Вначале функция Factorial2 должна проверять, что число меньше или равно 0. Факториал для чисел меньше нуля не определен, но это условие проверяется для подстраховки. Если бы функция проверяла только условие равенства числа нулю, то для отрицательных чисел рекурсия была бы бесконечной. Если входное значение меньше или равно 0, функция возвращает значение 1. В остальных случаях, значение функции равно произведению входного значения на факториал от входного значения, уменьшенного на единицу. То, что эта рекурсивная функция, в конце концов, остановится, гарантируется двумя фактами. Во-первых, при каждом последующем вызове, значение параметра num уменьшается на единицу. Во-вторых, когда num становится меньше либо равным 0, функция останавливает рекурсию.

Хотя рекурсия и может упростить понимание некоторых проблем, люди обычно не мыслят рекурсивно. Они обычно стремятся разбить сложные задачи на задачи меньшего объема, которые могут быть выполнены последовательно одна за другой до полного завершения. Для того чтобы думать рекурсивно, нужно разбить задачу на подзадачи, которые затем можно разбить на подзадачи меньшего размера. В какой-то момент подзадачи становятся настолько простыми, что могут быть выполнены непосредственно. Когда завершится выполнение подзадач, большие подзадачи, которые из них составлены, также будут выполнены. Исходная задача окажется выполненной, когда будут все выполнены образующие ее подзадачи.

Рассмотрим примеры решения задач с использованием рекурсивных функций.

Пример 1. Определить является ли строка палиндромом с использованием рекурсии.

Строка является палиндромом, если первый символ строки равен последнему символу строки и палиндромом является строка без первого и последнего символа. Такое определение палиндрома является рекурсивным.

Окончание рекурсии гарантируется уменьшением длины строки на 2 символа на каждой рекурсии и выходом из рекурсии при уменьшении строки до 0 символов.

Причем при очередном обращении к рекурсивной функции строка будет короче на 2 символа.

Когда получен результат по подстроке, можно определить является ли вся строка палиндромом: для этого необходимо, чтобы подстрока была палиндромом и первый и последний символы строки были равны друг другу.

Равенство первого и последнего символов строки определяется выражением s[0] == s[s.Length - 1].

Подстрока без первой и последней букв определяется выражением

s.Substring(1, s.Length – 2);

Рекурсивная функция возвращает результат в виде логической величины f равной true – если подстрока является палиндромом и false – в противном случае.

```
private void button1_Click(object sender, EventArgs e)
{
string s = "aceeca";
bool b = palindrom(s);
Text = b.ToString();
}

private bool palindrom(string s)
{
if (s.Length == 0) return true;
bool f = palindrom(s.Substring(1, s.Length - 2));
return f & (s[0] == s[s.Length - 1]);
}
```

В выше приведенном примере имеется ошибка. Пример не учитывает тот факт, что строка может содержать нечетное количество символов и ее длина не станет равной 0.

Пример 2. Возвести число *a* в степень *n* с использованием рекурсии.

На каждом шаге рекурсии будем уменьшать степень в 2 раза, то есть *a* в степени *n* запишем в виде $a^n = a^{n/2} \cdot a^{n/2}$. Однако для нечетного *n* $a^n = a \cdot a^{(n-1)/2} \cdot a^{(n-1)/2}$. Таким образом, степень будет уменьшаться на каждом шаге в 2 раза и при уменьшении степени до 1 задача становится тривиальной.

Программа будет иметь следующий вид

```
private void button1_Click(object sender, EventArgs e)
{
int a = 3;
int n = 33;
Int64 b = V_Stepen(a, n);
textBox1.Text = b.ToString();
}

private Int64 V_Stepen(int a, int n)
{
if (n==1) return a;
Int64 f;
```

```
if (n%2==0)
f = V_Stepen (a, n / 2) * V_Stepen (a, n / 2);
else
f = a * V_Stepen (a, n / 2) * V_Stepen (a, n / 2);
return f;
}
```

Данный алгоритм является неоптимальным по быстродействию.

Пример 3. Вычислить x^2 с использованием рекуррентной формулы $x^2 = ((x-1)+1)^2 = (x-1)^2 + 2*(x-1)+1$. При этом известно, что $1^2 = 1$, x - целое, положительное.

Поскольку аргумент *x* функции *F* уменьшается на каждом шаге на 1, то в определенный момент будет вызвана функция *F*(1) для которой известен результат равный 1. С этого значения начинаются шаги рекурсивного выхода.

$$F(x) = \begin{cases} F(x-1) + 2x - 1, \text{если } x > 1 \\ x, \text{если } x <= 1 \end{cases}$$

Таким образом функция *F*(*x*) будет иметь вид:

```
int F(int x)
{
  if(x>1) return F(x-1)+2*x+1;
  else return x;
}
```

Вызов функции, размещенный, например, в обработчике события Click для кнопки будет иметь вид:

```
int x=int.Parse(textBox1.Text);
int y = F(x);
label1.Text = y.ToString();
```

Пример 4. Вычислить $F(n) = F(n-2) * F(n-3)$, $n = 0,1,2,....$. Ясно, что по рекуррентной формуле можно осуществлять вычисления, начиная с $n = 3$. Значит для n=0,1,2 должны быть заданы начальные значения: F(0)=0, F(1)=2, F(2)=3, которые позволят начать шаги рекурсивного выхода.

Функция будет иметь вид:

```
int F(int n)
{
  if(n<0) throw new ArgumentException();
  if(n==0) return 0;
  else if(n==1) return 2;
  else if(n==2) return 3;
  else return F(n-2)*F(n-3);
}
```

Вызов рекурсивной функции *F*, размещенный, например, в функции buttonClick будет иметь вид:

```
int x=int.Parse(textBox1.Text);
int y = F(x);
label1.Text = y.ToString();
```

ЛЕКЦИЯ №11

3.2 Реализация рекурсивной программы «Ханойские башни»

Ханойские Башни — это головоломка, которую в 1883 г. придумал французский математик Эдуард Люка. Суть головоломки в следующем. Есть три стержня и восемь дисков разных диаметров. Вначале все диски собраны на одном стержне так, что меньшие диски лежат на больших. Люка предлагал переложить все диски с первого стержня на третий, используя второй. При этом следует соблюдать следующее правило: при перекладывании дисков нельзя класть диск поверх диска меньшего радиуса; за один ход можно переносить лишь один диск.

Если бы в пирамиде был только 1 диск, то решение очевидно. Если дисков 2, то переложим сначала меньший диск на второй стержень, затем перенесем второй диск на третий стержень, а за ним первый диск.

При большем количестве дисков для того чтобы перенести самый большой диск, нужно сначала перенести все диски кроме последнего на второй стержень, потом перенести самый большой на третий, после чего останется перенести все остальные диски со второго на третий. Задачу о переносе N-1 диска мы решаем аналогично, только поменяем стержни местами (N-2 диска перенесем со второго на первый, оставшийся диск перенесем со второго на третий и затем все диски с первого на третий). То есть задачу о N-1 дисках сведем к задаче о N-2 дисках, ту в свою очередь к N-3 дискам, и так вплоть до 1 диска. Этот метод легко программируется с помощью рекурсии.

Задача «Перенести *n* дисков со стержня *from* на стержень *to»*

сводится к трем следующим задачам:

«Перенести *n*-1 диск со стерженя *from* на стержень не-*to*» (рекурсия).

«Перенести *n*-й диск со стерженя *from* на стержень *to*» (тривиальная).

«Перенести *n*-1 диск со стерженя не-*to* на стержень *to*» (рекурсия).

То есть задача переноса пирамиды сводится к двум задачам переноса пирамиды, но в каждой пирамиде на один диск меньше, чем в исходной, и одной тривиальной задаче переноса одного диска.

Каждый диск пронумеруем целым числом, соответствующим размеру диска.

Поскольку диски помещаются и извлекаются со стержней по правилу «первый вошел — последний вышел», для хранения номеров дисков на каждом стержне предлагается использовать стек (как соответствующий логике задачи и наиболее простым по реализации). Стеков должно быть фиксированное количество (3 — по количеству стержней). В стеке будут храниться номера дисков.

```
Stack<int>[] h;
```

Для запуска процесса перестановки дисков на форму установим кнопку Button button1.

При нажатии на кнопку button1 создадим стеки, разместим 8 дисков на первом стержне (в 0-м стеке) и запустим процесс их перестановки с первого стержня на третий в количество 8 штук.

```
private void button1_Click(object sender, EventArgs e)
```

```
{
h = new Stack<int>[3];
h[0] = new Stack<int>();
h[1] = new Stack<int>();
h[2] = new Stack<int>();
h[0].Push(8);
h[0].Push(7);
h[0].Push(6);
h[0].Push(5);
h[0].Push(4);
h[0].Push(3);
h[0].Push(2);
h[0].Push(1);
Hanoi(1, 3, 8);
}
```

Функция Hanoi имеет следующий заголовок

```
private void Hanoi(int from, int to, int cou)
```

и служит для перестановки дисков со стержня номер *from* на стержень номер *to* в количестве *cou* штук. Данная задача является сложной, и поэтому будем ее упрощать путем уменьшения количества переставляемых дисков на один.

Перестановка же одного диска со стержня *from* на стержень *to* является элементарной операцией, которая выполняется командой

```
h[to - 1].Push(h[from - 1].Pop());
```

Таким образом функция Hanoi будет иметь вид:

```
private void Hanoi(int from, int to, int c)
{
int to2 = 6 - from - to; //to2 – номер промежуточной башни
if (c > 1)
{
Hanoi(from, to2, c – 1);
Hanoi(from, to, 1);
Hanoi(to2, to, c – 1);
}
else
{
h[to - 1].Push(h[from - 1].Pop());
}
}
```

ЛЕКЦИЯ №12

3.3. Алгоритмы поиска

Поиск элемента массива на основе линейного просмотра

Результатом работы алгоритма линейного поиска значения Val в массиве A являются индекс Pos и логическая переменная ResultOK, которая принимает значение TRUE, если такой элемент содержится в массиве A, и FALSE – в противном случае. Индекс Pos принимает значение, равное номеру искомого элемента, если такой найден, и значение, равное – 1 – в противном случае.

Алгоритм линейного поиска:

Шаг 1. Полагается Pos:=-1 и ResultOK:=FALSE, и значение переменной цикла J:=0.

Шаг 2. Если A[J]=Val, то переменным Pos и ResultOK присваиваются соотвественно значения Pos:=J, ResultOK:=TRUE и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу J:=J+1.

Шаг 3. Если J<Last, где Last – число элементов массива A, то выполняется Шаг 2, в противном случае – работа алгоритма завершена.

Конец алгоритма.

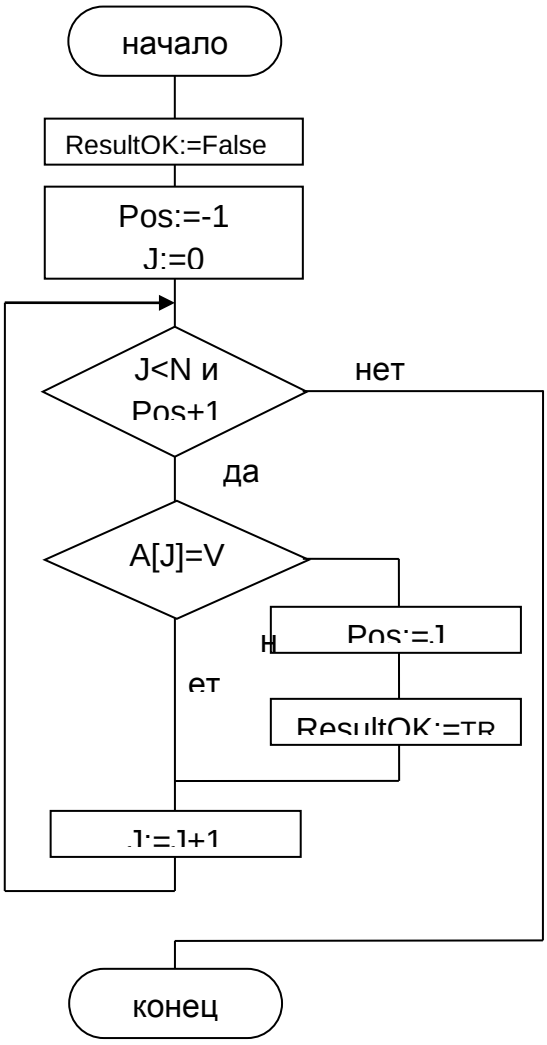


Рис. 3.1. Схема алгоритма линейного поиска

Метод двоичного поиска

Результатом работы алгоритма являектся индекс Pos, показывающий на место в упорядоченном массиве A с номервми элементов от First до Last, на которое необходимо поставить значение Val так, чтобы вновь образованный массив остался упорядоченным. Формируется в качестве результата и логическая переменная ResultOK, которая принимает значение TRUE, если искомый элемент содержится в массиве A, и – FALSE – в противном случае.

Алгоритм двоичного поиска:

Шаг 1. Пока справедливо условие First<Last, выполняется Шаг 2.

Шаг 2. Вычисляется середина массива Middle:=(Last+First) div 2. Если Val равно A[Middle], то пологается First:=Middle+1, в противном случае полагается Last:=Middle-1. После чего управление передается на Шаг 1.

Шаг 3. Полагатся Pos:=First.

Шаг 4. Проверка успеха поиска элемента Val в массиве. Полагается ResultOk:=FALSE. После чего проверяется, содержится ли элемент со знанием Val в массиве, и при положительном ответе на этот вопрос переменной ResultOk присваивается значение TRUE.

Конец алгоритма.

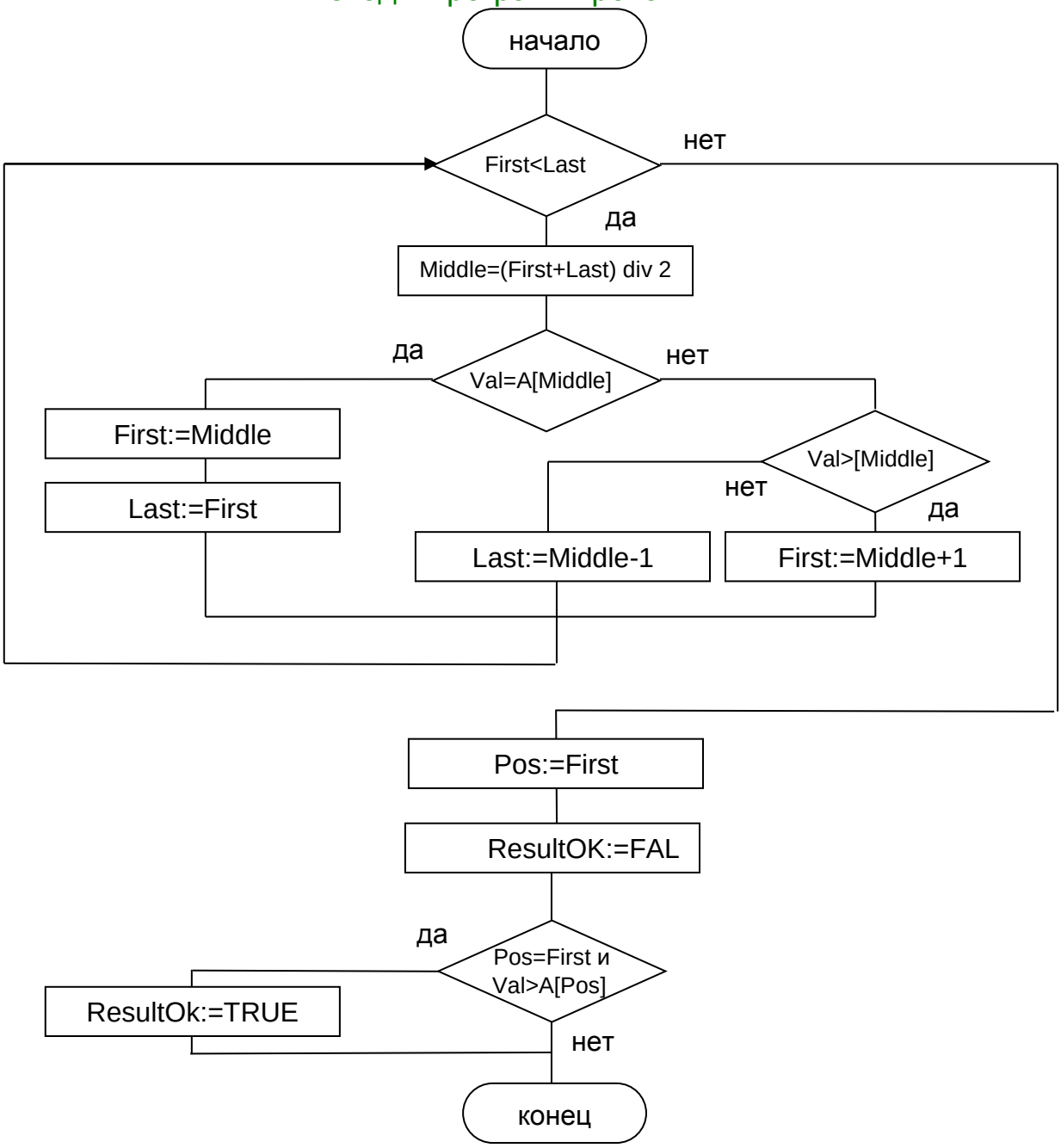


Рис. 3.2. Алгоритм двоичного поиска

ЛЕКЦИЯ №13

3.4. Алгоритмы сортировки

Часто нужно упорядочить предметы по какому-то признаку: записать данные числа в порядке возрастания, слова — по алфавиту, людей выстроить по росту. Если можно сравнить любые два предмета из данного набора, то этот набор всегда можно упорядочить. Процесс упорядочивания информации и называют «сортировкой».

Пусть нам требуется упорядочить набор $\{3, 6, 1, 2, 9, 5\}$ по возрастанию. Это лёгкая задача, ответ будет таким: $\{1, 2, 3, 5, 6, 9\}$.

Пусть есть последовательность a_0, a_1... a_n и функция сравнения, которая на любых двух элементах последовательности принимает одно из трех значений: меньше, больше или равно. Задача сортировки состоит в перестановке членов последовательности таким образом, чтобы выполнялось условие: $a_i <= a_{i+1}$, для всех i от 0 до n.

Возможна ситуация, когда элементы состоят из нескольких полей:

```
struct element {
  field x;
  field y;
}
```

Если значение функции сравнения зависит только от поля x, то x называют ключом, по которому производится сортировка. На практике, в качестве x часто выступает число, а поле y хранит какие-либо данные, никак не влияющие на работу алгоритма.

Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. Сортировка — процесс перестановки элементов последовательности в определенном порядке: по возрастанию, убыванию, последней цифре, сумме делителей, и т.д.

Время *(C(n))* — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для типичного алгоритма хорошее поведение — это O(N*logN) и плохое поведение — это $O(N^2)$. Идеальное поведение для упорядочения — O(N). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в N*logN сравнениях.

Память *(M(n))* — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(logN)памяти.

Классификация алгоритмов сортировки

Устойчивость — устойчивая сортировка не меняет взаимного расположения равных элементов.

Естественность поведения — эффективность метода при обработке уже упорядоченных, или частично упорядоченных данных. Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость *худшего случая* для этих алгоритмов составляет O(N*logN), но они

отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является его сфера применения. Здесь основных типов упорядочения два:

Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат. В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.

Внешняя сортировка оперирует с запоминающими устройствами большого объёма, но с доступом не произвольным, а последовательным (упорядочение файлов), т. е. в данный момент мы 'видим' только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

1. Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим.

2. Объем данных не позволяет им разместиться в ОЗУ.

Метод сортировки выбором

Исходный массив длиной N разбивается на две части: итог и остаток. Участок массива, называемый итогом, располагается с начала массива и должен быть упорядоченным, а участок массива, называемый остатком, располагается вплотную за итогом и содержит исходные числа не отсортированной части исходного массива. Пусть первый элемент остатка является J-ый элементом массива.

Алгоритм сортировки выбором:

Шаг 1. Полагается J:=0, т.е. считается, что итоговый участок – пуст.

Шаг 2. В остатке массива ищется минимальный и меняется место с первым элементом остатка (J-ым элементом массива).

Шаг 3. Если J<N-1, то повторяется Шаг 2. В противном случае – конец алгоритма, т.к. итог становится равный всему массиву.

Конец алгоритма.

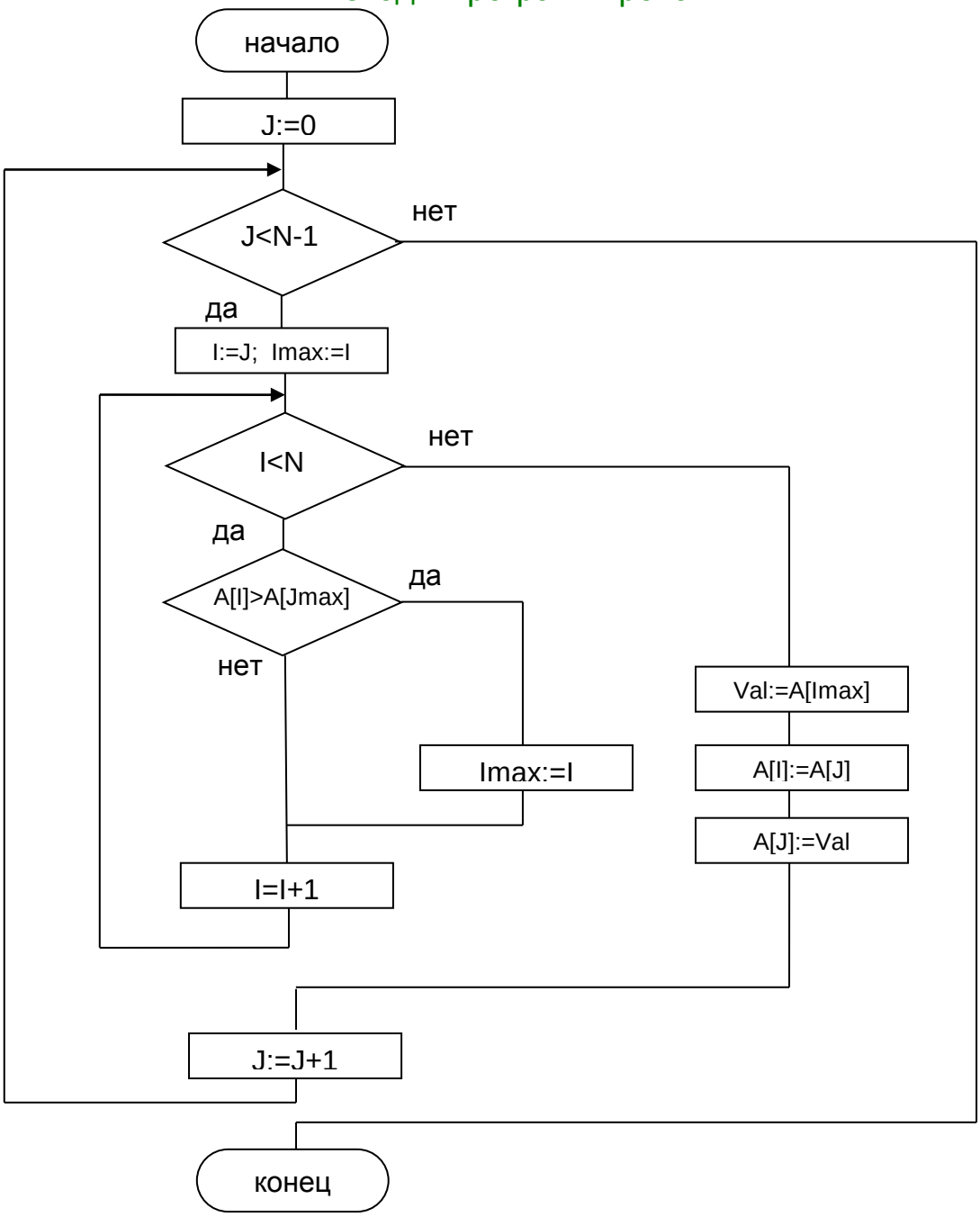


Рис. 3.3. Алгоритм сортировки выбором

Метод сортировки пузырьком

Аналогично, как и в методе выбора, исходный массив длиной N разбивается на две части: отсортированную (итог) и не отсортированную (остаток). Пусть первый элемент остатка будет J-ым элементом массива.

Алгоритм сортировки пузырьком:

Шаг 1. Пусть J:=1, т.е. итоговый участок состоит из одного элемента.

Шаг 2. Берется первый элемент остатка и перемещается на место в итоговый участок массива так, чтобы итог остался упорядоченным. Первый элемент остатка назовем перемещаемым. Перемещение выполняется путем сравнения перемещаемого элемента с предшествующим ему элементом. Если

предшествующий элемент меньше сравниваемого элемента, то процесс перемещения закончен. В противном случае сравниваемые элементы переставляются и, если элемент не достиг начала массива, то повторяется Шаг 2.

Шаг 3. После того, как первый элемент остатка переместился в итоговый участок, увеличивается на единицу значение переменной J, тем самым увеличивая отсортированную часть массива. Если J<N, то управление передается на Шаг 2, в противном случае – работа алгоритма завершена.

Конец алгоритма.

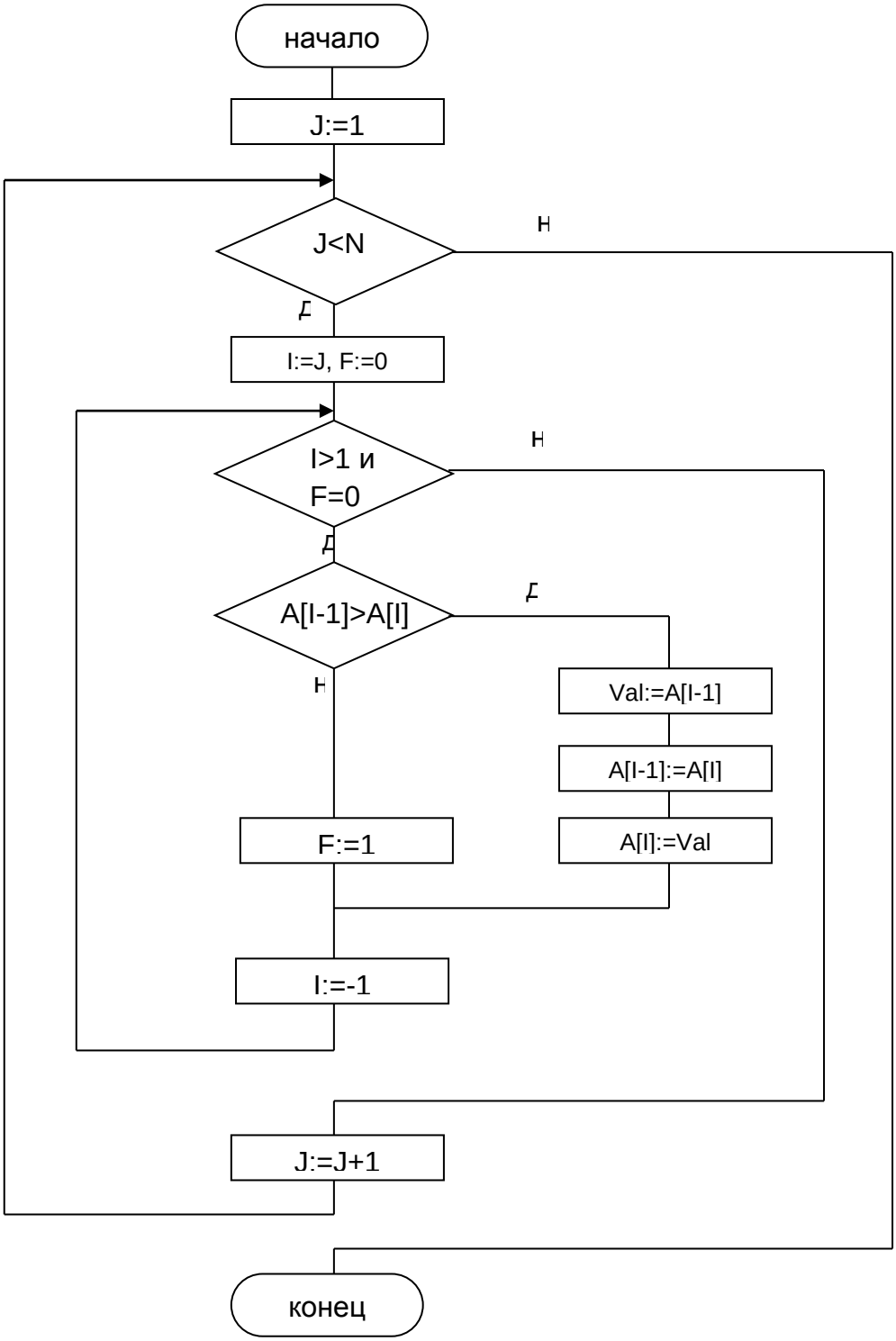


Рис. 3.4. Алгоритм сортировки пузырьком

Метод сортировки включением

Этот метод похож на метод пузырька. Происходит такое же разбиение массива на отсортированную и не отсортированную части, но перемещение первого элемента остатка на принадлежащее ему место в итоге делается не сравнением двух соседних элементов, а с помощью метода двоичного поиска, который удобно оформить в виде отдельной процедуры.

Алгоритм метода включения:

Шаг 1. Пусть J=1, т.е. итоговый участок состоит из одного элемента.

Шаг 2. Берется первый элемент остатка и перемещается в отсортированную часть массива так, чтобы итоговый участок остался упорядоченным. Делается это с помощью обращения к процедуре двоичного поиска, которая в качестве выходного параметра дает номер элемента массива, на месте которого должен бы находится перемещаемый элемент. Если этот номер указывает на место в итоговом участке массива, то сдвигаются все элементы итогового участка массива, начиная с этого номера на одно место вправо, а перемещаемый элемент ставится на освободившееся место.

Шаг 3. После того, как первый элемент остатка переместился в итоговый участок, увеличивается на единицу значение переменной J, тем самым увеличивая отсортированную часть массива. Если J<N, то управление передается на Шаг 2, в противном случае – работа алгоритма завершена.

Конец алгоритма.

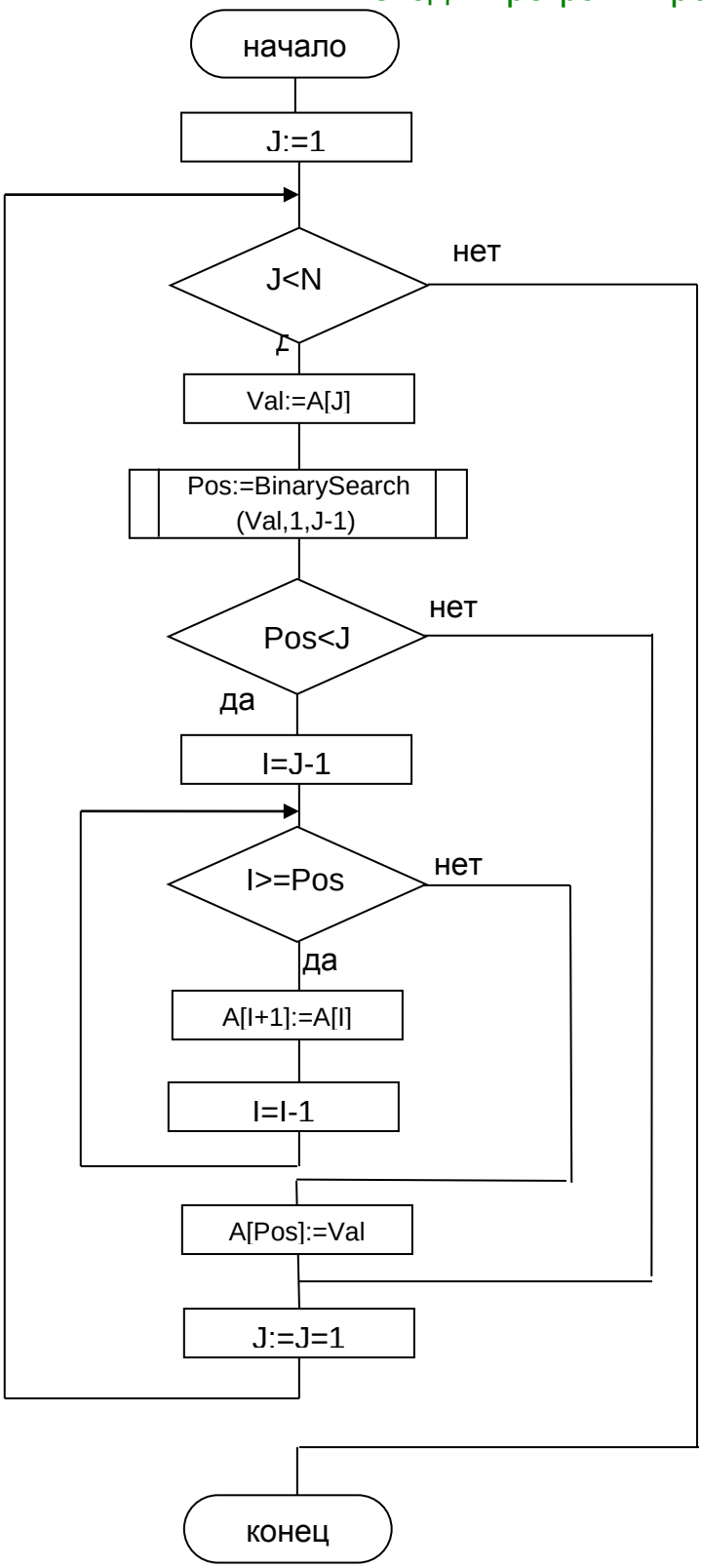


Рис.3.5. Алгоритм сортировки включением

Метод быстрой сортировки

Исходным является массив A с номерами элементов от First до Last. В алгоритме используются еще два индекса массива, обозначенные как Index и ContrIndex. Первый из них всегда указывает на переставляемый элемент, а

второй – на элемент, который сравнивается по значению с переставляемым. В процессе вычислений применяются переменная h (равная либо 1, либо -1) – шаг движения индексов навстречу друг другу, используемая для обозначения направления движения индекса ContrIndex, и логическая переменная Condition (равная либо TRUE, либо FALSE), используемая для изменения условия сравнения на противоположное при обратном движении индекса ContrIndex.

Алгоритм быстрой сортировки:

Шаг 1. Если First>=Last, то происходит выход из алгоритма. В противном случае полагается h:=1, Condition:=TRUE, Index:=First, ContrIndex:=Last и делаются шаги: Шаг 2 – Шаг 3.

Шаг 2. Пока Index не равно ContrIndex, делаются шаги: Шаг 2a – Шаг 2b.

Шаг2a. Если справедливо ((A[Index]>A[ContrIndex])=Condition), то переставляются как сами элементы, на которые указывают Index и ContrIndex, (Val:=A[Index], A[Index]:=A[ContrIndex], A[ContrIndex]:=Val), так и сами вспомогательные индексы массивов (Val:=Index, Index:=ContrIndex, ConterIndex:=Val). Затем меняется направление движения (h:=-h) и условие сравнения (Condition: = not Condition). В процессе таких перестановок слева от переставляемого элемента всегда будут находится меньшие значения, а справа – большие значения.

Шаг2b. Сдвигается индекс массива Index навстречу инвексу ContrIndex, т.е. Index:=Index+h.

Шаг3. Перед выполнением этого шага индексы Index = ContrIndex и элемент A[Index] находится на нужном месте. т.е. исходный массив разбит на три части: часть массива до этого элемента, значения в котором меньше величины A[Index], часть массива до этого элемента с значениями большими значения A[Index] и сам этот элемент A[Index]. Поэтому для дальнейшего упорядочивания массива достаточно рекурсивно обратиться к алгоритму быстрой сортировки два раза: для первой и второй частей массива. т.к длина сортируемых участков массива уменьшается, то в итоге алгоритм конечен и после применения алгоритма массив будет полностью отсортирован.

Конец алгоритма.

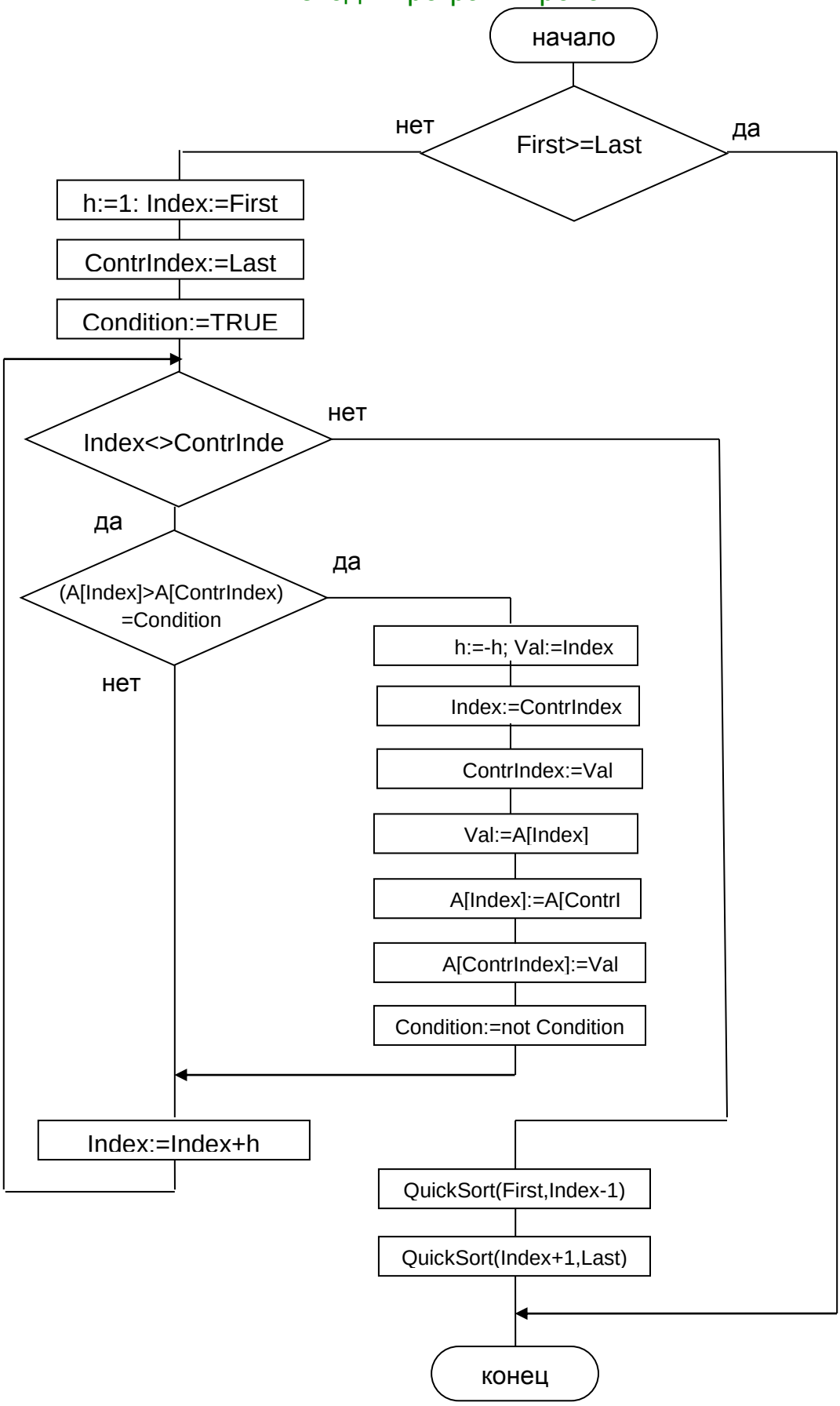


Рис. 3.6. Алгоритм быстрой сортировки

Сравнение быстрой сортировки QuickSort и сортировки пузырьком. Количество перестановок (массивы заполнены случайными данными):

	QuickSort	«пузырек»
10	11	24
100	184	2263
200	426	9055
500	1346	63529
1000	3074	248547

Тематика лабораторных работ по дисциплине «Методы программирования»

1. Организация хранения объектов в списке на языке C# с возможностью редактирования
2. Интерфейсы и абстрактные классы
3. Работа с коллекциями библиотеки .NET
4. Использование технологии LINQ для работы с коллекцией чисел
5. Использование технологии LINQ для работы с коллекцией экземпляров класса
6. Технология обработки исключений на языке C#
7. Делегаты и события
8. Технология COM и события
9. Регулярные выражения
10. Рекурсивные алгоритмы
11. Реализация рекурсивных алгоритмов на примере головоломки «Ханойские башни»
12. Реализация графического вывода для программы «Ханойские башни»
13. Алгоритмы поиска
14. Алгоритмы сортировки