Новосибирский государственный университет
Высший колледж информатики

Иванчева Н.А., Иваньчева Т.А.

Постреляционная СУБД Caché

(методическое пособие)

Новосибирск 2004

Наряду с реляционным подходом к разработке информационных приложений для баз данных все большее распространение получает объектный подход. Это связано с ограниченностью самой реляционной модели, например, при описании и манипулировании сложными структурами данных, усложнением запросов при выборках данных из нескольких таблиц, снижением производительности при соединениях больших таблиц и др.

Объектный подход дает такие преимущества как:

- естественное представление данных;
- возможность разработки структур любого уровня сложности;
- высокую производительность процесса разработки;
- использование объектно-ориентированных CASE-средств проектирования и разработки приложений.

В последнее время появляется все больше объектных и объектно-ориентированных СУБД таких как Versant, Jasmine, ODB-Jupiter, Caché и др., которые приобретают все большую популярность и признание. Некоторые эксперты полагают, что, несмотря на доминирование на рынке реляционных СУБД, за объектными СУБД будущее.

Высокоэффективная постреляционная система управления базами данных Caché разработана в русле новых технологий, объединяющих сервер многомерных данных и многофункциональный сервер приложений. Главные свойства Caché – развитая объектная технология, быстрота разработки Web-приложений, усовершенствованная база SQL и уникальная технология получения данных – позволяют достичь такого высокого уровня производительности и масштабируемости, который был недоступен в рамках реляционной технологии.

В виду недостатка учебных пособий по постреляционной СУБД Caché, считаем, что издание данного пособия будет весьма полезным и своевременным. При разработке пособия авторы внимательно изучили и использовали всю доступную литературу по СУБД Caché. Пособие не претендует на полноту, тем не менее, авторы постарались отразить основные моменты, связанные с использованием инструментария, который предлагает СУБД Cache.

Пособие рассчитано на широкий круг учащихся специализированных колледжей и ВУЗов, специалистов в области разработки приложений для баз данных, а также всех интересующихся новыми технологиями в области объектной разработки приложений для баз данных.

Авторы выражают свою искреннюю благодарность преподавателям ВКИ НГУ Остапчуку В.В. и Шину К.Ю., зав. кафедрой Информатики ВКИ НГУ Куликову А.И., менеджеру Московского представительства компании InterSystems Сиротюку О.В., а также рецензенту пособия к.т.н. Загорулько Ю.А.

Рецензент:
к т. н. Загорулько Ю.А.

Оглавление

Глава 1. Объектный и реляционный подход

1.1. Сравнение объектного и реляционного подхода

Реляционная модель элегантна, проста, имеет прочное математическое обоснование, простой и ясный язык запросов. Но эта простота имеет и свою оборотную сторону.

При разработке современных информационных систем требуется обеспечить:

- хранение и манипулирование объектами сложной структуры;
- наличие сложных взаимосвязей между объектами;
- хранение и манипулирование разнородными объектами;
- высокую производительность.

Для выполнения вышеперечисленного реляционная модель не всегда является удобной, по следующим причинам:

1. реляционная модель работает с нормализованными отношениями, т.е. отношениями, уже находящимися в первой нормальной форме, а это означает, что реляционная модель структуру значения атрибута не рассматривает. При описании многих предметных областей, напротив, требуется наличие вложенных структур, когда значение атрибута может быть списком или другим объектом сложной структуры. При этом должен поставляться инструмент для обработки таких структур. Это приводит к возникновению проблем при разработке структур данных, т.к. плоские таблицы плохо подходят для описания сложных предметных областей.
2. современные информационные системы требуют использования разнородных данных, например, фотографий, видеоизображений, звука. Но язык запросов SQL обеспечивает возможность только их хранения, но не обработки.
3. процесс нормализации требует разнесения данных по нескольким таблицам для исключения избыточности. Выполнение запросов часто требует, напротив, соединения (join) нескольких таблиц, что порождает две проблемы. Во-первых, усложнение запросов при соединении нескольких таблиц. Во-вторых, в случае объемных таблиц данная операция требует больших ресурсов и снижает производительность системы.

Альтернативой реляционному подходу является объектный подход.

Объектная технология и объектные базы данных являются практическим результатом работы по моделированию деятельности мозга. Замечено, что мозг может хранить очень сложную и разнородную информацию и манипулировать ею, используя единый принцип. Сложное поведение объектов реального мира также должно быть реализовано в программных продуктах по единому принципу, чтобы скрыть эту сложность.

Объектный подход к проектированию систем позволяет наиболее естественным образом записать объекты в базу данных «как есть», обеспечить хранение объектов и манипулирование ими.

Объектно-ориентированные базы данных (ООБД) по сравнению с традиционными реляционными базами данных дают следующие преимущества:

- ООБД хранят не только данные, но и методы их обработки, инкапсулированные в одном объекте;
- ООБД позволяют обрабатывать мультимедийные данные;
- ООБД допускают работу на высоком уровне абстракции;
- ООБД позволяют пользователям создавать структуры данных любой сложности.

При всех достоинствах ООБД, переход на такие СУБД затруднен из-за значительного объема разработок, опирающихся на реляционные СУБД. Кроме того, объектная технология, поддерживаемая в ряде постреляционных СУБД, не имеет развитого и стандартизированного языка генерации отчетов и анализа данных, каким является структурированный язык запросов SQL. Данные проблемы были решены при создании постреляционной СУБД Caché от InterSystems (www.intersystems.ru). СУБД Caché не только реализу-

ет основные возможности объектно-ориентированной технологии, но и позволяет во многом облегчить переход с реляционной технологии на объектную, а также может выступать в роли шлюза к реляционным базам данных.

Несмотря на наличие многих теоретических проблем, ключевой из которых является, безусловно, сложность строгой формализации объектной модели данных, многие эксперты полагают, что за этими системами будущее.

1.2. Основные положения стандарта ODMG

Объектно-ориентированные СУБД (ООСУБД) отличает ряд обязательных свойств. Эти свойства продекларированы в «Манифесте систем объектно-ориентированных баз данных», а впоследствии закреплены в документах ODMG (Object Data Management Group), организации, объединяющей ведущих производителей объектно-ориентированных СУБД. Основной целью эта организация ставит задачу выработки единых стандартов ООСУБД, соблюдение которых обеспечило бы переносимость приложений.

Рассмотрим основные положения стандарта ODMG.

1.2.1. Модель данных

В соответствии со стандартом ODMG 2.0 объектная модель данных характеризуется следующими свойствами.

- Базовыми примитивами являются объекты и литералы. Каждый объект имеет уникальный идентификатор, литерал не имеет идентификатора.
- Объекты и литералы различаются по типу. Состояние объекта характеризуется его свойствами, поведение объекта характеризуется его методами (операциями). Все объекты одного типа имеют одинаковое поведение (набор операций) и одинаковый диапазон изменения свойств.
- Свойствами могут быть атрибуты объекта или связи между объектом и одним или несколькими другими объектами.
- Поведение объекта определяется набором операций, которые могут быть выполнены над объектом или самим объектом. Операции могут иметь список входных и выходных параметров строго определенного типа. Каждая операция может также возвращать типизированный результат.
- База данных хранит объекты, позволяя совместно использовать их различным пользователям и приложениям. База данных основана на схеме данных, определяемой языком определения данных, и содержит экземпляры типов, определенных схемой.

Тип также является объектом. Поддерживается иерархия супертипов и подтипов, реализуя тем самым стандартный механизм объектно-ориентированного программирования — наследование. Тип имеет набор свойств, а объект характеризуется состоянием в зависимости от значения каждого свойства. Операции, определяющие поведение типа, едины для всех объектов одного типа. Свойство едино для всего типа, а все объекты типа также имеют одинаковый набор свойств. Значение свойства относится к конкретному объекту. На рис.1 схематично изображены основные элементы ООСУБД.

Рисунок 1. Основные элементы ООСУБД

1.2.2. Идентификатор объекта

Как следует из модели данных, каждый объект в базе данных уникален. Существует несколько подходов для идентификации объекта.

Первый самый простой — объекту присваивается уникальный номер (OID — object identificator) в базе, который никогда больше не повторяется, даже если предыдущий объект с таким номером уже удален. Недостаток такого подхода состоит в невозможности перенести объекты в другую базу без потери связности между ними.

Второй подход состоит в использовании составного идентификатора. Например, в ООСУБД Versant идентификатор OID имеет формат xxxxxxxx:yyyyyyyy, где xxxxxxxx — идентификатор базы данных, yyyyyyyy — идентификатор объекта в базе. Составленный таким образом OID позволяет переносить объекты из базы в базу без потери связи между объектами или без удаления объектов с перекрывающими номерами.

Третий подход, который применяется в ООСУБД Jasmine и Caché, состоит в том, что OID формируется из имени класса объекта и собственно номера объекта. Идентификатор имеет вид classnamexxxxxxxx, где classname — имя класса, а xxxxxxxx — номер объекта этого класса. Недостаток такого подхода заключается в том, что проблема переноса объектов не решена полностью. В разных базах могут оказаться объекты одного класса, а уникальность номеров соблюдается только в пределах одной базы. Преимущество подхода — в простоте извлечения объектов нужного класса: объекты одного класса будут иметь идентификатор, имеющий общую часть.

Четвертый вариант, самый лучший, состоит в использовании OID, состоящего из трех частей: номер базы, номер класса, номер объекта. Однако и при этом остается вопрос о том, как обеспечить уникальность номеров баз и классов на глобальном уровне — при использовании ООСУБД на различных платформах, в разных городах и странах.

1.2.3. Новые типы данных

Объектные базы данных имеют возможность создания и использования новых типов данных. В этом заключается одно из принципиальных отличий их от реляционных баз данных. Создание нового типа не требует модификации ядра базы и основано на принципах объектно-ориентированного программирования: инкапсуляции, наследовании, перегрузке операций и позднем связывании.

В ООСУБД для объектов, которые предполагается хранить в базе (постоянные объекты), требуется, чтобы их предком был конкретный базовый тип, определяющий все основные операции взаимодействия с сервером баз данных. Концептуально объект характеризуют поведение и состояние. Тип объекта определяется через его поведение, т.е. через операции, которые могут быть выполнены объектом.

Поэтому для создания нового типа необходимо унаследовать свойства любого имеющегося типа, наиболее подходящего по своему поведению и состоянию, добавить новые операции и атрибуты и переопределить, по необходимости, уже существующие.

Пример на рис. 2 иллюстрирует возможность наращивания типа «человек» (Person), который может быть «мужчиной» (MalePerson), «женщиной» (FemalePerson), «взрослым» (Adult) или «ребенком» (Child). Соответственно возможны по парные пересечения этих

типов. Каждый из этих типов может иметь свой набор свойств и операций. Любой объект типа MalePerson, FemalePerson, Adult или Child является объектом типа Person. Аналогично объект подтипа MaleAdult, полученного наследованием типов MalePerson и Adult, является человеком, мужского пола, взрослым и, соответственно, может пользоваться свойствами и операциями всех своих супертипов.

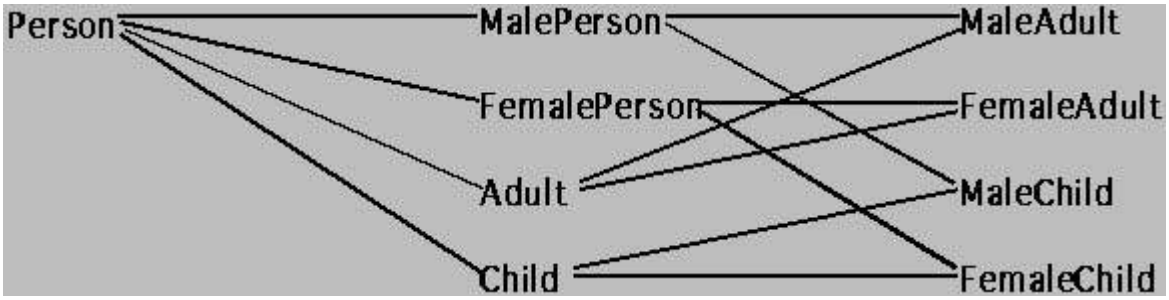


Рисунок 2. Пример наследования типов

1.2.4. Оптимизация ядра СУБД

В отличие от объектно-реляционных СУБД, где ядро остается реляционным, а «объектность» реализуется в виде специальной надстройки, ядро ООСУБД оптимизировано для операций с объектами и производит операции кэширования объектов, ведение версий объектов, разделение прав доступа к конкретным объектам.

Поэтому ООСУБД имеет более высокое быстродействие на операциях, требующих доступа и получения данных, упакованных в объекты, по сравнению с реляционными СУБД, для которых необходимость выборки связных данных ведет к выполнению дополнительных внутренних операций.

1.2.5. Язык СУБД и запросы

Варианты языков запросов можно разделить на две группы. Первая объединяет языки, унаследованные от SQL и представляющие собой разновидность OQL (Object Query Language), языка, стандартизованного ODMG. Объектно-реляционные СУБД используют различные варианты ограниченных объектных расширений SQL.

Вторая, сравнительно новая (применяется с 1998 года) группа языков запросов базируется на XML. Собирательное название языков этой группы — XML QL (или XQL). Они могут применяться в качестве языков запросов в объектных и объектно-реляционных базах данных. Использование в чисто реляционных базах не целесообразно, поскольку языком предусматривается объектная структура запроса.

1.2.6. Физические хранилища

Как правило, база данных объединяет несколько хранилищ данных, каждое из которых ассоциируется с одним или несколькими файлами. Хранятся как метаданные (класс, атрибут, определения операций), так и пользовательские данные. Выделяют три типа хранилищ:

- **системное хранилище**, используется для хранения системы классов, создается на этапе формирования базы данных и содержит информацию о классах, о наличии и месторасположении пользовательских хранилищ;
- **пользовательское хранилище** предназначено для хранения пользовательских объектов;
- **служебное хранилище** содержит временную информацию, например сведения о заблокированных объектах, об активных транзакциях, различного вида списки запросов пользователей и т.д.

1.3. Постреляционная СУБД Caché

Высокопроизводительная постреляционная СУБД Caché появилась в 1997 году, фирмой разработчиком является компания InterSystems. В настоящее время вышла новая версия Caché 5.0.

Компания основана в 1978 году. По данным независимого исследования, проведенного компанией IDC, InterSystems входит в десятку ведущих поставщиков СУБД и в сотню ведущих независимых разработчиков программного обеспечения. В числе клиентов компании – крупнейшие финансовые компании, такие как Chase Manhattan, Lloyd Bank, большинство крупнейших госпиталей Америки, телекоммуникационные и промышленные компании. В России продукты InterSystems получили наибольшее распространение в телекоммуникационной и банковской сферах.

К достоинствам СУБД Caché можно отнести следующие:

- Независимость хранения данных от способа представления, одни и те же данные можно рассматривать либо как объекты, либо как таблицы, либо как глобалы. Это достигается с помощью единой архитектуры СУБД Caché.
- Высокая производительность обеспечивается благодаря использованию многомерной модели данных и методов разреженного хранения данных, при этом доступ и редактирование требуют гораздо меньше операций ввода-вывода, чем в случае реляционных таблиц, а это означает, что приложение будет работать быстрее.
- Масштабируемость. Транзакционная модель ядра Caché позволяет легко наращивать число клиентов (до многих тысяч), сохраняя высокую производительность. Транзакции получают необходимые им данные без выполнения сложных соединений таблиц или «прыжков» из таблицы в таблицу.
- Высокая продуктивность процесса программирования обеспечивается предоставлением пользователю объектной технологии построения приложений, мощного инструмента повышения продуктивности программирования. Даже объекты самой сложной структуры предстают в простом и естественном виде, ускоряя тем самым процесс разработки приложений.
- Сокращение затрат. В отличие от реляционных, приложения на основе Caché прекрасно обходятся гораздо менее мощным оборудованием и не требуют администраторов баз данных. Управление и эксплуатация систем отличается простотой.
- Высокопроизводительный SQL. Существующие реляционные приложения получают значительный выигрыш в производительности даже без изменения кода, используя Caché SQL для доступа к производительной постреляционной базе данных Caché.
- Оригинальная технология CSP для быстрой разработки Web-приложений.

1.3.1. Основные компоненты СУБД Caché

Основные компоненты СУБД Caché:

- Многомерное ядро системы, ориентированное на работу с транзакциями, позволяет реализовать развитую технологию обработки транзакций и разрешения конфликтов. Блокировка данных производится на логическом уровне. Это позволяет учитывать особенность многих транзакций, производящих изменения небольшого объема информации. Кроме этого, в Caché реализованы атомарные операции добавления и удаления без проведения блокировки, в частности, это применяется для счетчика идентификаторов объектов.
- Сервер Caché Objects. Представление многомерных структур данных ядра системы в виде объектов, инкапсулирующих как данные, так и методы их обработки. Позволяет реализовать объектный доступ к данным.

- Сервер Caché Objects. Представление многомерных структур данных ядра системы в виде объектов, инкапсулирующих как данные, так и методы их обработки. Позволяет реализовать объектный доступ к данным.
- Сервер Caché SQL. Представление многомерных структур данных в виде реляционных таблиц. Позволяет реализовать реляционный доступ к данным.
- Сервер прямого доступа (Caché Direct). Предоставление прямого чрезвычайно эффективного доступа к многомерным структурам данных ядра системы.

На рис. 3. представлена архитектура Caché.

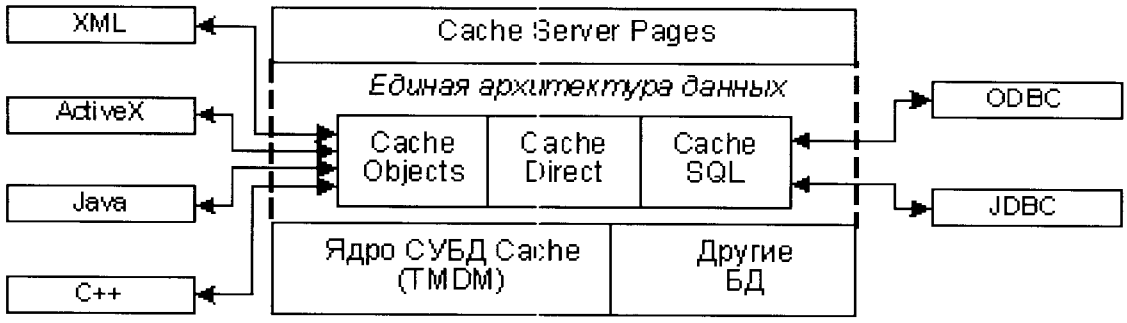


Рисунок 3. Архитектура системы Caché

Глава 2. Среда разработки Caché

Для разработки приложений и работы с базами данных система Caché предлагает следующий набор утилит:

- Редактор конфигурации
- Studio
- Terminal
- Проводник
- Панель управления
- SQL-менеджер

Все утилиты запускаются из Caché-куба, который располагается в правой части экрана, щелчком правой кнопки мыши.

Утилита «Редактор конфигурации» предназначена для задания и просмотра следующих параметров конфигурации:

- системной информации
- рабочих областей
- баз данных
- сетевых соединений
- CSP-приложений
- лицензии Caché

Большинство параметров конфигурации выполняются динамически и не требуют перезапуска системы Caché.

«Редактор конфигурации» может использоваться для создания базы данных. База данных Caché это файл с именем Cache.dat. По умолчанию база данных Caché располагается в директории C:\CacheSys\MGR\User. Для создания своей базы данных в другой директории на вкладке «Базы данных» Редактора конфигурации по кнопке «Мастера…» выбирается соответствующий мастер. По умолчанию начальный размер базы равен 1 МБ. На рис. 4 изображено окно «Редактора конфигурации».

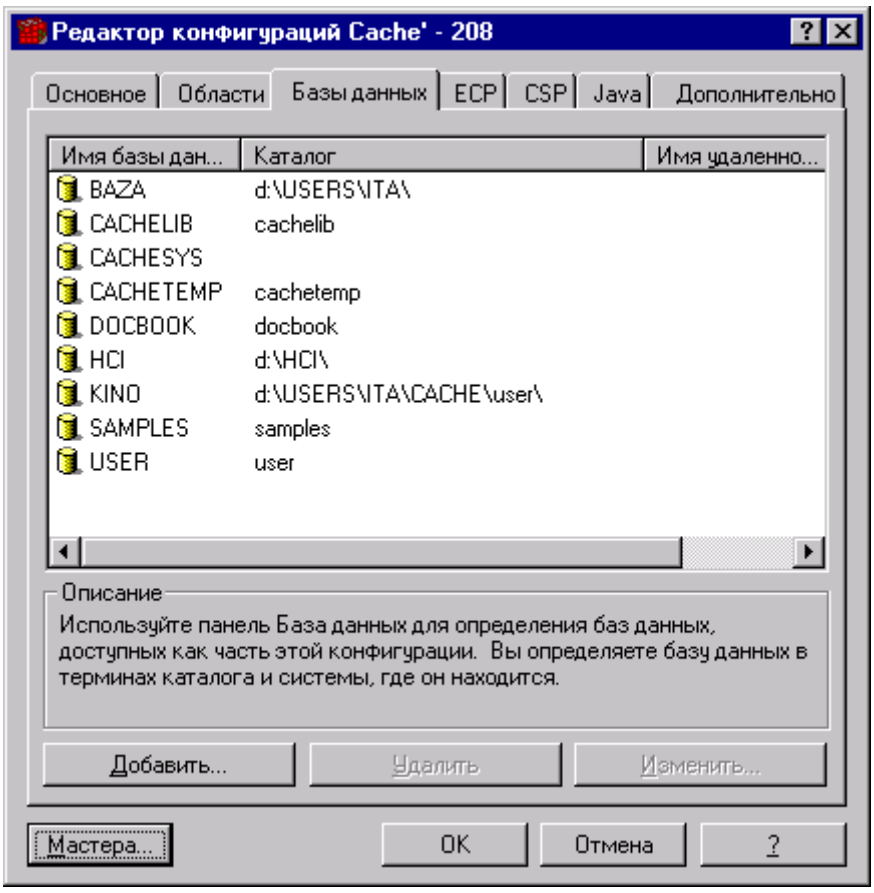


Рисунок 4. Окно «Редактора конфигурации».

На рис. 5 изображено окно выбора мастеров.

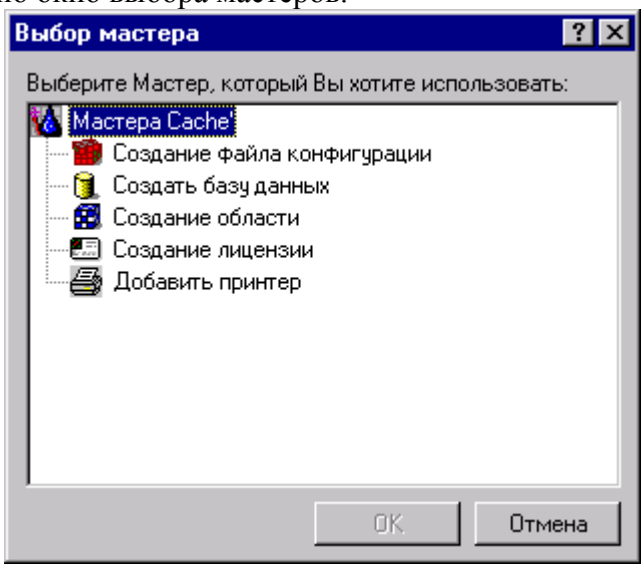


Рисунок 5. Окно выбора мастера

Созданная база данных связывается со своей рабочей областью (namespace), которая так же может быть создана с помощью Редактора конфигурации, для этого необходимо выбрать мастер «Создание области». По умолчанию разработка ведется в рабочей области User.

Caché Studio это интегрированная, визуальная среда создания объектно-ориентированных баз данных и Web-приложений. Она имеет следующие особенности для обеспечения быстрой разработки:

- Возможность редактирования определений классов, CSP-страниц, программ на Caché Object Script(COS) в единой интегрированной среде;
- Полнотекстовое редактирование с подсветкой команд, а также проверкой синтаксиса на следующих языках: Caché Object Script, Basic, Java, SQL, JavaScript, HTML, и XML;
- Поддержка команды разработчиков, работающих с общим репозиторием кода приложения;
- Наличие отладчика;
- Организация кода в проект;
- Наличие мастеров, позволяющих создавать классы, методы, свойства, связи, WEB-формы.

Исходный код приложения Caché оформляется в виде проекта. Проект может содержать следующие компоненты: классы, программы, CSP-файлы. Можно создавать новые проекты, открывать и изменять существующие. Также существует возможность добавления или удаления компонентов из проекта.

Внешний вид среды разработки Caché Studio приведен на рис. 6.

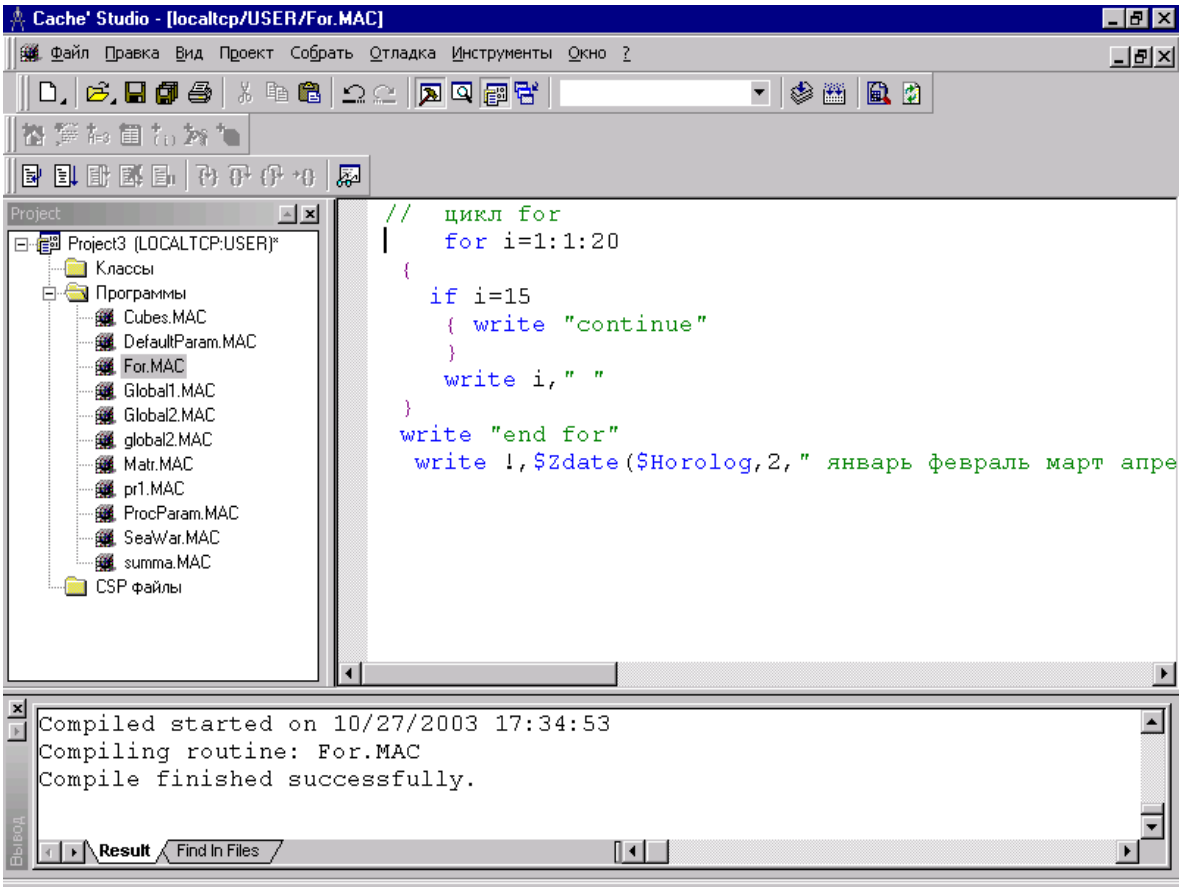


Рисунок 6. Среда разработки Caché Studio.

Утилита Caché Terminal может быть использована для отладки программ, процедур, функций, проверки работы отдельных операторов. Работа ведется в режиме командной строки. Утилита чрезвычайно проста. После приглашения набирается команда и нажимается ENTER. Здесь же выводится результат выполнения команды. Окно утилиты приведено на рис. 7.

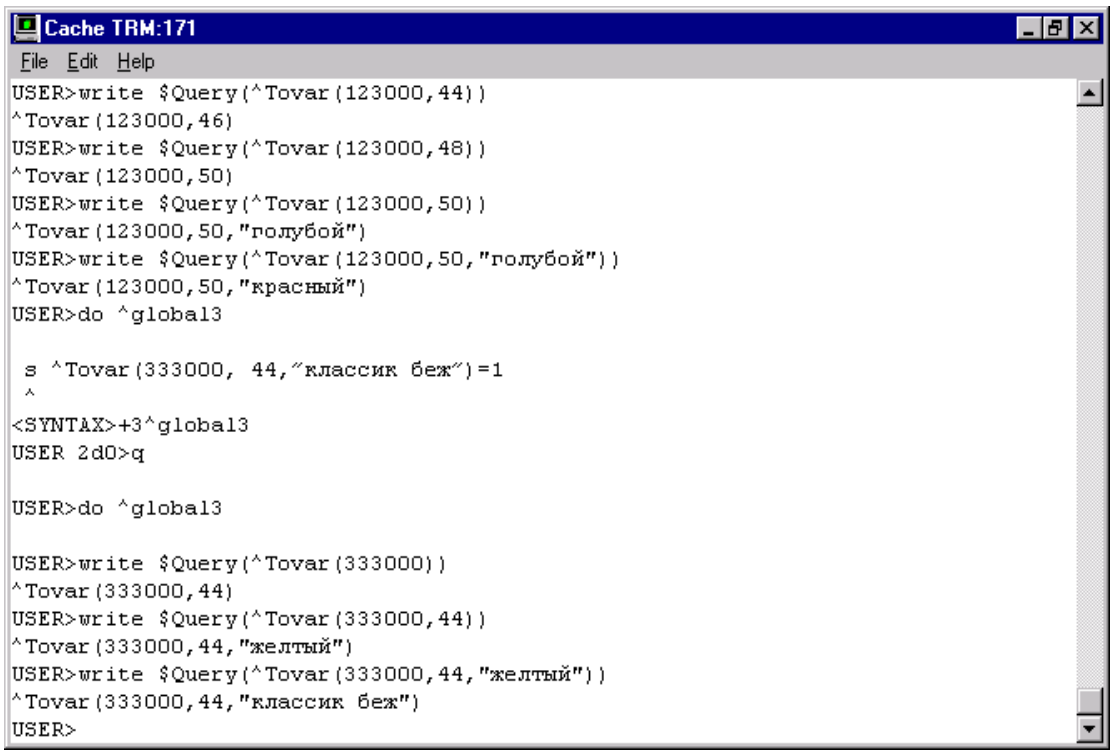


Рисунок 7. Окно утилиты Caché Terminal

Для доступа к многомерным структурам ядра СУБД Caché можно воспользоваться **утилитой Caché Проводник**. Утилита Caché Проводник предоставляет ряд интерфейсов для просмотра, импорта/экспорта, печати на принтер глобалей, классов Caché, программ Caché. Внешний вид утилиты приведен на рис. 8.

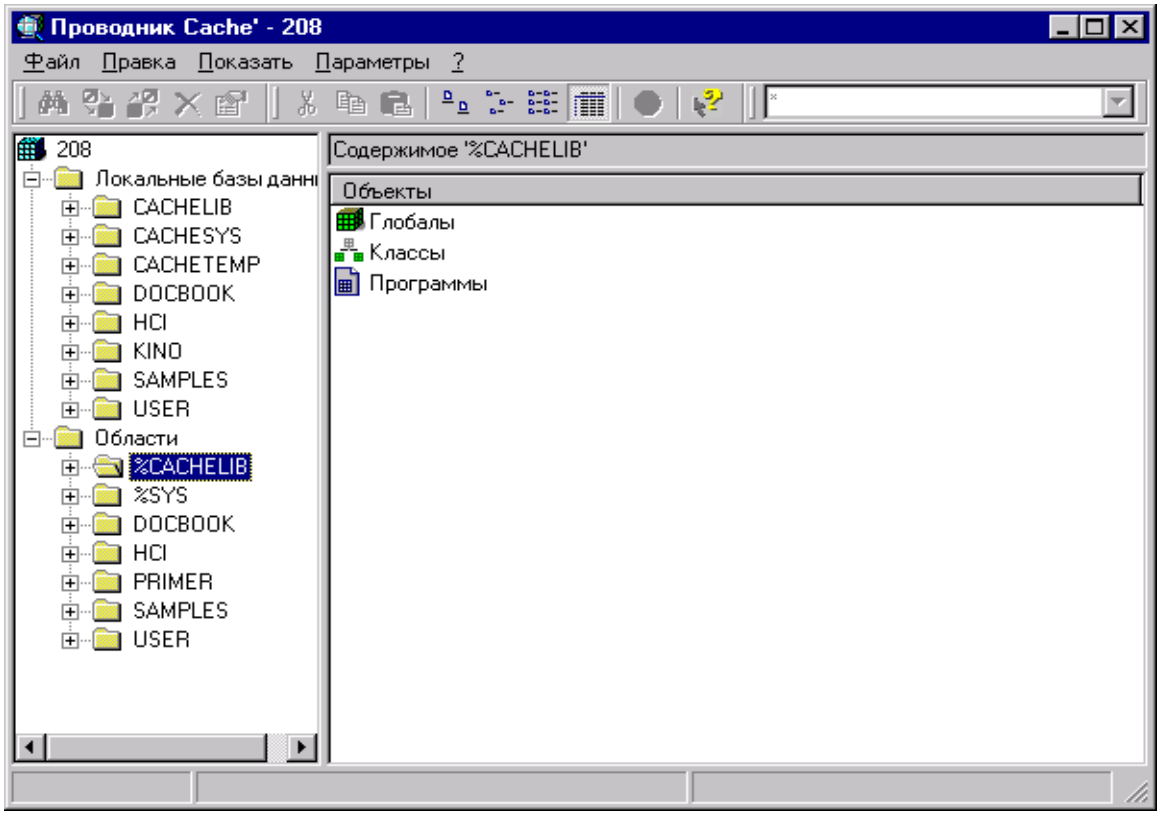


Рисунок 8. Утилита Caché Проводник.

Утилита SQL-менеджер позволяет работать с объектами как с реляционными таблицами, это типичный проводник для работы со схемой реляционной базы данных. Окно утилиты приведено на рис. 9.

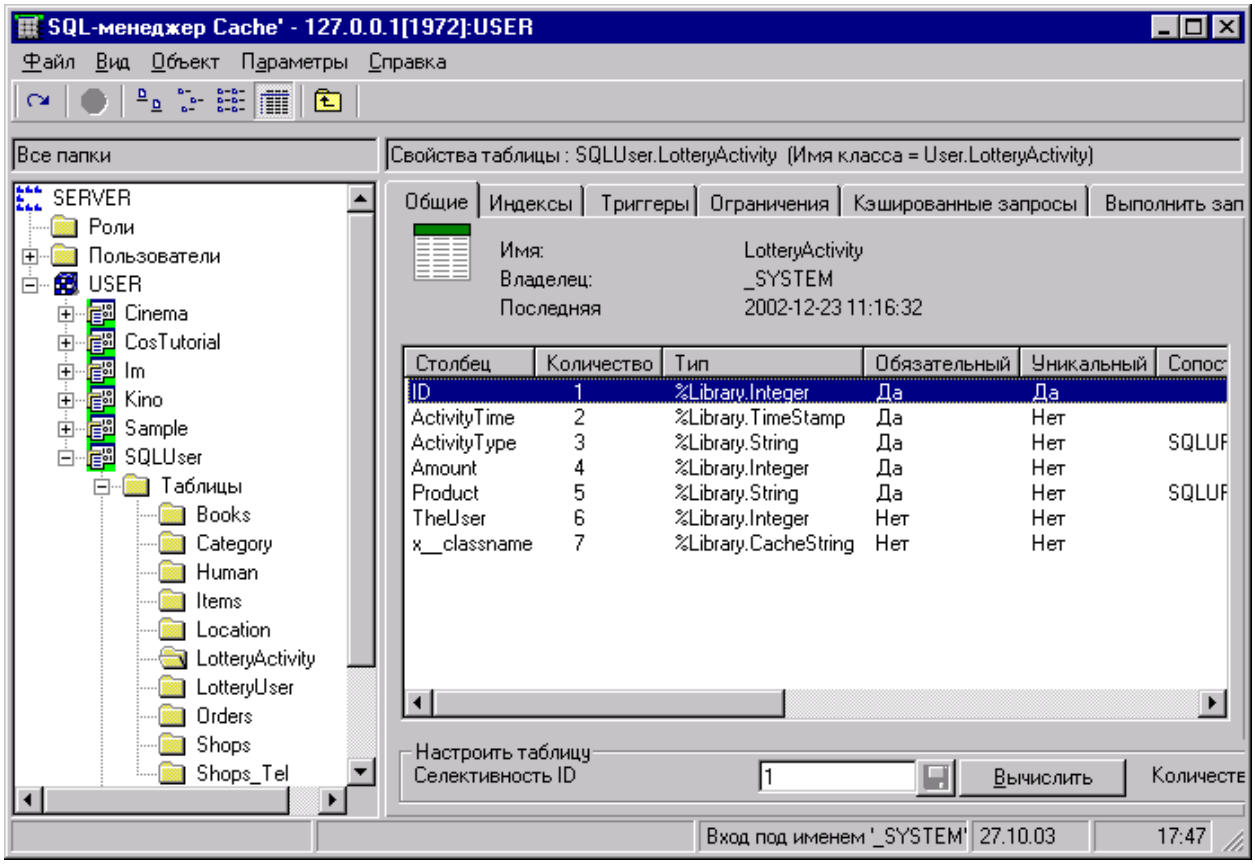


Рисунок 9. Утилита SQL-менеджер.

Глава 3. Язык Caché Object Script

В СУБД Caché реализован собственный язык программирования Caché Object Script (COS). COS — это расширенная и переработанная версия языка программирования M (ANSI MUMPS). В первую очередь, COS предназначен для написания исходного кода методов класса. Кроме этого, в Caché вводится понятие Caché-программы. Caché-программа не является составной частью классов и предназначена для написания прикладного программного обеспечения для текстовых терминальных систем. Особенности языка Object Script:

- Язык не имеет зарезервированных слов: вы свободны в выборе имен идентификаторов, также как и имен переменных. Например: set set=12, где первый set – это команда, второй – это имя переменной.
- Не является языком строгой типизации. Используемые переменные не требуют предварительного объявления.
- Имеется большой набор встроенных функций, которые отличаются наличием знака $ перед именем функции. Например: $Piece, $Select и т.д.
- Существует довольно много встроенных команд языка. Например: write, read и т.д.
- Поддержка работы с классами и объектами.
- Поддержка работы с многомерными массивами.
- Широкий набор функций для прямого управления потоками внутри приложений.
- Набор команд ввода/вывода.
- Поддержка встроенного SQL.

3.1. Переменные, идентификация

Частично ObjectScript чувствителен к регистру букв, частично нет. В общем случае, то, что определяется пользователем, чувствительно к регистру букв, в то время как имена встроенных команд и функций – нет. Определяемые пользователем идентификаторы (переменные, подпрограммы, имена меток) чувствительны к регистру букв. String, string, и STRING это разные переменные.

Различают два вида переменных: локальные и глобальные.

Локальные – существуют лишь в оперативной памяти, всегда привязаны к одному процессу и являются локальными для данного процесса.

Глобальные длительно хранятся в базе данных. Глобальные данные (глобалы) представляют основу чрезвычайно эффективного прямого доступа (Direct Access), являются многопользовательскими, доступны для обработки другими процессами. Для того чтобы отличить их от локальных, перед именем глобала ставится символ «^». Глобальные переменные могут быть скалярами, массивами или подпрограммами.

Идентификация переменных

1. Длина имени не ограничена, но определяющими являются только первые 31 символов.
2. Первый символ – строчная или прописная буква или «%».
3. Все остальные символы могут представлять собой смесь букв и цифр из 7-ми битного набора символов ASCII.
4. Имена чувствительны к регистру букв.

Локальные и глобальные переменные могут создаваться тремя командами: Read, Set, For. С помощью команды Kill переменные могут быть уничтожены.

Примеры:

Set Article=”Брюки”

Read “Введите число”, number
Kill Article, number

В Caché отсутствует декларирование переменных: единственный тип данных – это строка символов переменной длины. Т.е. все переменные – это переменные типа String. Тем не менее, для выполнения арифметических или логических операций переменные интерпретируются как числа, либо как логические данные.

Еще одна категория переменных – это **системные переменные**, которые являются предопределенными и могут меняться программно только в редких случаях. Их имена начинаются с символа $. Пример системной переменной — $Horolog, которая содержит внутреннее системное время.

Внутренние структурированные системные переменные – это смешение описанных выше системных и глобальных переменных. Синтаксически они предваряются двумя символами “^$”. Они содержат системную информацию. Например, в системной структурированной переменной ^$Job хранится информация о текущем процессе (см. табл.1).

Таблица 1

Примеры системных переменных

Системная переменная	Описание	Пример
$S[torage]	Размер области текущего процесса, предназначенный для хранения локальных переменных в байтах	> write $S > 16694336
$ZN[ame]	Имя загруженной в данный момент программы	>write $ZN >
$ZV[ersion]	Версия Caché	
$J[OB]	Содержит положительное целое число, однозначно идентифицирующее каждый текущий процесс, происходящий в системе. Не меняется в течение всего времени, пока процесс является активным	
$ZPI	Число пи	

3.2. Команды

3.2.1. Общий синтаксис команд

Общий синтаксис команды:

<метка> <пробел> <команда> <пробел> <список аргументов через запятую>

Можно использовать более одной команды в одной строке. При этом должен быть как минимум один пробел между последним аргументом одной команды и началом следующей команды. Многие команды могут не иметь аргументов.

Использование пробелов в строках кода:

- После безаргументной команды должны следовать как минимум 2 пробела.
- В именах переменных не может быть пробелов.
- Каждая строка кода должна начинаться с пробела, единственное исключение из этого правила – использование меток. Если строка имеет метку, должен быть один пробел между меткой и кодом.
- Между именем команды и списком аргументов должен быть один и только один пробел. Если в команде используется постусловие, не должно быть пробелов между командой и условием.

Пример ошибочного использования оператора (команда набирается в Caché Terminal):

USER> SET▪▪nextvar="Переменная"

Где ▪ - это пробел

Примеры, демонстрирующие правильное использование пробелов:

USER> SET·nextvar="Переменная"

USER> KILL··SET·nextvar=5··WRITE

USER> KILL▪SET▪nextvar=5

3.2.2. Команда Write

Команда Write используется для вывода информации на экран монитора. Общий синтаксис команды:

Write <список аргументов через запятую>

Список аргументов может содержать: переменные, выражения, строки, формат. Все аргументы команды необязательны. Write без аргументов выдает содержимое переменных памяти. Для задания формата используются следующие символы:

! – начать новую строку;

– начать новую страницу;

?n – вывод с позиции n экрана.

Например:

SAMPLES>write "Привет всем!!!"

SAMPLES>write !, "Это", !, "многострочное ", !, "сообщение", !

Это

Многострочное

сообщение

SAMPLES>write !, "Это", ?10, "сообщение выводится", ?50, " по колонкам", !

Это сообщение выводится по колонкам

SAMPLES> write a

1

SAMPLES>write 7.95 * 1.15

9.1425

3.2.3. Команда Read

Позволяет выдавать подсказку пользователю и вводить ответ пользователя в переменную. Синтаксис:

Read <список аргументов через запятую>

Список аргументов может содержать: переменные, выражения, строки, формат. Все аргументы команды необязательны. Для задания формата используются следующие символы:

! – начать новую строку;

– начать новую страницу;

?n – ввод с позиции n экрана;

* (звездочка) перед переменной – это ввод одного символа;

: (двоеточие) после имени переменной – задает время ожидания ввода в секундах;

<имя переменной>#<число> – задает ввод нескольких символов, количество которых задается числом.

Можно использовать команду write для вывода значения переменной. Имена переменных чувствительны к регистру букв. Пример:

```
SAMPLES>read x
      Большой привет
SAMPLES>write "Вы ввели: ", x
```

Вы ввели: Большой привет

Так как READ разрешает ввод подсказки, то она разделяет функциональность команды write. Можно ждать ответ пользователя неограниченное время или задать ограничение на ожидание: как долго (в сек.) ждать. Если пользователь ничего не ввел и нажал Enter, переменная будет содержать пустую строку.

Можно ограничить число вводимых символов, например:

```
SAMPLES>read !, "Введите 5 символов и не нажимайте  <Enter>: ", z#5
Введите 5 символов и не нажимайте  <Enter>: abcde
```

Можно ограничить время ожидания. Например:

```
SAMPLES>read ?30, "Введите Ваше имя: ", n
Введите Ваше имя: Александр
SAMPLES>read !, "У Вас только 5 секунд на ответ: ", x:5
У Вас только 5 секунд на ответ:
SAMPLES>
```

3.2.4. Команда Set

Команду Set можно использовать для задания арифметических выражений и назначения значений переменным. Формы команды Set:

Set <перем1> = <выр1>, < перем2> = < выр2>

Set (<перем1>, <перем2>, <перем3>, …) = <выр>

Set <объект>.<свойство> = <значение>

Set (<объект1>.<свойство>, <объект2>.<свойство>, <объект3>.<свойство>) = <значение>

Например:

```
Set a=4,b=5,c=10
Set (sc, fam, d)=”Строка”
Set Human.Im=”Александр”
Set (per1.pol, per2.pol, per3.pol)=”Ж”
```

3.2.5. Арифметические операторы

ObjectScript использует 7 бинарных арифметических операторов. Первые два являются также унарными операторами. В табл. 2 приведены основные арифметические операции.

Таблица 2

Арифметические операции

Оператор	Операция
+	Сложение. Если используется перед выражением, то это унарный оператор, который усиливает числовую интерпретацию выражения. Например: set x = 4 + 2 (в результате x=6) set z = "546-FRJ" set y = +z (в результате y=546)
-	Вычитание. Если используется перед выражением, то трактуется как унарный минус, т.е. меняет знак выражения и усиливает числовую интерпретацию выражения. Например: set x = 4 - 2 (x=2) set z = "-10 degrees" set y = -z (y=10)
*	Умножение. Например: set x = 4 * 2 (x=8)
/	Деление. Например: set x = 5 / 2 (x=2.5)
**	Возведение в степень: set x = 5 ** 2 (x=25)

\	Целочисленное деление. Например: set x = 5 \ 2 (x=2)
#	Остаток от деления: set x = 5 # 2 (x=1)

Например:

```
SAMPLES>set x = 4 + 2
SAMPLES>write x
  6
```

Операторы в выражении выполняются строго слева направо**, нет приоритетов операций**, об этом нужно помнить, чтобы избежать ошибок. Для изменения порядка следования операций можно использовать скобки. Например, сравните 2 выражения:

Set sum=sum+(num#10)+(num\10) и Set sum=sum+num#10+num\10.

Вы получите разный результат.

3.2.6. Оператор Do

В ObjectScript все подпрограммы как системные, так и пользовательские, запускаются с помощью команды Do. Формы использования команды:

Вызов программы P1: Do ^P1, где P1 – имя файла, где располагается программа.

Вызов подпрограммы P2 из программы P1: Do P2^P1, где ^ - означает вызов глобальной программы, т.е. программы, которая располагается в другом файле.

Вызов глобальной программы без параметров: Do ^sum.

Вызов глобальной программы с параметрами a и b: Do ^sum(a, b).

Вызов подпрограммы внутри текущей программы Do sum.

Для демонстрации, рассмотрим системную подпрограмму, которая вычисляет квадратный корень числа – %SQROOT. Можно запустить ее с помощью команды Do, поставив перед ней знак ^ (^ – циркумфлекс). Заметим, что подпрограммы, перед которыми стоит знак %, являются системными. В процессе выполнения программа запрашивает число, из которого нужно извлечь квадратный корень. Например:

```
SAMPLES>Do ^%SQROOT
Square root of: 100 is: 10
Square root of:
```

К тому же, можно запустить процедуру внутри подпрограммы, ссылаясь на метку строки (называемую также тегом), где процедура начинается в подпрограмме. Метка располагается непосредственно перед циркумфлексом (^). Например:

```
Set %X = 523
Do INT^%SQROOT
Write %Y
```

В данном примере задается значение системной переменной %X, которое используется процедурой INT программы %SQROOT. Процедура INT вычисляет квадратный корень из числа 523 и сохраняет его в системной переменной %Y.

3.2.7. Команда Kill

Команду Kill можно использовать для удаления переменной, а безаргументная форма может использоваться для удаления всех переменных памяти. Синтаксис:

Kill <переменные>

Команды Kill и безаргументная форма Write могут быть полезны при отладке программы. Например:

```
SAMPLES>set a = 1, b = 2, c = 3, d = 4, e = 5
SAMPLES>write
a=1
```

```
b=2
c=3
d=4
e=5
        SAMPLES>kill c
        SAMPLES>write
a=1
b=2
d=4
e=5
```

3.2.8. Команда Quit

Эта команда используется для завершения выполнения подпрограммы. Ее можно размещать в нескольких местах подпрограммы. Если Quit не имеет аргументов, оставляйте всегда 2 пробела после нее.

Замечание по поводу ошибок в подпрограммах. Caché Studio будет выявлять синтаксические ошибки, которые делаются при вводе оператора. Но другие ошибки, такие как undefined variable во время выполнения будут завершать выполнение программы. Вы увидите сообщение об ошибке вместе с номером строки, содержащей ошибку в окне Caché Terminal. К тому же, подсказка:

```
SAMPLES>
```

будет изменена на подсказку:

```
SAMPLES 2d0>.
```

Введите Q (аббревиатура команды Quit) для того, чтобы вернуть подсказку в нормальное состояние. Например вызов программы ^badroutine приводит к ошибке <UNDEFINED> :

```
SAMPLES>do ^badroutine
<UNDEFINED>start+3^badroutine
```

Введите команду Quit, чтобы вернуть подсказку в нормальное состояние:

```
SAMPLES 2d0>Quit
SAMPLES>
```

3.2.9. Комментарии

// – текст после // и до конца строки рассматривается как комментарий.

; – текст после ; и до конца строки рассматривается как комментарий

/* */ – многострочный комментарий

3.2.10. Конструкция If

Синтаксис команды:

```
if condition {code} elseif condition {code} else {code}
```

Где condition - это условие вида:

```
condition = <вып1>[,<вып2>,...,<вып3>]
```

Чтобы блок программы выполнился, все условия в списке должны быть логически истинными.

code это блок кода, т.е. несколько строк кода, заключенных в фигурные скобки.

Пример 1:

```
Set x=0,y=-1,a=0
      if x<1,y<1 Set a=1
  write a
```

Пример 2:

```
root
   read "Введите пароль", parol
     if ( parol="salut")
     { write !, "Приветствуем Вас!"
      }
      else
     { write !,"Пароль не верен",!    }
   quit
```

3.2.11. Операции сравнения и логические операции

Операции сравнения :

= (равно), > (больше) , < (меньше), '= (не равно), '> (не больше), '< (не меньше).

Логические операции:

& , && – логическое И ;

!, || – логическое ИЛИ ;

' – отрицание.

Отличие & от && и ! от || заключается в том, что операторы && и || прекращают выполнение операции, если вычисление левого операнда уже дает окончательный результат.

Примеры истинных логических выражений:

```
if (2 = 2) && (3 = 3) {  write "оба выражения истинны" }
if (2 = 2) || (3 = 4) { write "только одно истинно" }
if '(2 = 3) {  write "не равны" }
```

3.2.12. Постусловные конструкции

Синтаксис:

<команда> : <условие>

Команда выполняется, если условие истинно. Почти все команды Caché поддерживают постусловный синтаксис, кроме команд if else/elseif, For, While, Do/While.

Например:

1. Quit: name=” ” , что эквивалентно If name=” ” quit
2. Write: count<5 “число меньше 5”,!
3. Write: count>5 “число больше 5”,!

3.2.13. Функции $Case и $Select

Функция $Case вычисляет значение первого аргумента и возвращает результат, который соответствует значению выражения в списке аргументов. Последний аргумент может быть значением по умолчанию. Синтаксис функции $Case:

$Case(<перем>,<знач1>:<выр1>[,<знач2>:<выр2>...][,:<знач по умолч>]))

$Case может возвращать литерал, имя процедуры или подпрограммы, например:

```
READ "Введите число от 1 до 3: ", x
SET multi = $CASE(x, 1:"единица", 2:"двойка", 3:"тройка", :" ошибка ввода")
write multi
```

```
do $case( surv, 1:celebrate^survivor() , :complain^survivor() )
Yippee! I won!    // результат выполнения подпрограммы celebrate^survivor()
```

Функция $Select очень напоминает функцию $Case и имеет следующий формат:

$SELECT(<выражение1>:<значение1>,<выражение2>:<значение2>,...)
$S(<выражение1>:<значение1>,<выражение2>:<значение2>,...)

Логические выражения отделяются от значений двоеточиями. Функция $SELECT просматривает выражения слева направо и возвращает значение соответствующее первому истинному выражению. Например:

```
Start
  READ !,"Задайте номер уровня: ", a
  QUIT:a=""
  SET x=$SELECT(a=1:"Level1",a=2:"Level2",a=3:"Level3")
  GOTO @x
```

3.2.14. Конструкция For

Конструкция For используется для создания циклов. Общая форма команды For:

For <переменная>=<параметр> {код}

Где переменная – это счетчик цикла. Параметр может иметь несколько форм использования:

- список выражений, разделитель списка – запятая.
- <начальное значение>:<инкремент>
- <начальное значение>:<инкремент>:<конечное значение>

Все параметры цикла необязательны. Цикл For имеет две формы использования без аргументов и с аргументами. Безаргументная форма реализует бесконечный цикл. Должен быть обеспечен выход из цикла либо с помощью оператора Quit, либо с помощью Goto.

Различные формы конструкций цикла могут свободно комбинироваться.

Примеры:

1. Параметр – это список различных выражений:

```
For prime=2,3,5,7,11 {write !,prime, "это простое число" }
```

2. Задание числового диапазона:

```
For i=1:1:10 {write !,"i"," ",i**2 }
```

3. Числовой диапазон без конечного значения:

```
for i = 2:2
{ write !,i
   Quit:i>15
  }
  write i
```

4. Безаргументная форма. В этой форме нет ни переменной цикла, ни начального, ни конечного значения:

```
For  {код}   // как минимум 2 пробела после For
 Например:
 For    {
        Read !,"Введите число:", num
        Quit:num=""
        Do Calc(num)
       }
```

5. Комбинированная форма:

```
For lv = 2, 5, 49, 1:1:15, 1:2 { код }
```

В этом случае, цикл сначала выполнится для значений параметра 2, 5, 49, затем для значений 1:1:15, затем для 1:2. В последнем случае должен быть обеспечен выход из цикла.

3.2.15. Конструкция While и Do/While

Работают также как и for без аргумента, т.е. повторяют набор команд, и завершают выполнение по условию. While – цикл с предусловием, Do/While – цикл с постусловием. Как и в случае for команда Quit внутри блока завершает цикл. Синтаксис:

```
do {code} while condition
while condition {code}
```

где condition - это условие:

condition = <вып1>[,<вып2>,...,<вып3>]

Чтобы блок программы выполнился, все условия в списке должны быть логически истинными.

Пример программы с числами Фибоначчи. Программа набирается в Caché Studio, текст программы приведен ниже:

```
fibonacci   ; генерирует числа Фибоначчи
 read !"Генерируем числа Фибоначчи до: ", upto
 set t1=1, t2=1, fib=1
 write !
  do { write fib," " set fib=t1+t2, t1=t2 ,t2= fib
     }
  while (fib'>upto)
 set t1=1, t2=1, fib=1
   write !
  while (fib'>upto)
    { write fib," " set fib=t1+t2, t1=t2 ,t2= fib
    }
```

Выполнение программы в Caché Terminal:

```
SAMPLES>do ^fibonacci
Генерируем числа Фибоначчи до: 100
1  2  3  5  8  13  21  34  55  89
1  2  3  5  8  13  21  34  55  89
```

3. 3. Программы Caché

3.3.1. Виды программ

Caché позволяет создавать 3 вида программ:

1. Макрокод (расширение MAC) может содержать код ObjectScript, макродирективы, макросы, встроенный SQL, встроенный HTML. Он компилируется в промежуточный код, а тот уже в объектный(OBJ).
2. Промежуточный код (расширение INT) это рабочий код ObjectScript. Макросы и встроенный SQL сначала преобразуются в промежуточный код.
3. Включаемые макропрограммы (расширение INC) могут содержать любой код, разрешенный в MAC-программах. Они применяются для построения макробиблиотек и через директиву Include – отсюда и их название – могут быть использованы в макропрограммах.
4. Объектный код (.OBJ) – это промежуточный код, компилируется в объектный код. Это происходит автоматически при сохранении отредактированной программы. Во время выполнения используется только объектный код. Фирмы разработчики тиражируют свои приложения большей частью в объектном коде.

Диалог сохранения также позволяет задать расширение RTN, которое не является реальным расширением для программ. Это просто способ показать все программы сразу без дублирования.

Текст программы набирается в Caché Studio. Проверка выполнения производится в Caché Terminal.

В случае ошибки выдается имя метки+ смещение, где произошла ошибка. Например:

```
SAMPLES>do ^badroutine
<UNDEFINED>start+3^badroutine
SAMPLES 2do>quit
SAMPLES>
```

Строка программы может иметь метку в начале, называемую тегом, код команды и комментарий, но все эти элементы являются необязательными. Все эти элементы отделяются друг от друга табуляцией или пробелами. Например:

```
end  quit     ; конец
```

3.3.2. Основные структуры программы Caché

Программа в Caché состоит из индивидуальных блоков, создаваемых и редактируемых как исходный код. Программа состоит из подпрограмм, процедур и функций.

Имя программы – определяется именем MAC-файла, в котором она хранится, это любая комбинация буквенно-цифровых символов ASCII, имя не должно начинаться с цифры. При сохранении программы в файле ее имя не должно содержать символы: _ (подчерк), - (минус), ; (точка с запятой). Имя, включающее эти символы, не является верным.

Вызов программы:

```
Do ^RoutineName
```

Первой строкой программы может быть оператор с меткой (тегом), метка может сопровождаться списком входных параметров, заключенных в круглые скобки.

Программа – это контейнер для подпрограмм, методов, процедур и функций. Программы, имена которых начинаются с %(процента), считаются библиотечными программами.

Строение программной строки

[метка] [код команды] [комментарий]

В программах в основном используются процедурные блоки, которые заключены в фигурные скобки и поэтому могут располагаться на нескольких строках, а также командные конструкции условного выполнения программы, в которых исполняемый программный код также заключен в фигурные скобки.

3.3.3. Общая структура подпрограммы

Подпрограмма это именованный блок кода внутри программы, состоит из входной метки, отдельных программных строк, содержащих коды, и заключительной команды Quit. Она может содержать параметры и никогда не возвращает значение. Синтаксис подпрограммы:

```
Label ([param [= default]])          // комментарий
        // код
 Quit                          // заметьте, что Quit не имеет параметров
```

Где, Label – имя подпрограммы, param – это параметры подпрограммы, default – значения по умолчанию.

Вызов из этой же программы возможен с параметрами, при этом можно задать значения по умолчанию.

```
Do Label[(param)]
```

Синтаксис вызова подпрограммы из другой программы:

```
Do Label^Routine[(param)]
```

Где Routine – программа, которая находится в файле Routine.Mac, Label – подпрограмма программы Routine, param – список параметров.

Пример 1. Программа, находящаяся в файле routine.Mac имеет вид:

```
 set x=1,y=2
  do minim(x,y)   // вызов из этой же программы
 write !,"x=",x,"  y=",y
 quit
minim(a=0,b=0)  // подпрограмма
 if (a>b)
  { set mn=b }
 else
  { set mn=a}
 write !,"min=",mn
quit
```

Вызов подпрограммы minim из этой же программы:

```
do minim(x,y)
```

Вызов подпрограммы minim из внешней программы:

```
do minim^routine(9,2)
```

Возможен вызов без параметров, в этом случае используются значения параметров по умолчанию:

```
do minim^routine()
```

Если циркумфлекс(^) при вызове отсутствует, например Do minim(x,y), это означает, что метку следует искать в текущей программе.

Можно также использовать синтаксис вызова: метка + смещение, и таким образом войти в N-ую строку программы. Пример: Do minim+3^routine – вход в программу через вторую строку кода. Такую возможность можно использовать при тестировании.

3.3.4. Встроенные функции

В Caché имеется набор предопределенных (встроенных) функций, которые можно использовать для различных целей. Встроенные функции могут использоваться в любых выражениях. Вызов функции заменяется ее значением, вычисляемым в зависимости от аргументов. Функции в Caché являются рекурсивными, т.е. могут содержать в виде аргументов самих себя. Имена встроенных функций начинаются со знака $. Вызов имеет следующий синтаксис:

$Function(Arg1, Arg2, ...).

Имена функций пишутся строчными или прописными буквами. В табл.3 приведен список наиболее часто используемых встроенных функций.

Таблица 3

Классификация встроенных функций

Класс функций	**Типичные представители**
Общие функции	$Ascii, $CASE, $Char, $Random, $Select, $Stack, $Text, $View
Операции с переменными и базой данных	$Data, $Get, $Order, $Name, $Query, $QSubscript, $QLength
Обработка строк	$Extract, $Find, $Length, $Piece, $Reverse, $Translate
Форматирование чисел	$FNumber, $Justify, $Inumber, $NUMBER
Обработка списков	$List, $ListBuild, $ListData, $ListFind, $ListGet,

	$ListLength
Обработка транзакций	$Increment
Математические функции	$ZABS, $ZEXP, $ZLN, $ZSIN
Дата, время	$ZDate, $ZDateTime, $ZDateH, ZTime, ZtimeH
Общие функции	$ZWAscii, $ZWChar, $ZF,$ZHex, $ZLAscii, $ZLChar, $ZSEArch, $ZSEEK
Строковые функции	$ZCONVert, $ZCyc, $ZSTRIP
Обработка битовых строк	$ZBitAND, $ZbitCount, $Zboolean

Более подробная информация о встроенных функциях содержится в документации по Caché.

3.3.5. Пользовательские функции

Наряду со встроенными функциями, пользователю предоставляется возможность писать свои функции. По способу построения и использования в выражениях функция очень похожа на подпрограмму, отличие заключается в том, что функция может возвращать значение. Если код функции заключить в фигурные скобки, то функция будет трактоваться как процедура. Синтаксис:

```
ИмяФункции (СписокПараметров)
        <код>
    Quit <выражение>
```

Функция может использоваться в выражениях т.к. возвращает значение.

Вызов функции из этой же программы:

```
Set <Переменная>=$$<ИмяФункции>(Список Параметров)
Do <ИмяФункции>(СписокПараметров)
```

Вызов функции с помощью Do ничем не отличается от вызова подпрограммы.

Внешний вызов из другой программы выглядит следующим образом.

Функцию можно вызвать оператором Do, если не требуется возвращаемое функцией значение, в этом случае вызов ничем не отчается от вызова подпрограммы:

```
Do <Имя Функции>^<Имя Программы>(СписокПерем)
```

Если требуется возвращаемое значение:

```
Set <Перем>=$$<Функция>^<Программа>(СписокПерем)
```

Пример 1. Функция, которая возвращает случайное число в некотором диапазоне в файле FRnd.Mac:

```
        set a=200,b=100
        set r=$$Rnd(a,b)  // вызов из той же программы
        write !,"r=",r
        Quit
    Rnd(m,n) // функция
        Quit $Random(m-n)+1+m
```

Вызов функции из другой программы:

```
set r=$$Rnd^FRnd(200,100)
```

Пример 2. Программа func1.Mac содержит функцию minim:

```
    set x=6,y=2
     set mn=$$minim(x,y) // вызов из этой же программы
     write !,"x=",x,"  y=",y,"min=",mn
```

```
        quit

    minim(a,b)  // функция
        write "minim"
        if (a>b)
         { quit b }
        else
         { quit a}
       quit
```

Вызов функции из внешней программы:

```
Set m=$$minim^func1(9,2)
```

3.3.6. Процедуры

Процедура именует набор операторов языка ObjectScript, для того, чтобы этот набор можно было вызвать из любого места программы, возможно несколько раз, возможно с разными параметрами. Этот набор операторов заключается в фигурные скобки {}. Форма описания:

<имя>(СписокФормальныхПараметров)[<СписокОбщедоступныхПараметров>] <Доступ>
{ <тело процедуры>
[Quit <значение>]
}

СписокФормальныхПараметров – локальные параметры процедуры, доступны только внутри процедуры, в вызываемых ею процедурах не видны.

СписокОбщедоступныхПараметров – необязателен. Общедоступные параметры доступны во всех процедурах, которые данная процедура вызывает и в тех, которые вызывают данную процедуру. Для определения public-переменных их список помещается в квадратных скобках за именем процедуры и списком ее локальных параметров.

Доступ – задает доступность процедуры: private или public. Private (по умолчанию) – означает, что процедура доступна только внутри данной программы, public – процедура доступна для внешних программ.

Задание параметра:

(<имя параметра>)[=<Значение по умолчанию>]

Вызов процедуры из этой же программы:

```
Do Proc(<список параметров>)
```

Вызов процедуры из внешних программ:

```
Do Proc^Routine(<список параметров>)        // не возвращает значение
```

Где Proc – имя процедуры, Routine – имя программы.

Пример 1: Файл proc2.Mac содержит следующий код:

```
        set c=1,d=3
        do MyProc(1,2)        // внутренний вызов процедуры
        quit
MyProc(X,Y)[A,B] Public
        {  write "X+Y=",  X+Y,  !
           set A=1,B=4
           write "A+B=", A+B, !
        }
```

Вызов процедуры из той же программы:

```
Do MyProc(1,2)
```

Вызов процедуры из другой программы:

```
Do MyProc^proc2(1,2)
```

Пример 2. Программа proc2.Mac, содержит вызов процедуры MyProc. Процедура MyProc в свою очередь вызывает процедуру abc, которой передаются локальные параметры x, y и глобальные параметры a, b:

```
set c=1,d=3
 do MyProc(1,2)

 // процедура MyProc
MyProc(X,Y)[A,B] Public
 {  write "X+Y=",  X+Y,  !
   set A=1,B=4
     do abc(X,Y)  // вызов процедуры abc из MyProc
  write "A=", A, " B= ", B, !
}

 // Процедура abc, вызываемая из MyProc
abc(n,m)[A,B]
 { set n=9,m=5
  write "a=",A,"b=",B
 }
```

3.3.7. Область действия локальных переменных

Все однажды созданные переменные программ, подпрограмм и функций видны повсюду, даже после завершения программы, и доступ к ним ограничивается программистом, если он к этому стремится.

В примере 1 п.3.3.3. переменные x и y, созданные в программе routine.Mac, сохраняются также в подпрограмме minim, более того, они сохраняются после выполнения программы. Это может быть нежелательно. Часто необходимо ограничить доступ к некоторым переменным до уровня конкретных подпрограмм или процедур.

В Caché имеется ряд возможностей для «очистки» локальных переменных:

1. С помощью команды Kill в конце программы, но перед заключительной командой Quit для удаления переменных после выхода из программы.
2. Использование команды New для задания списка локальных переменных для программы или подпрограммы, которые из нее вызываются.
3. Передача параметров по значению.
4. Определение public и private переменных в процедурах, рассматривая их как инкапсулированные блоки в подпрограмме.

3.3.8. Команда New

Команда New задает список переменных, локальных для данной программы и всех вызываемых из нее подпрограмм. При выходе из программы такие переменные неявно удаляются. Синтаксис:

```
New  [( Список переменных)]
```

New – без аргументов изолирует все переменные, ее следует применять осторожно.

Пример:

```
New a,b   // изолирует две переменные a и b
```

Пример:

```
P1                  //  первая строка программы
```

```
    New a
    Set a=1 Do P2
   Write
  Quit
P2                  // первая строка подпрограммы
    New x, y
    Set x=a, y=3
Quit
```

В этом примере переменная a существует в P1 и P2, т.к. P2 вызывается из P1. Переменные x и y существуют только в P2. Передача их значений в вызывающую программу невозможна.

Можно использовать команду New в процедурах. При этом переменные, созданные с помощью команды New отличаются от локальных(private) переменных процедур. Команда New используется только для public-переменных, при этом public-переменная, объявленная с помощью New, существует с момента объявления до оператора Quit процедуры, в которой она была объявлена. Таким образом, можно скрыть public-переменные с теми же именами, которые возможно уже существуют на момент вызова процедуры.

Например, имеем код в файле abc.Mac:

```
abc
 set name="Tom"
 do MyProc(9,4)
 write !,"abc name= ", name

MyProc(x,y)[name]{
 New name
 Set name="John"
 write !,"MyProc name = ",name
 Do xyz()
 Quit
}
xyz()[name]
 { write !,"xyz name = ",name

 }
```

В данном примере процедуре “xyz” в программе “abc” позволяется видеть значение “John” переменной name, т.к. name – public-переменная. Вызов команды New для переменной name защитит (скроет) public-переменную с именем “name” и со значением “Tom”, которая уже существует к моменту вызова “MyProc”.

В результате вызова программы ^abc будет напечатано:

```
MyProc name = John
xyz name = John
abc name= Tom
```

3.3.8. Передача параметров

Передача параметров происходит при:

- Глобальном или локальном вызове программы
- Вызове пользовательской функции
- Вызове процедуры.

Формы передачи параметров:

- По значению

- По ссылке.

3.3.10. Передача по значению

При передаче по значению формальные параметры в начале вызываемой программы подчиняются неявной команде New. При завершении вызываемой программы выполняется неявная команда Kill. Т.е. формальные параметры существуют только внутри вызываемой программы. Пример:

```
 Set a=1, b=2, c=3
 Do  p1(a,b,c)  // фактические параметры
 write
…
P1(r,s,t)          // формальные параметры
  Set summ  =  r+s+t
  Quit
```

При вызове подпрограммы список фактических параметров должен точно соответствовать списку формальных параметров. Отдельные фактические параметры могут отсутствовать, однако запятые, обозначающие пропуск параметра должны быть поставлены.

3.3.11. Передача по ссылке

Синтаксис передачи по ссылке – это точка перед именем переменной в списке параметров. Переменная может, в отличие от метода, не существовать. Пример:

```
    Kill var                          // var не существует
Do P1(.var)                    // вызов с передачей параметра по ссылке
…
P1(x)                                 // формальный параметр x
Set x=0, x(1)=1, x(2), x(3)=3
…
Quit
```

После выхода из подпрограммы P1 в основной программе существуют переменные var, var(1), var(2), var(3).

При вызове подпрограммы обе формы передачи параметров могут смешиваться:

```
Do ^Summ(a, b, c, .var)
```

3.3.12. Процедуры, программы, подпрограммы, функции, методы. Отличия и особенности

Наиболее гибкой и мощной формой является пользовательский блок, оформленный в виде процедуры. Особенности процедур:

- Может быть private или public.
- Может иметь ноль или более аргументов.
- Любые внутренние переменные процедуры являются локальными.
- Может изменять внешние переменные или ссылаться на них.
- Может возвращать значение любого типа или не возвращать ничего.

Для сравнения:

- Подпрограмма всегда является public и не может возвращать значение.
- Функция всегда является public; требует явного объявления локальных переменных, иначе перекрывает внешние переменные; должна возвращать значение.
- По умолчанию, метод является процедурой, которая объявляется как часть определения класса.
- Программа Caché ObjectScript может включать одну или более процедур, подпрограмм, функций, так же и различные их комбинации.

Замечание: ObjectScript также поддерживает особую форму пользовательского кода с помощью механизма макровызовов.

3.4. Оператор косвенности @

Оператор косвенности позволяет преобразовать строку символов в программный код в четырех четко описанных случаях:

- Косвенное имя: преобразование в имя (например, переменной).
- Косвенный аргумент: преобразование в полный аргумент команды.
- Индексная косвенность: преобразование в индексированную переменную.
- Косвенный шаблон: преобразование в шаблон при проверке по шаблону.

3.4.1. Косвенное имя

Строка преобразуется в имя переменной или имя программы.

Пример:

```
Set pname=”Prog1”
  do @pname  // будет вызвана программа с именем  Prog1
Set var=”locvar”
Set @var=4   // переменной с именем locvar будет присвоено число 4
```

3.4.2. Косвенный аргумент

Строка преобразуется в аргумент команды, за исключением for, который косвенности не допускает.

Пример:

```
 Set isetarg=”x=1”, @isetarg    // set x=1
 Set ikill=”(e,f,g)”
   Kill @ikill
Set inew=”(a,b,c)”   New @inew
```

3.4.3. Индексная косвенность

Служит для расширения имен глобальных и локальных переменных. Ее синтаксический признак – дважды встречающийся оператор косвенности. Первый из них преобразует идущую вслед за ним строку в имя переменной, второй добавляет к полученному имени заданный в скобках индекс (или индексы):

Пример:

```
Set x(2, 5,3)=“IndInd”
Set feld=”x(2,5)”,  d1=3
Write @feld@(d1)
Будет напечатано:
   IndInd
```

Мы видим расширение ссылки x(2,5) индексом третьего уровня со значением 3.

3.4.4. Косвенный шаблон

С помощью косвенного шаблона появляется возможность при сравнении по шаблону рассматривать полный шаблон косвенно. Пример:

```
Set lvmuster=”1.3N”
If  eingabe ‘?@lvmuster Do error
```

3.5. Работа со строками

Так как данные в Caché хранятся как строки, то работа со строками имеет большое значение. COS не является языком строгой типизации, в отличие от большинства известных языков программирования. Все значения данных интерпретируются с соответствии с контекстом, в котором они используются. При этом имена переменных чувствительны к регистру букв, в то время как команды нет. Примеры интерпретации данных в соответствии с контекстом приведены ниже.

```
SAMPLES>set x = 4  write x + 2
6
SAMPLES>set x = "5"   write x * 2
10
SAMPLES>set x = "fred"  write x + 3
3
SAMPLES>set x = "32hut"  write x / 4
8
SAMPLES>write +x
32
SAMPLES>write 5 + 2, ?10, "5" + "2" , ?20, "5 + 2"
7        7        5 + 2
SAMPLES>
```

3.5.1. Сохранение строк на диске

Для сохранения строки на диске перед именем переменной нужно поставить ^ (циркумфлекс). В этом случае, если удалить все переменные с помощью команды kill, глобальные переменные сохраняются. Пример:

```
SAMPLES>set x = 4
SAMPLES>set ^x = "Глобальная переменная"
SAMPLES>kill
SAMPLES>write
SAMPLES>write ^x
Глобальная переменная
```

Для удаления глобальной переменной введите: SAMPLES > kill ^x.

3.5.2. Операторы для строк

Для строк определены два оператора:

(_) – оператор конкатенации, сцепления строк

(?) – оператор соответствия шаблону, один из наиболее интересных операторов Cache Object Script.

3.5.3. Оператор соответствия шаблону – ?

Оператор ? – используется в логических выражениях и проверяет соответствие строки шаблону. Синтаксис:

<Строка> ?<шаблон>

Если соответствует, то выражение возвращает 1 (true), если нет то 0 (false).

Шаблон состоит из одного или нескольких шаблонов, следующего вида:

<кол-во><чего>[<кол-во><чего>]

где <кол-во> это фактор повторения шаблона, <чего> – класс символов, которые могут повторяться. В табл. 4 приведены форматы факторов повторения, в табл.5 классы символов.

Таблица 4

Форматы факторов повторения

Кол-во	Значение
3(число)	Шаблон должен повторяться ровно столько раз
1.3	Шаблон должен повторяться от 1 до 3 раз
.3	Шаблон должен повторяться самое большее 3 раза
3.	Шаблон должен повторяться не меньше 3 раз
.	Шаблон должен повторяться любое число, включая 0

Таблица 5

Классы символов

Код	Значение	Код	Значение
A	Буква (в верхнем или нижнем регистре)	C	Управляющий символ
U	Буква в верхнем регистре	E	Любой символ
L	Буква в нижнем регистре	ANP	Комбинация кодов
N	Число	"abd"	Литеральная строка
P	Пунктуация		

Примеры:

1. 348?3N – выдаст истину, т.к. строка – это 3 цифры.
2. в переменной date находится дата в формате дд.мм.гггг. Следующая строка кода может служить проверкой правильности даты:
   if date?2N1P2N1P4N write “Ввод даты верен”.
3. Для даты также может использовать следующий шаблон: 2N1”.”2N1”.”4N.
4. 1U.AP – первый символ – прописная буква, остальные любое количество букв и знаков препинания.

3.5.4. Функции для работы со строками

Со строками работают следующие основные функции:

$Length(<строка>,[<разделитель>]) – возвращает длину строки или число подстрок строки, если указан второй параметр.

$Extract(<строка>,<выр2>,<выр3>) – возвращает подстроку из строки, начало и конец которой определяются вторым и третьим аргументом.

$Find(<строка>,<подстрока>,<позиция>) – ищет подстроку в строке и возвращает позицию символа следующего за подстрокой, третий аргумент задает начальную позицию для поиска.

$Piece(<строка>,<разделитель>,<начало>,<конец>) – рассматривает строку как набор подстрок, разделенных разделителями, возвращает нужную подстроку, находящуюся между начальным и конечным разделителями. Второй аргумент задает номер начального разделителя, третий аргумент – задает номер конечного разделителя.

$Justify(<строка или числ выражение>,<длина>,<позиция точки>) – выравнивание числа, которое может быть задано в виде строки заданной длины, с указанием нужного количества цифр после точки.

Каждая из этих функций имеет альтернативную форму, см. документацию по Caché. Подобно командам функции не чувствительны к регистру букв.

```
SAMPLES>write $length("Длина этой строки?")
18
SAMPLES>write $extract("Клад в горах", 8, 12)
горах   ; извлечь символы с 8 по 12
SAMPLES>write $find("Клад в горах", "гор")
11
SAMPLES>write $piece("кусочек пиццы", " ", 2)
пиццы
SAMPLES>write $justify(3.1415, 10, 3)
     3.142
```

Функция $Piece уникальна и требует дополнительного обсуждения. Эта функция работает со строкой, использующей разделители, например: «22-55-77», где "-" – разделитель. Она может возвращать пустую строку, если строка находится между двумя разделителями. Сама строка рассматривается как запись, а подстроки как поля этой записи. $Piece рассматривает передаваемую в виде первого аргумента строку символов как последовательность полей, разделенных разделителем, передаваемым в качестве второго аргумента.

В форме с двумя аргументами функция возвращает первое поле:

```
$Piece(<строка>, <разделитель>[, ])
```

Например:

```
SAMPLES>write $piece("кусочек пиццы", " ")
кусочек
```

В форме с тремя аргументами функция возвращает заданное третьим аргументом поле:

```
SAMPLES>write $piece("кусочек пиццы ", "ц", 1)
кусочек пи
SAMPLES>set street=$piece("Новосибирск^Русская^35^630058", "^", 2)
SAMPLES>write street
Русская
```

В форме с четырьмя аргументами – поля в области от аргумента три до аргумента четыре:

```
SAMPLES>write $piece("Новосибирск^Русская^35^630058", "^",2,3)
Русская^35
```

Также заметим, что $Length имеет ориентированный на такие строки вариант: она возвращает число подстрок в такой строке, базирующейся на разделителях.

```
SAMPLES>write $length("Новосибирск^Русская^35^630058", "^")
4
```

3.5.5. Работа с подстроками

Для замещения части строки можно использовать функции $Extract и $Piece совместно с командой Set. Функция $Extract извлекает заданные символы и заменяет их подстрокой, вне зависимости от длины подстроки.

```
SAMPLES>set empty=""
SAMPLES>set $extract(empty, 4) = "abcde" ; вставит 3 пробела слева, т.к. с 4-ой позиции нужно вставить строку "abcde"
SAMPLES>write empty
   abcde
SAMPLES>set $extract(empty, 4, 5) = "12345" ; вместо 4 и 5-го символа вставить строку "12345"
SAMPLES>write empty
   12345cde
```

Можно также использовать функцию $Piece совместно с командой Set для вставки подстроки внутри строки. Подстрока может включать участки с другими разделителями.

```
SAMPLES>set empty=""
SAMPLES>set $piece(empty, "^", 3) = "abcde"  write empty
^^abcde
SAMPLES>write $length(empty, "^")
3
SAMPLES>set $piece(empty, "^", 2) = "9/9/1999"  write empty
^9/9/1999^abcde
SAMPLES>write $piece($piece(empty, "^", 2), "/", 3)
1999
SAMPLES>
```

3.6. Работа со списками

Все данные Object Script хранятся как строки. Списки это специальный вид строки, отдельные перечни значений. Они занимают особое место и носят характер собственного типа данных. Например, имеем список:

L1={красный, зеленый, голубой}

Этот список состоит из 3-х элементов. Последовательность элементов в списке имеет значение, в отличие от множеств. Например, список:

L2={красный, голубой, зеленый} отличен от списка L1.

Список L3={красный, голубой, , зеленый} имеет 4 элемента, третий элемент не определен.

Список L4={красный, голубой, “ ”, зеленый} содержит 4 элемента, при этом 3-ий элемент это пустая строка.

Список может содержать подсписок, например:

L5={зеленый, красный, { светло-зеленый, темно-красный, сизо-голубой}, голубой}.

В Caché обработка списков играет решающую роль. При этом отдельные элементы списка обычно понимаются как строки символов, исключая ситуацию, когда элемент списка сам является подсписком. Типичные задачи для списков:

- Создать список
- Определить количество элементов списка.
- Выделить один или несколько элементов списка.
- Поиск указанного значения в списке.
- Замена отдельных значений списка новыми значениями и т.д.

Списки могут создаваться и управляться с помощью списковых функций. Например, создание списка выполняется с помощью функции $ListBuild:

USER> Set L2=$ListBuild(“красный”,”голубой”,”зеленый”)

В табл.6 приведены функции для работы со списками.

Таблица 6

Функции для работы со списками

Списковая функция	**Сокращение**	**Описание**
$ListBuild	$LB	Создать список
$ListLength	$LL	Возвращает количество элементов списка
$List	$LI	Извлекает один или несколько элементов списка
$ListGet	$LG	Предотвращает ошибку <null value>, если заданный элемент списка не определен. По смыслу эквивалентна $Get
$ListData	$LD	Логическая функция, проверяющая существование элемента списка
$ListFind	$LF	Поиск заданного элемента списка

3.6.1. Функция $ListBuild

Позволяет создать список. Синтаксис:

```
$LISTBUILD(element,...)
$LB(element,...)
```

Где element это элемент списка, строка символов в кавычках. Разделитель элементов списка – запятая. Например, создание списка X:

```
SET X=$LISTBUILD("Red","Blue","Green")
```

3.6.2. Функция $ListLength

Возвращает количество элементов списка. Синтаксис:

```
$LISTLENGTH(list)
$LL(list)
```

Где list это список. Например:

```
WRITE $LISTLENGTH($LISTBUILD("Red","Blue","Green")); вернет 3
WRITE $LISTLENGTH("") ; вернет 0
```

3.6.3. Функция $List

Извлекает один или несколько элементов списка. Синтаксис:

```
$LIST(list,position,end)
$LI(list,position,end)
```

Где list это список, Position – задает начальную позицию элементов в списке, End – конечную позицию.

Извлекаемые элементы зависят от количества параметров:

- $LIST(list) возвращает первый элемент списка.
- $LIST(list,position) возвращает элемент в указанной позиции.
- $LIST(list,position,end) возвращает подсписок, содержащий элементы списка, начиная с позиции position и заканчивая end.

Например:

```
WRITE $LIST($LISTBUILD("RED","BLUE","GREEN")) ; напечатает RED
WRITE $LIST($LISTBUILD("RED","BLUE","GREEN"),1) ;  напечатает RED
 SET X=$LISTBUILD("Red","Blue","Green")
 WRITE $LIST(X,2) ; напечатает Blue

SET X=$LISTBUILD("Green ","White ","Brown ","Black ")
   SET LIST2=$LIST(X,3,-1)
   WRITE LIST2 ; напечатает BrownBlack
```

Функция $List совместно с оператором Set позволяет заменять элементы списка, добавлять новые элементы к пустым спискам. Элемент списка сам может быть списком. Можно вкладывать функции $List для извлечения подсписков. Например:

```
>Set L1=$LISTBUILD("RED","BLUE","GREEN")
>Set $List(L1,2)= "hhhh"   ; вставить hhhh на место второго элемента
>Write $List(L1,2)
hhhh
> Set $List(L1,4)= "abcd"   ; добавить 4-ый элемент в список
```

3.6.4. Функция $ListGet

Возвращает элемент списка или требуемое значение по умолчанию, если элемент не определен. Синтаксис:

```
$LISTGET(list,position,default)
$LG(list,position,default)
```

Где list - список

Position – позиция извлекаемого элемента

Default – значение по умолчанию, если элемент не определен.

Например:

```
SET LIST=$LISTBUILD("A","B","C")
WRITE $LISTGET(LIST) ; напечатает A
WRITE $LISTGET(LIST,1) ; напечатает A
SET LIST=$LISTBUILD("A","B","C")
WRITE $LISTGET(LIST,4)  ; напечатает пробел, т.к. 4-го элемента нет
WRITE $LISTGET(LIST,4,"ERR");  напечатает "ERR"
```

3.6.5. Функция $ListData

Логическая функция, проверяющая существование элемента списка. Синтаксис:

```
$LISTDATA(list,position)
$LD(list,position)
```

Где list – список

Position – позиция элемента.

Возвращает 1 (true) – если элемент найден и имеет значение, 0 (false) – если не найден или не имеет значения. Например:

```
KILL Y  SET X=$LISTBUILD("Red",,Y,"","Green")
WRITE $LISTDATA(X,2) ; второй элемент не определен
0
WRITE $LISTDATA(X,3) ; третий элемент – переменная которой нет
0
WRITE $LISTDATA("")  ; пустая строка
0
WRITE $LISTDATA(X,0) ; элемент  в нулевой позиции
0
WRITE $LISTDATA(X,6) ; 6-ая позиция в 5-ти элементном списке
0
```

3.6.6. Функция $ListFind

Поиск заданного элемента списка. Синтаксис:

```
$LISTFIND(list,value,startafter)
$LF(list,value,startafter)
```

Где list – список, value - искомое значение.

Startafter – задает позицию списка, после которой следует выполнять поиск. Необязательный элемент. Если не задан, то поиск с первой позиции.

Возвращает номер найденного элемента, если элемент найден, и ноль (0), если элемент не найден. Например:

```
SET X=$LISTBUILD("A","B","C","D")
WRITE $LISTFIND(X,"B")
2
SET X=$LISTBUILD("A","B","C","D")
 WRITE $LISTFIND(X,"B",2)
0
```

3.7. Работа с датами

В ObjectScript дата представляется в двух форматах – внутреннем и внешнем. Внутренний формат даты это целое число, начиная с 0 для даты 12/31/1840 до 2980013 для да-

ты 12/31/9999. Внешний формат это формат более привычный для нас, например: 15/12/2003.

Также используется системная переменная $Horolog. Системные переменные работают как функции без аргументов. $Horolog содержит текущее время в виде пары чисел, первое число задает внутренний формат даты, второе число задает количество секунд, прошедших с полуночи. Например:

```
USER> write $Horolog
59469,47554
```

3.7.1. Функция $ZDate

Функция $ZDate преобразует дату из внутреннего формата во внешний формат. Функция имеет 9 аргументов. Для нас существенными являются следующие аргументы:

1 аргумент – внутренний формат даты, это число, обязателен, все остальные аргументы не обязательны.

2 аргумент – формат возвращаемой даты. Некоторые значения формата: 1 – американский, 2 и 4 – европейский, 10 – порядковый номер дня недели, при этом 0 – это воскресенье, 1- понедельник и т.д.

3 аргумент – список месяцев. Например: « январь февраль март апрель май июнь июль август сентябрь октябрь ноябрь декабрь».

9 аргумент – подавляет вывод сообщения об ошибке. Если значение аргумента равно –1, то в случае неправильной даты функция возвращает –1 (минус 1).

Пример:

```
SAMPLES> write !,$Zdate($Horolog, 2, " январь февраль март апрель май июнь июль август сентябрь октябрь ноябрь декабрь")
          27 октябрь 2003
SAMPLES>write $zdate($horolog)
11/24/2003
SAMPLES>write $zdate($horolog, 2)
24 Nov 2003
SAMPLES>write $zdate($horolog, 10)
          1    ; понедельник
USER>write $zdate(1111111111111111111, 1,,,,,,, -1)
          -1
```

3.7.2. Функция $ZDateH

Функция $ZDateH преобразует внешний формат даты во внутренний формат. Функция также имеет девять входных аргументов, из которых только первый является обязательным, он задает дату во внешнем формате. Все остальные аргументы имеют тот же смысл, что и для функции $ZDate.

По умолчанию, $ZDateH генерирует ошибку для неправильной даты, но ее девятый аргумент позволяет подавить вывод сообщения об ошибке.

В выражениях для задания внешнего представления даты может использоваться T (для текущей даты) и добавление или вычитание дней, как видно из примеров ниже.

```
SAMPLES >write $ZDateH("24/11/2003", 4)
59497
SAMPLES >w $ZDateH("21/05/2002", 4)
58945
SAMPLES>write $ ZDateH ("21/9/1999", 5,,,,,,, -1)
-1
SAMPLES>write $ ZDateH ("21 SEP", 5,,,,,,, -1)
```

```
59433
SAMPLES>write $ ZDateH ("SEP 21 1998", 5,,,,,,, -1)
57607
SAMPLES>write $ ZDateH ("T+12", 5,,,,,,, -1)
59509
SAMPLES>write $ ZDateH ("MAT 3", 5,,,,,,, -1)
-1
```

3.8. Массивы

Работа с массивами с Caché чрезвычайно проста. Caché включает поддержку массивов различной размерности. Массивы не требуют предварительного объявления. Массив возникает только тогда, когда его элементам присваивается значение, при этом тип элемента определяется типом присваиваемого значения. В качестве индекса могут выступать переменные любых типов.

Например, массив MyVar может иметь следующие элементы:

```
Set MyVar=7
Set MyVar(22)=-1
Set MyVar(-3)=89
Set MyVar(“MyString”)=”gg”
Set MyVar(-123409, “MyString”)=”k”
Set MyVar(“MyString”, 2398)=23
Set MyVar(1.2, 3, 4, “Five”, “Six”, 7)
```

Обратите внимание, в качестве индекса массива могут использоваться целые числа как положительные, так и отрицательные, дробные числа, а также строки.

Из факта существования узла MyVar(22) вовсе не следует существование узла, скажем, MyVar(10). Т.е. многомерные массивы являются разреженными. Это означает, что в примере выше используется только семь элементов для хранения, по количеству узлов, т.к. нет необходимости объявлять массивы или определять их размеры, это дает дополнительную выгоду в памяти: место под пустые элементы не резервируется, элемент начинает использовать память по мере возникновения.

Глава 4. Объектная модель Caché

Объектная модель Caché разработана в соответствие со стандартом ODMG (Object Data Management Group). Класс – это некий шаблон, в соответствии с которым строятся конкретные экземпляры класса. Объект в Caché имеет определенный тип, т.е. является экземпляром какого-либо класса. Состояние объекта задают значения его свойств, поведение объекта задают его методы (операции). Интерфейс объекта может иметь одну или несколько реализаций. Объектная модель Caché представлена на рис. 10.

В Caché реализовано два типа классов:

- Классы типов данных (литералы).
- Классы объектов (объекты).

Классы типов данных задают допустимые значения констант (литералов) и позволяют их контролировать. Каждый объект имеет уникальный идентификатор, в то время как литерал не имеет идентификации, и от него не могут образовываться экземпляры. Классы типов данных имеют предопределенный набор методов проверки и преобразования значений атрибутов. Эти классы не могут содержать свойств.

Классы типов данных подразделяются на два подкласса типов:

- Атомарные.
- Структурированные.

Атомарными литеральными типами в Caché являются традиционные скалярные типы данных (%String, %Integer, %Float, %Date и др.). В Caché реализованы две структуры классов типов данных – список и массив. Каждый литерал уникально идентифицируется индексом в массиве или порядковым номером в списке.

Рисунок 10. Объектная модель Caché

Классы объектов подразделяются на незарегистрированные и зарегистрированные классы.

Объект может существовать в двух формах: в памяти процесса и в виде хранимой версии объекта в базе данных. В соответствии с этим различают два вида ссылок на объект: для идентификации объекта в оперативной памяти используется ссылка OREF (object reference), если же объект сохраняется в базе данных, ему назначается долговременный объектный идентификатор – OID (object ID). Объект получает OID в момент первой записи в базу данных. OID не меняется, пока существует объект. OREF – назначается объекту, когда он попадает в оперативную память. При каждой новой загрузке в оперативную память объект может получить новую объектную ссылку OREF.

Незарегистрированные классы не несут в себе предопределенного поведения, все их методы разработчик делает сам. При этом разработчик сам отвечает за назначение и поддержку уникальных идентификаторов объектов (OID) и объектных ссылок (OREF). Из-за того, что идентификация не реализована на уровне Caché, незарегистрированные классы обнаруживают ряд ограничений:

- Система не выделяет память для значений свойств объектов.
- Отсутствует автоматическая подкачка объектов, на которые делаются ссылки.
- Полиморфизм не поддерживается.
- Переменные, ссылающиеся на незарегистрированные объекты, должны явно декларироваться с указанием соответствующего класса.

Зарегистрированные классы, напротив, обеспечены обширным набором встроенных методов, например, таких как %New() для создания экземпляра объекта или %Close() – для удаления его из памяти. Это предопределенное поведение наследуется от системного класса %RegisteredObject. Экземпляры зарегистрированных классов существуют лишь временно в памяти процесса, поэтому их называют временными объектами. Противоположность им составляют встраиваемые и хранимые классы, которые могут длительное время храниться в базе данных. Создание новых зарегистрированных объектов классов и управление ими в оперативной памяти выполняет Caché. При этом программисту предоставляется объектная ссылка на экземпляр объекта, которая позволяет ссылаться на объект в оперативной памяти. Зарегистрированные классы поддерживают полиморфизм.

Сериализуемые (встраиваемые) классы наследуют свое поведение от системного класса %SerialObject и могут длительное время храниться в базе данных. Основной особенностью хранения сериализуемого класса является то, что объекты сериализуемых классов существуют в оперативной памяти как независимые экземпляры, однако могут быть сохранены в базе данных, только будучи встроенными в другой класс. Основным преимуществом использования сериализуемых классов является минимум издержек при изменении структуры классов.

Встроенные объекты представляются в оперативной памяти и базе данных совершенно по-разному:

- В оперативной памяти этот объект ничем не отличается от других объектов. На него указывает объектная ссылка – OREF.
- В базе данных встроенный объект хранится как часть объекта, в который он встроен. При этом у встроенного объекта отсутствует OID, и он не может использоваться другими объектами без содержащего его объекта.

Хранимые классы наследуют свое поведение от системного класса %Persistent. %Persistent предоставляет своим наследникам обширный набор функций, включающий: создание объекта, подкачку объекта из базы данных в память, удаление объекта и т.п. Каждый экземпляр хранимого класса имеет 2 уникальных идентификатора – OID и OREF. OID идентифицирует объект, записанный в базу данных, а OREF идентифицирует объект, который был подкачан из базы данных и находится в памяти.

Объектная модель Caché в полном объеме поддерживает все основные концепции объектной технологии:

Наследование. Объектная модель Caché позволяет наследовать классы от произвольного количества родительских классов.

Полиморфизм. Объектная модель Caché позволяет создавать приложения целиком и полностью независимыми от внутренней реализации методов объекта.

Инкапсуляция. Объектная модель Caché обеспечивает сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей. Разделяют интерфейсную часть класса и конкретную реализацию. Интерфейсная часть необходима для взаимодействия с любыми другими объектами. Реализация же скрывает особенности реализации класса, т.е. все, что не относится к интерфейсной части.

Хранимость. Система Caché поддерживает несколько видов хранения объектов: автоматическое хранение в многомерной базе данных Caché; хранение в любых структурах, определенных пользователем; хранение в таблицах внешних реляционных баз данных, доступных через шлюз Caché SQL Gateway.

Класс объектов в Caché хранится в двух формах:

- Описательная форма. Поддерживается развитый язык описания классов объектов UDL (unified definition language), построенный на базе XML (extensible markup language).
- Объектная run-time форма. Использование класса возможно только после его компиляции в объектный код.

Глава 5. Работа с классами

5.1. Правила идентификации

Каждый идентификатор должен быть уникален внутри своего контекста. Максимальные длины всех идентификаторов приведены в табл.7:

Таблица 7

Идентификатор	Ограничения
Имя пакета	<=31 символ
Имя класса	Ограничений на длину имени нет, но только первые 25 символов используются для идентификации класса.
Имя метода	<=31 символ
Имя свойства	<=31 символ
Другие члены класса	<=128 символов

Внутри Caché имена преобразуются в верхний регистр, что следует учитывать при назначении имени. Идентификатор должен начинаться с буквы, но может содержать цифры. Не может содержать пробелы или знаки пунктуации, за исключением имен пакетов, которые могут содержать символ «.» (точку). Идентификаторы, начинающиеся со знака %(процент) являются системными элементами. Например, многие методы и пакеты, обеспечивающиеся библиотекой Caché, начинаются со знака «%».

5.2. Элементы классов

Полный список элементов определения класса охватывает:

- Однозначное имя класса
- **Ключевые слова** – несколько ключевых слов, позволяющих модифицировать определение класса
- **Свойства** (состояния) – элементы данных для хранения в экземплярах класса. Могут быть константами, встроенными объектами и ссылками на хранимые объекты. Классы типов не содержат свойств.
- **Методы** – программный код, определяющий поведение объекта.
- **Параметры класса** – константы, осуществляющие настройку функциональных возможностей класса во время его компиляции (обычно с использованием генераторов методов).
- **Запросы** – операции с множеством экземпляров класса
- **Индексы** – структуры в долговременной памяти, оптимизирующие доступ к объектам.

Работа с классами выполняется в Caché Studio с помощью мастеров, которые позволяют в режиме диалога создавать классы, параметры класса, свойства, методы, запросы, индексы. Перед использованием класс необходимо откомпилировать.

Синтаксис определения класса:

```
Class <Пакет.Имя класса> Extends <Список суперклассов>
[ ключевое слово=значение, …]
{
  код
```

}

Пример 1:

Class Cinema.TicketConfirm Extends (%CSP.Page, Cinema.Utils)
{
}

где Cinema – имя пакета, TicketConfirm – класс, %CSP.Page, Cinema.Utils – суперклассы, наследником которых является класс TicketConfirm.

Пример 2:

Class Im.Class1 Extends %Persistent [classType = persistent, procedureBlock]
{ …
}

где Im – имя пакета, Class1 – имя класса, %Persistent – суперкласс, classType и procedureBlock – ключевые слова.

5.3. Имя класса

Имя класса – уникальный идентификатор класса, для облегчения восприятия допускается использование обоих регистров букв. Внутри Caché имена преобразуются в верхний регистр, что следует учитывать при назначении имени. Имена, начинающиеся со знака %(процент) – зарезервированы для элементов системных классов.

Ограничений на длину имени нет, но только первые 25 символов используются для идентификации класса.

5.4. Ключевые слова

Определение класса может быть модифицировано посредством нескольких ключевых слов. Все ключевые слова необязательны и имеют стандартное значение на тот случай, если они не заданы. Ключевые слова необходимы, прежде всего, при разработке определений классов. Примеры ключевых слов:

Abstract – означает, что нельзя создать экземпляр данного класса, используется для классов типов данных.

ClassType – определяет поведение класса. Допустимые значения: datatype, persistent, и serial. Например:

1. ClassType = datatype – означает, что это класс типа данных.
2. ClassType = persistent – хранимый класс
3. ClassType =serial – сериализуемый класс

Final – означает, что это финальный класс, т.е. от него невозможно образование подклассов.

Super – задает один или несколько суперклассов для данного класса. По умолчанию класс не имеет суперкласса.

5.5. Свойства

Свойства представляют состояние объектов. Существует два типа свойств:

- Свойства, содержащие значения
- Свойства-связи, задающие связи между объектами.

Свойство имеет однозначное имя, тип, необязательный список ключевых слов, необязательный список параметров, определенных для соответствующего типа данных.

Синтаксис:

Property <Имя свойства> as <тип> (параметры) [ключевые слова]

Примеры свойств, содержащих значения:

```
Property Total As %Float (SCALE = 2);
Property Description As %Library.String (MAXLEN = 300);
Property PlayingNow As %Library.Boolean [ InitialExpression = 1 ];
Property Rating As %Library.String (VALUELIST = ",G,PG,PG-13,R");
Property Pr As %String [ Required ];
Property SALARY As %Integer (MINVAL = "0", MAXVAL = "1000000");
Property SecretName As %String (MAXLEN = "20") [ Private, Required ];
```

Пример свойства связи:

```
Relationship Chapter As Chapter [Inverse = Book, Cardinality = Many];
```

Для свойств в Caché справедливы следующие утверждения:

- Свойства могут быть константами, ссылками на хранимые объекты, встроенными объектами, потоками данных (CLOB'ы), либо коллекциями констант или объектов.
- Со свойствами непосредственно связан набор автоматически выполняемых методов проверки и сохранения значений.
- При осуществлении доступа к значениям свойств, а также при их сохранении, возможно прозрачное изменение формата и другие изменения данных.
- Объекты, на которые делаются ссылки, равно как и встроенные объекты, при обращениях к соответствующим свойствам автоматически загружаются в память (подкачиваются).

5.5.1. Видимость свойств

Свойства могут быть определены как открытые(public) или закрытые (private). Закрытое свойство может использоваться только методами класса, к которому относится. Открытые свойства могут использоваться без ограничений. Закрытые свойства наследуются и внутри подклассов являются видимыми. В других языках программирования свойства с подобным поведением известны как «защищенные».

5.5.2. Поведение свойств

Со свойствами автоматически связано несколько методов, Эти методы не создаются простым наследованием. Каждое свойство наследует свой набор методов от двух различных источников:

- **От класса** – в зависимости от вида свойства – наследуется общее поведение. Это, например, методы Get(), Set(), а также методы проверки.
- **От класса типа данных** – в зависимости от класса типа данных наследуется поведение, зависящее от типа данных. Многие из этих методов являются генераторами методов, которые, например, позволяют задать минимальное и максимальное значение свойства целочисленного типа.

Все классы, наследуют свое поведение от системных классов. Для пользовательских типов данных поведение, унаследованное от системных классов, может быть переопределено.

5.5.3. Ключевые слова

Можно изменять определение свойства, используя одно или несколько ключевых слов. Все ключевые слова являются необязательными и имеют значение по умолчанию, если ключевое слово явно не задано. Используются следующие ключевые слова:

Calculated – задает вычисляемое свойство, которое не сохраняется. Подклассы наследуют это ключевое свойство и не могут его перекрывать.

Description – описание свойства, которое Caché использует для построения документации класса. Подклассами не наследуется.

Final – свойство, которое не может перекрываться в подклассах. По умолчанию, свойства не являются финальными. Наследуется подклассами.

InitialExpression – задает начальное значение для свойства. По умолчанию свойство не имеет начального значения. Наследуется подклассами и может перекрываться.

Private – задает свойство типа private. По умолчанию, свойства не является private. Наследуется подклассами и не может перекрываться.

Required – свойство должно получить значение перед сохранением на диске. По умолчанию свойства не являются required. Наследуется подклассами и может перекрываться.

Transient – определяет временное свойство, которое не сохраняется в базе данных. По умолчанию свойства не являются transient. Наследуется подклассами и не может перекрываться.

Type – задает имя класса, ассоциированного со свойством, это может быть класс типа данных, хранимый или встроенный класс. По умолчанию это строка (%String). Наследуется подклассами.

5.5.4. Виды свойств

Существует несколько видов свойств:

- Свойства типов данных
- Свойства ссылки на объекты
- Встроенные объекты
- Потоки данных
- Многомерные свойства
- Свойства коллекции

5.5.5. Свойства типов данных

Самый простой тип свойств это свойства типов данных. Это литеральные значения, чье поведение задается классом типа данных, ассоциированным со свойством. Например:

```
Property Count As %Integer;
```

Где Count имя свойства, %Integer – тип данных, ассоциированный со свойством.

При задании свойства можно использовать параметр, например:

```
Property Count As %Integer(MAXVAL = 100);
```

Являясь особой формой классов, типы данных имеют принципиальные отличия от классов объектов:

- от классов типов данных невозможно образование экземпляров;
- классы типов данных не могут содержать свойств;
- методы классов типов данных предоставляются программисту через интерфейс типов данных.

Классы типов данных обладают следующими функциональными возможностями:

- Отвечают за проверку значений, которая в дальнейшем может конкретизироваться с помощью параметров классов типов данных
- Они определяют преобразования между форматом хранения (в базе данных), логическим (в памяти) и форматом отображения значений.
- Обеспечивают взаимодействие с SQL, ODBC, ActiveX и Java, предоставляя в их распоряжение необходимые в каждом случае операции и методы преобразования данных.

Классы типов данных, поддерживаемые Caché приведены в табл. 8.

Таблица 8

Тип данных	Назначение	Аналогичный SQL-тип
%Binary	Двоичное значение	BINARY, BINARY VARYING, RAW, VBINARY
%Boolean	Логическое значение (0- ложь, 1- истина)	Не определен
%Currency	Валюта	MONEY, SMALLMONEY
%Date	Дата (внутренний формат)	DATE
%Float	Число с плавающей точкой	DOUBLE, DOUBLE PRECISION, FLOAT, REAL
%Integer	Целое число	BIT, INT, INTEGER, SMALLINT, TINYINT
%List	Данные в формате $List, специфическом для Caché	не определен
%Name	Имя в формате “ Фамилия.Имя”	не определен
%Numeric	Число с фиксированной точкой	DEC, DECIMAL, NUMBER, NUMERIC
%Status	Код ошибки	не определен
%String	Строка символов	CHAR, CHAR VARYING, CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, NATIONAL VARCHAR, NCHAR, NVARCHAR, VARCHAR, VARCHAR2
%Time	Время во внутреннем представлении	TIME
%TimeStamp	Отметка времени, состоящая из даты и времени	TIMESTAMP

5.5.6. Параметры

Классы типов данных поддерживают различные параметры, которые меняются от типа к типу и выполняют различные действия. Примеры параметров:

MAXLEN — задает максимальное число символов, которое может содержать строка;

MAXVAL — задает максимальное значение;

MINLEN —. задает минимальное число символов, которое может содержать строка;

MINVAL — задает минимальное значение;

FORMAT — задает формат отображения. Значение параметра соответствует опции форматирования функции **$FNUMBER**, которая выполняет форматирование.

SCALE — определяет число цифр после десятичной точки.

PATTERN — задает шаблон, которому должна соответствовать строка. Значение параметра PATTERN должно быть правильным выражением шаблона Caché.

TRUNCATE — если значение равно 1, то строка символов должна обрезаться до длины, указанной в параметре MAXLEN (умолчание), если равно 0, то нет.

ODBCDELIMITER — задает символ-разделитель в списке %List, когда он передается через ODBC.

VALUELIST – задает список значений для перечислимых свойств.

DISPLAYLIST – определяет дополнительный список значений перечислимого свойства, задает формат отображения для перечислимого свойства, используется вместе с параметром VALUELIST.

В табл.9 приведены параметры, поддерживаемые каждым классом типов данных.

Таблица 9

Класс типа данных	**Поддерживаемые параметры**
%Binary	MAXLEN, MINLEN
%Boolean	
%Currency	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, VALUELIST
%Date	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, VALUELIST
%Float	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, SCALE, VALUELIST, XSDTYPE
%Integer	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, VALUELIST, XSDTYPE
%List	ODBCDELIMITER
%Name	COLLATION, INDEXSUBSCRIPTS, MAXLEN, XSDTYPE
%Numeric	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, SCALE, VALUELIST
%Status	
%String	COLLATION, DISPLAYLIST, MAXLEN, MINLEN, PATTERN, TRUNCATE, VALUELIST, XSDTYPE
%Time	DISPLAYLIST, FORMAT, MAXVAL, MINVAL, VALUELIST
%TimeStamp	DISPLAYLIST, MAXVAL, MINVAL, VALUELIST

5.5.7. Форматы данных и методы преобразования классов типов данных

Оперируя с константами, Caché распознает различные форматы данных. В табл. 10 содержится их обзор.

Таблица 10

Формат данных	Назначение
Display	Формат отображения данных
Logical	Формат внутреннего представления данных в оперативной памяти
Storage	Формат хранения данных
ODBC	Формат представления данных для ODBC и SQL

Существуют методы для преобразования форматов, представленные в следующей таблице:

Таблица 11

Метод	Назначение
DisplayToLogical()	Преобразует отображаемые значения во внутренний формат
LogicalToDisplay ()	Преобразует значение из внутреннего формата в формат отображения
LogicalToODBC()	Преобразует значение из внутреннего формата в формат ODBC (опциональный метод)
ODBCToLogical()	Преобразует значение из формата ODBC во внутренний формат (опциональный метод)
LogicaToStorage()	Преобразует значение из внутреннего формата в формат базы данных (опциональный метод)
StorageToLogical()	Преобразует значение из формата базы данных во внутренний формат (опциональный метод)

Эти методы служат основой для создания методов свойств в соответствующих классах. В качестве примера рассмотрим класс Person со свойством DateOfBirth типа данных %Library.Date. Тогда для класса Person Caché будет автоматически создавать методы DateOfBirthDisplayToLogical(), DateOfBirthLogicalToDisplay() и т.д. и применять их для соответствующих преобразований.

5.5.8. Свойства ссылки на объекты

В Caché каждый класс можно рассматривать и как тип данных. Если при определении свойства вместо класса типа данных указать хранимый класс, тем самым будет образована ссылка на хранимый объект.

В этом случае значениями свойства будут ссылки на экземпляры соответствующего класса. Например, так можно определить свойство Doc, являющееся ссылкой на хранимый объект типа Doctor:

```
Property Doc As Doctor;
```

5.5.9. Встроенные объекты

Встроенные объекты функционируют аналогично ссылкам на объекты. Существенное различие состоит в том, что в качестве типа данных теперь задается не хранимый объект, а встраиваемый класс объектов. Следовательно, значение этого свойства – не ссылка на некий самостоятельный экземпляр объекта, а, напротив, сам этот объект, встраиваемый в объект, который на него ссылается. С точки зрения синтаксиса определение идентично ссылке на хранимый класс объектов. Например

```
Property Adr As Address;
```

Где Adr это имя свойства, Address – это встроенный объект.

5.5.10. Свойства коллекции

Свойство коллекция состоит из набора элементов одного типа. Caché поддерживает два типа коллекций: списки (List) и массивы (Array). Коллекция список состоит из упоря-

доченного набора элементов, т.е. каждому элементу ставится в соответствие его порядковый номер. Коллекция массив состоит из набора пар вида: <ключ> <значение>, упорядоченных по ключу. Например, массив прививок:

Ключ	Значение
2.04.1999	От гриппа
10.05.2001	От клещевого энцефалита
20.11.2002	От дифтерии

Коллекции могут содержать константы, встроенные объекты и ссылки на объекты.

Например, следующий код задает свойство *Colors* как коллекцию список типа %String**:**

```
Property Colors As %String [collection = list];
```

Аналогично, Caché поддерживает коллекции объектов, как, например, в следующем примере свойство Doctors является коллекцией ссылок типа массив на объекты Doctor:

```
Property Doctors As Doctor [collection = array];
```

5.5.11. Потоки данных

Потоком данных называют большое неструктурированное множество данных, которое нужно хранить в базе данных. Caché SQL обеспечивает хранение данных типа BLOBs (Binary Large Objects) и CLOBs (Character Large Objects) в базе данных. BLOBs используются для хранения двоичных данных, таких как рисунки, изображения, в то время как CLOBs используются для хранения символьной информации. Например, потоки данных могут применяться для хранения документов, технических чертежей или рисунков. Поскольку они могут быть очень велики, Caché не оперирует с потоками данных как с атомарной информацией, а предоставляет прикладным программам методы блочной записи и считывания данных. В соответствии с содержимым различают потоки данных типа CHARACTERSTREAM (состоящие из символов) BINARYSTREAM(состоящие из двоичных данных). Для манипулирования потоками используются методы класса %STREAM. Например:

```
Class MyApp.JournalEntry Extends %Persistent [ClassType = persistent]
{
Property ItemDate As %Date;
Property PictureOfTheDay As %Stream [ Collection = binarystream ];
Property EventsOfTheDay As %Stream [ Collection = characterstream ];
}
```

Класс MyApp.JournalEntry имеет два свойства типа потока: PictureOfTheDay – двоичный поток, EventsOfTheDay – символьный поток.

В зависимости от того, как определено соответствующее свойство, потоки данных автоматически сохраняются в базе данных Caché либо файлах операционной системы. Это различие достигается путем присвоения параметру STORAGE значения GLOBAL, либо FILE. Следующий параметр LOCATION применяется, когда требуется задать имя глобала либо имя каталога файловой системы, в котором должен храниться поток данных. При этом поток данных всегда представляет собой поток данных, а не какой-либо самостоятельный объект. Например:

```
Property GMemo As %Stream (STORAGE="GLOBAL", LOCATION="^MyStream") [collection = binarystream];
Property FMemo As %Stream (STORAGE="FILE", LOCATION="C:\Stream") [collection = binarystream];
```

В примере значения свойства GMemo будут сохранено в глобале ^MyStream, в то время как значения свойства FMemo будут сохраняться в файле с автоматически генерируемым именем в директории C:\Stream.

По умолчанию потоки типа GLOBAL сохраняются в глобале ^*PackageClassS*. Файловые потоки по умолчанию сохраняются в текущей директории.

5.5.12. Многомерные свойства

Свойство может быть объявлено как многомерное. Многомерное свойство ведет себя как многомерная переменная, для которой могут использоваться такие функции как $Order. Например, свойство abc задано как многомерное:

```
Property abc [ MultiDimensional ];
```

Следующий пример демонстрирует работу со свойством *abc*:

```
Set x = $DATA(obj.abc)
Set x = $DATA(obj.abc(3))
Set x = $GET(obj.abc(3))
Set x = $ORDER(obj.abc("hello",3))
KILL obj.abc
```

где obj – это oref-ссылка на объект.

Для работы с многомерными свойствами не используются методы, характерные для классов типов данных. Таким образом, для многомерного свойства с именем **Kids** не существует методов **KidsGet**, **KidsSet**, или **KidsLogicalToDisplay**. Многомерные свойства не могут отображаться в SQL таблицах.

5.6. Методы

Методы – это операции, ассоциированные с объектом. Метод выполняется внутри Caché процесса. Каждый метод может иметь имя, список формальных параметров, возвращаемое значение и программный код. Имя метода должно быть уникально внутри своего класса. Как правило, методы реализованы в виде функций или процедур на языке Caché Object Script (COS). Параметры методу могут передаваться как по ссылке, так и по значению. Тип возвращаемого значения может быть каким угодно, но, как правило, это тип %Library.Status, который возвращает информацию о статусе завершения метода.

Есть методы классов и методы экземпляров. Для задания метода класса используется ключевое слово ClassMethod, для задания метода экземпляра используется ключевое слово Method. Например, следующий синтаксис используется для задания метода класса:

```
ClassMethod <имя метода> (список параметров) [ключевые слова]
{ код
}
```

Такой синтаксис используется для задания метода экземпляра:

```
Method <имя метода> (список параметров) [ключевые слова]
{ код
}
```

5.6.1. Аргументы метода

Метод может иметь любое число аргументов. При определении метода задаются аргументы со своими типами данных. Можно задать значение по умолчанию для некоторых аргументов, так же как и аргументы, передаваемые по ссылке (по умолчанию аргументы передаются по значению). Например, метод Calculate имеет три аргумента:

```
Method Calculate(count As %Integer, name, state As
%String = "CA")
{
      // ...
}
```

где count и state объявлены как %Integer и %String, соответственно. По умолчанию, типом данных необъявленного аргумента является тип %String, в нашем случае name имеет тип %String. Т.е. можно неявно задавать типы аргументов.

5.6.2. Определение значений по умолчанию для аргументов метода

Например:

```
Method Test(flag As %Integer = 0)
{
}
```

При вызове метода без аргументов, он будет использовать значение по умолчанию.

5.6.3. Передача аргументов по ссылке

По умолчанию, аргументы передаются методу по значению. Можно использовать передачу по ссылке, используя модификатор ByRef. Например:

```
Method Swap(ByRef x As %Integer, ByRef y As %Integer)
{
      Set temp = x
      Set x = y
      Set y = temp
}
```

Можно также определить аргументы, которые возвращаются по ссылке, но не имеют входного значения, используя модификатор Output. Это эквивалентно ByRef, за исключением того, что входное значение аргумента игнорируется.

```
Method CreateObject(Output newobj As MyApp.MyClass) As
%Status
{
      Set newobj = ##class(MyApp.MyClass).%New()
      Quit $$$OK
}
```

В Caché Object Script при вызове метода, аргументы которого передаются по ссылке (ByRef или Output), нужно ставить точку перед каждым аргументом:

```
Do obj.Swap(.arg1, .arg2)
```

5.6.4. Возвращаемое значение метода

При определении метода задается, возвращает ли метод значение и если да, то задается его тип. Тип, возвращаемый методом, есть класс. Если возвращаемый тип есть класс типа данных, метод возвращает литеральное значение (такое как число или строка). В противном случае, метод возвращает ссылку (OREF) на экземпляр класса. Синтаксис:

```
Method MethodName() As ReturnType
{
}
```

где MethodName это метод, ReturnType задает имя класса, возвращаемого методом.
Например, следующий метод возвращает значение типа %Integer:

```
Method Add(x As %Integer, y As %Integer) As %Integer
{
      Quit x + y
}
```

Пример метода, который создает и возвращает экземпляр объекта:

```
Method FindPerson(id As %String) As Person
{
      Set person = ##class(Person).%OpenId(id)
      Quit person }
```

5.6.5. Видимость методов

Метод может быть public или private. Private-метод может использоваться только методами класса, которому принадлежит. Public-метод может использоваться любыми внешними методами. В Caché private-методы наследуются и являются видимыми внутри подклассов данного класса. В других языках данные методы носят название «защищенных» (protected).

5.6.6. Язык метода

При создании метода можно использовать следующие языки: “Caché” (Caché ObjectScript), “Basic”, и “Java”. По умолчанию, метод использует язык, определенный в ключевом слове его класса. Можно его перекрыть, используя ключевое слово класса Language. Например:

```
Class MyApp.Test {

/// метод Basic
Method TestB() As %Integer [ Language = basic]
{
      'код Basic
      Print "Это тест"
      Return 1
}

/// метод Cache ObjectScript
Method TestC() As %Integer [ Language = cache]
{
      // код Cache ObjectScript
      Write "Это тест"
      Quit 1
}
}
```

Внутри класса можно использовать для реализации методов и другие языки. Все методы взаимодействуют, не взирая на конкретный язык реализации.

Методы, написанные на языке Caché ObjectScript и Basic, компилируются в выполнимый Caché-код. Java-методы выполняются по-другому. Когда создается Java-представление класса (с использованием Caché Java Binding), все Java-методы генерируются непосредственно в Java-код. Это означает, что все Java-методы выполняются внутри родной JVM (виртуальной Java машины) и получают доступ к полноценной Java-среде.

5.6.7. Ключевые слова метода

Можно модифицировать определение метода посредством использования одного или более ключевых слов. Все ключевые слова не являются обязательными, при этом каждое ключевое слово имеет значение по умолчанию, если слово явно не задается.

Основные ключевые слова:

ClassMethod – определяет метод как метод класса. По умолчанию, методы являются методами экземпляров. Подклассы не могут изменить значение ключевого слова ClassMethod.

Description – обеспечивает необязательное описание метода, которое используется для построения документации класса. По умолчанию, метод не имеет описания. Подклассы не наследуют значение описания.

Final – означает, что подклассы не могут перекрыть метод, т.е. метод финальный. По умолчанию, методы не являются финальными. Ключевое слово Final наследуется подклассами.

Private – задает, что данный метод может быть вызван только методами этого же класса или подклассов (если метод не является private, тогда ограничений на вызов метода нет). Подклассы наследуют private-методы и могут изменять значение этого ключевого

слова. По умолчанию методы являются public. (Заметим, что другие языки программирования используют слово «protected» для описания такого способа видимости и используют “private” для того, чтобы помешать видимости метода из подклассов).

ReturnType – определяет тип возвращаемого значения. Установка ReturnType равным пустой строке (“ ”), означает, что нет возвращаемого значения.

Значение ReturnType наследуется подклассами и может быть изменено в подклассе. По умолчанию, метод не имеет возвращаемого значения.

SQLProc – задает метод как хранимую процедуру SQL. По умолчанию метод не является хранимой процедурой.

5.6.8. Методы класса и экземпляров

Методы экземпляров используются, если создан экземпляр объекта. Причем внутри метода сам объект доступен с помощью синтаксиса ##this. Например:

```
set sc= ##this.%Save()
```

В данном примере экземпляр объекта доступен через ##this, %Save – метод экземпляра.

Метод класса не относится к конкретному экземпляру. Метод класса вызывается без ссылки на объект, поэтому синтаксис ##this в них отсутствует. Свойства и методы экземпляра для них не имеют смысла. Методам класса доступны параметры класса.

Примеры методов класса:

%New() – создать экземпляр объекта.

%OpenId() – открыть существующий экземпляр

%DeleteId() – удалить экземпляр в базе данных.

5.6.9. Вызов метода

Для вызова метода экземпляра сначала нужно создать или открыть экземпляр объекта. Для выполнения метода используется синтаксис, подобный одной из следующих строк кода:

```
Do oref.MethodName(arglist)
Set value = oref.MethodName(arglist)
```

где oref это ссылка на подходящий объект, который уже находится в памяти, MethodName это имя метода для выполнения, arglist это список аргументов передаваемых методу, во второй строке локальной переменной присваивается значение, возвращаемое методом.

Например, для выполнения метода Admit объекта Patient используется следующий код:

```
Do pat.Admit()
```

где pat это OREF на желаемый объект Patient.

Аналогично, для выполнения метода SearchMedicalHistory объекта Patient используется следующий код:

```
Set result = pat.SearchMedicalHistory("MUMPS")
```

Где pat это OREF-ссылка на желаемый объект Patient и result содержит значение, возвращаемое методом SearchMedicalHistory.

Метод класса выполняется без создания экземпляра. Можно выполнить метод класса из метода того же класса. В этом случае, можно вызвать метод класса, как если бы вызывался метод экземпляра.

Для выполнения метода класса используется следующий синтаксис:

```
Do ##class(Classname).MethodName(arglist)
Set value = ##class(Classname).MethodName(arglist)
```

Где Classname это имя класса с подходящим методом, MethodName это имя метода выполнения, arglist это список аргументов, передаваемых методу. Value это переменная, получающая значение, возвращаемое методом.

Имя класса может содержать имя пакета. Например, вызов метода PurgeAll класса ReportCard:

Do ##class(MyApp.ReportCard).PurgeAll()

Где MyApp – это имя пакета.

Аналогично, можно выполнить метод Exists класса %File, который определяет, существует ли файл, используя следующий код:

```
Set exists = ##class(%File).Exists("\temp\file.txt")
```

Где exists равно true (1), если файл существует, или false (0), если такого файла нет.

5.6.10. Виды методов

В объектной модели Caché различают четыре разновидности методов:

- Метод-код
- Метод-выражение
- Метод-вызов
- Генератор метода

Метод-код – содержит код, написанный на языке Caché Object Script. Это просто реализация метода в виде строк кода. Этот вид метода является видом по умолчанию. Например, следующий метод определяет код метода Speak для класса Dog:

```
Class MyApp.Dog Extends %Persistent [ClassType = persistent]
{
   Method Speak() As %String
     {
             Quit "Woof, Woof"
     }
}
```

Метод-код может содержать любой правильный код на Caché ObjectScript, включая встроенный SQL и HTML, или код на Basic, в зависимости от ключевого слова. При вызове метода используется ссылка на экземпляр:

```
Write dog.Speak()    // выдаст:  Woof, Woof
```

Метод-выражение – содержит выражение на языке Caché Object Script. Передача параметров по ссылке недопустима. Не может содержать макросы, встроенный SQL, встроенный HTML.

При компиляции программы все вызовы метода заменяются этим выражением. Обычно используется для простых методов, которые, например, присутствуют в классах типов данных, когда требуется быстрая скорость вычисления выражений. Например, можно конвертировать метод Speak класса из предыдущего примера в метод-выражение:

```
Method Speak() As %String [CodeMode = expression]
{    "Woof, Woof" }
```

Этот метод можно вызывать так:

```
Write dog.Speak()
```

Будет сгенерирован следующий код:

```
Write "Woof, Woof"
```

Неплохо было бы назначить формальным параметрам метода-выражения значения по умолчанию. Тогда не нужно было бы их назначать во время выполнения.

Замечание: Caché не разрешает использование макросов или вызовов аргументов по ссылке внутри методов-выражений.

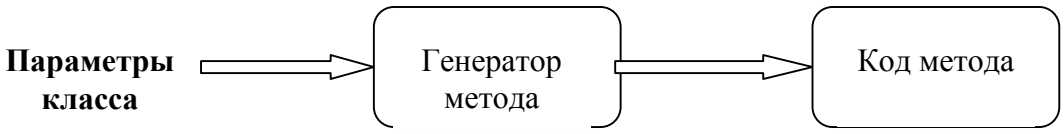
Метод-вызов – вызывает существующую программу Caché. Как правило, такие вызовы используются для того, чтобы инкапсулировать существующий код в объектно-ориентированный проект.

Определение метода как метода-вызова указывает, что при вызове метода вызывается специальная подпрограмма. Синтаксис метода вызова:

```
Method Call() [ CodeMode = call ]
```

```
{     Tag^Routine }
```
где “Tag^Routine” задает имя метки Tag внутри подпрограммы Routine.

Генераторы методов – особые методы, содержащие код для порождения кода на языке Caché Object Script. Они используются во время компиляции классов для генерации версии метода времени выполнения. При этом они могут обращаться к параметрам класса для выбора подходящего варианта генерации кода.



5.7. Параметры класса и генераторы методов

Параметры класса – это константы, то есть значения, устанавливаемые во время определения класса или в любое другое время перед компиляцией класса. Параметры класса устанавливаются для всех объектов этого класса и могут использоваться в методах класса. Во время выполнения значения параметров класса изменениям не подлежат.

Значения параметров класса наследуются и могут перекрываться в производных классах, либо пополняться дальнейшими параметрами класса. Например:

Например, класс MyApp.A имеет параметр XYZ со значением 100

```
Class MyApp.A
{ Parameter XYZ = 100; }
```

Подкласс MyApp.B может перекрыть значение этого параметра.

```
Class MyApp.B Extends MyApp.A
{ Parameter XYZ = 200; }
```

Параметры класса обычно используются при компиляции с участием генераторов методов. Генератор метода использует параметр класса с целью управления ходом генерации, например для конкретизации поведения типов данных.

Параметр класса имеет специальное поведение, когда используется с классами типов данных. С классом типов данных, параметр класса используется для обеспечения способа изменения поведения типа данных.

Например, класс типа данных **%Integer** имеет параметр класса MAXVAL, который определяет максимальное значение для свойства типа %Integer. Определим класс со свойством NumKids следующим образом:

```
Property NumKids As %Integer (MAXVAL=10);
```

Это означает, что параметр *MAXVAL для класса* %Integer должен быть установлен равным 10 для свойства NumKids.

Это работает следующим образом: методы проверки для классов типов данных все реализованы в виде генераторов методов и используют параметры классов для контроля генерации методов. В этом примере, это определение свойства генерирует код метода **NumKidsIsValidDT,** который проверяет, превышает ли значение свойства NumKids число *10.* Без использования параметров класса для подобной функциональности потребуется создать определение класса **IntegerLessThanTen.**

У любого зарегистрированного класса имеется параметр класса PROPERTYVALIDATION. Он указывает должны или нет проверяться значения свойств, и если да, то когда именно:

- 0 – означает отсутствие проверки
- 1 – предполагает проверку в момент присваивания значения.
- 2 (по умолчанию) – означает проверку при сохранении.

При каждом изменении этого параметра, сопровождаемом перекомпиляцией, генераторы методов вставляют соответствующий код проверки значений свойств каждый раз в

подходящее место. Таким образом, параметры класса служат для согласования поведения класса с конкретными потребностями данного приложения.

5.8. Запросы

Предоставляют в распоряжение разработчика операции с множествами экземпляров классов. Можно считать, что запросы образуют для объектов класса некий фильтр.

Запросы можно формулировать либо на языке Caché Object Script, либо на SQL. Результат запроса становится доступен через ResultSet – специальный интерфейс для обработки результатов запроса в прикладных программах, написанных на языке Caché Object Script, или на любом другом с использованием ActiveX, либо на языке Java. Кроме того, запросы могут быть представлены в виде хранимых процедур SQL или же представлений(View), а значит, далее обрабатываться средствами SQL.

5.9. Индексы

Индекс – это не что иное, как путь доступа к экземплярам класса. Индексы используются для оптимизации скорости выполнения запросов. Обычно индекс строится для всех экземпляров класса, включая все его подклассы. Например, индекс класса Person с подклассом Student включит в себя не только всех «просто» лиц, но также всех студентов. Индекс подкласса студент Student, напротив, будет содержать только студентов.

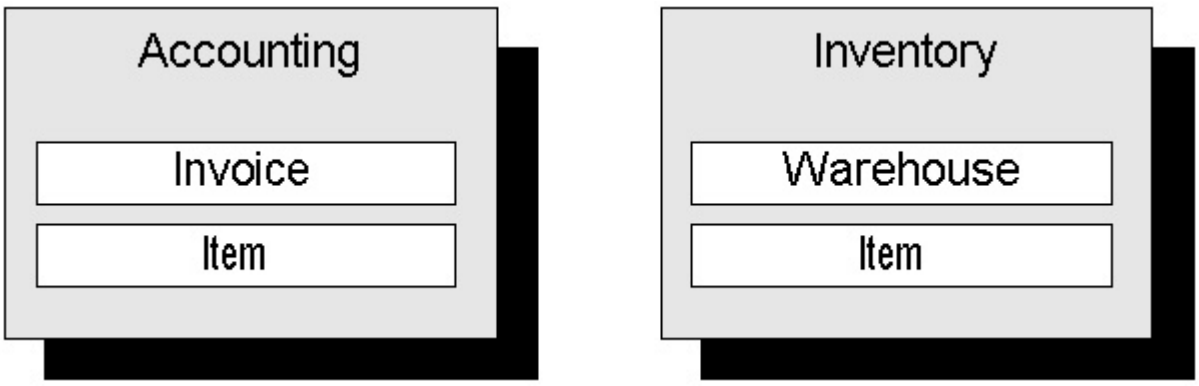
Каждый индекс создается на основе одного или нескольких свойств класса. При этом можно задать способ сортировки. Возможны следующие способы сортировки:

EXACT – без преобразований как есть;
UPPER – с преобразованием в прописные;
ALPHAUP – с преобразованием в прописные и удалением всех знаков препинания;
SPACE – принудительно устанавливается алфавитная сортировка также и для чисел;
PLUS – устанавливается числовая сортировка даже для строк символов;
MINUS – обратная числовая сортировка.

В дополнение к свойствам, по которым производится сортировка, в индексах могут содержаться другие свойства в виде полей данных. Благодаря этому их значения уже известны из индекса, что при определенных обстоятельствах позволяет избежать операций, связанных с доступом к данным.

5.10. Пакеты

Пакет это простой способ группирования взаимосвязанных классов под общим именем. Например, приложение может иметь систему счетов: “Accounting” и систему вложений: “Inventory”. Классы, реализованные в этом приложении, могут быть организованы в пакеты “Accounting” и “Inventory”.



На каждый из этих классов можно ссылаться, используя полное его полное имя, которое состоит из имени пакета и имени класса:

```
Do ##class(Accounting.Invoice).Method()
Do ##class(Inventory.Item).Method()
```

Если имя пакета может быть определено из контекста, то имя пакета может быть опущено:

```
Do ##class(Invoice).Method()
```

Пакет это просто соглашение об именах: он не обеспечивает никакой другой функциональности, кроме именования класса.

Так же как и класс, определение пакета существует внутри рабочей области Caché. Рабочая область Caché это логическое представление данных, как правило, одной базы данных.

5.10.1. Имя пакета

Имя пакета это просто строка. Оно может содержать точку ".", но больше никакой другой пунктуации. Например, полное имя класса "Test.Subtest.TestClass" означает, что «TestClass» это имя класса, а "Test.Subtest" это имя пакета. В схеме базы данных в SQL имя будет преобразовано к виду: "Test_TestClass".

Существует несколько ограничений на длину и использование имен пакетов:

- Имя не более 31 символа, включая точку, но различаются только первые 25 символов.
- Внутри области, имя каждого пакета должно быть уникально.

5.10.2. Определение пакетов

Пакеты подразумеваются при именовании классов. Самый простой способ создания пакета – задать его имя при создании нового класса в Caché Studio с использованием мастера, в котором также можно просмотреть список всех пакетов. При удалении последнего класса пакета, сам пакет автоматически удаляется. Пакет можно указать при программном создании класса. Например:

```
Class Accounting.Invoice {
}
```

Класс Invoice внутри пакета "Accounting".

Используя Caché Studio, можно просматривать и редактировать дополнительные характеристики пакетов, такие как его описание, щелкнув правой кнопкой мыши на имени пакета в окне проекта и выбрав меню «Информация о пакете».

5.10.3. Использование пакетов

Существует два пути использования имени класса:

- Использование полного имени класса вместе с именем пакета. Например: пакет.класс
- Использование короткого имени класса, позволив компилятору классов самому решать какому пакету будет принадлежать класс.

Пример использования полного имени:

```
// создать экземпляр Lab.Patient
 Set patient = ##class(Lab.Patient).%New()
```

Для того, чтобы компилятор сам мог решать к какому имени будет принадлежать класс, необходима директива #IMPORT в коде .MAC или внутри определения класса. Если такой директивы нет, то имя класса будет ассоциировано с пакетом "User" или "%Library". Например:

```
// создание экземпляра класса Person
 Set person = ##class(Person).%New()
```

```
// это тоже самое, что и
Set person = ##class(User.Person).%New()
```

5.10.4. Директива #IMPORT

Директива #IMPORT позволяет задавать пакет, где следует искать класс. Например:

```
#import Lab
    // Класс "Patient" внутри пакета Lab
 Set patient = ##class(Patient).%New()
```

Можно использовать несколько директив #IMPORT внутри программы MAC:

```
#import Lab
#import Accounting
 // Поиск класса "Patient" внутри пакетов Lab & Accounting
Set pat = ##class(Patient).%OpenId(1)
 // Поиск класса "Invoice" внутри пакетов Lab & Accounting
Set inv = ##class(Invoice).%OpenId(1)
```

Порядок следования директив #IMPORT не имеет значения и может привести к ошибке в случае двусмысленного использования. Например, если имеется одно и то же имя класса в двух разных пакетах, заданных директивой #IMPORT. Чтобы избежать этого используйте полное имя класса.

Директива IMPORT определяет, какой пакет используется для разрешения ссылки внутри определения класса. Если директива не размещена, то предполагается использование следующего кода:

```
#import User
```

Если одна директива уже определена, то директива с User автоматически не подставляется, нужно написать:

```
#import MyPackage
#import User
```

5.10.5. Пакеты и SQL

Каждый пакет соответствует SQL схеме. Например, если класс называется Team.Player (класс Player в пакете “Team”), соответствующая таблица называется “Team.Player” (таблица Player в схеме “Team”).

Пакет по умолчанию “User” соответствует схеме “SQLUser”. Следовательно, класс с именем User.Person соответствует таблице с именем SQLUser.Person.

Если имя пакета содержит точку, то она заменяется подчерком. Например: класс MyTest.Test.MyClass (класс MyClass, пакет “MyTest.Test”) становится таблицей MyTest_Test.MyClass (MyClass – таблица, “MyTest_Test” – схема).

Схема по умолчанию SQLUser. Например:

```
Select ID, Name from Person
// То же самое что и:
Select ID, Name from SQLUser.Person
```

5.10.6. Встроенные пакеты

Для совместимости с ранними версиями используются «встроенные» пакеты:

- “%Library” – любой %class без имени пакета это просто часть пакета “%Library”
- “User” – любой не-% class без имени пакета принадлежит пакету “User”

Глава 6. Работа с объектами

6.1. Создание новых объектов

Основной синтаксис для создания новых экземпляров объектов это использование метода класса %New:

```
Set oref = ##class(Classname).%New()
```

где oref ссылка OREF на новый объект и Classname это имя класса, чувствительно к регистру букв и может содержать имя пакета.

##class() – это макровызов, который используется для вызова методов класса.

Например, для создания нового объекта Person используется следующий синтаксис:

```
Set person = ##class(MyApp.Person).%New()
```

Методу %New можно передать необязательный параметр. Если он присутствует, этот аргумент передается методу-триггеру %OnNew объекта. Использование этого аргумента зависит от реализации конкретного класса.

6.2. Ссылки на экземпляр объекта

Для доступа к экземпляру объекта можно использовать следующие ссылки:

1. OID – идентификатор экземпляра объекта в базе данных
2. ID – идентификатор экземпляра внутри класса в базе данных
3. OREF – ссылка на экземпляр объекта в оперативной памяти.

Связь между OID и ID задается следующим образом:

Set OID = $LB(ID, “пакет.класс”)

Например:

Set OID=$LB(ID,”User.Books”)

6.3. Открытие объектов

Можно открыть существующий хранимый объект с помощью его идентификатора ID, используя следующий синтаксис:

```
Set oref = ##class(Classname).%OpenId(ID)
```

где oref – это переменная, содержащая OREF открываемого объекта, Classname это имя класса, ID это идентификатор открываемого объекта. Classname чувствителен к регистру букв.

Например, чтобы открыть объект Sample.Person с ID 22:

```
Set person = ##class(Sample.Person).%OpenId(22)
 Write person.Name,!
```

где person это новая объектная ссылка OREF на объект Person.

Можно также открыть объект, используя его полный OID:

```
Set oref = ##class(Classname).%Open(oid)
```

где oref содержит ссылку OREF, Classname имя класса, oid это полный OID объекта.

Обычно приложения используют метод %OpenId.

Имея открытый объект можно просматривать или изменять его свойства, используя методы описанные ниже.

6.4. Изменение свойств объекта

Наряду с методами объект имеет свойства, которые определяют его состояние. Большинство свойств имеют значения, например, такие как имя человека, его дата рождения, так же есть свойства-связи, которые задают связи между объектами.

Синтаксис для изменения свойства объекта:

```
Set oref.PropertyName = value
```

где oref содержит OREF на определенный объект, PropertyName имя свойства, value – значение. Например, назначить свойству Name объекта Person новое значение:

```
Set person.Name = "Александр"
```

где person ссылка OREF на объект Person и “Александр ” новое значение свойства Name. Этот синтаксис работает для любого типа свойств.

Внутри методов экземпляра доступ к свойствам выполняется с помощью синтаксиса:

set ..<свойство>=<значение>. Например:

set ..Adress=”Русская 35”.

Также для доступа к свойствам экземпляра можно использовать переменную ##this, которая содержит OREF- ссылку на экземпляр объекта:

Set ##this.Name=”Иванов”

Ниже описан синтаксис для работы с разными типами свойств:

- свойства ссылки на хранимые объекты
- свойства ссылки на встраиваемые объекты
- свойства коллекции: свойства-списки и свойства-массивы
- свойства потоки.

6.5. Работа со свойствами ссылками на хранимые объекты

Данные, на которые указывает свойство ссылка, существуют как независимые объекты в оперативной памяти. Для установки связи между двумя объектами, объект, на который ссылаются, должен быть связан со свойством-ссылкой другого объекта. Как только эта связь установлена, можно изменять объект, на который ссылаются либо прямо, либо используя точечный синтаксис.

6.5.1. Связывание объекта со свойством ссылкой

Существует два способа ассоциирования объекта со свойством ссылкой. Если объект в памяти, можно использовать его OREF. Если он на диске, можно использовать его объектный идентификатор (ID).

Для объекта в памяти используется следующий синтаксис:

```
Set oref.PropertyName = RefOref
```

где oref – ссылка OREF конкретного объекта, PropertyName – имя свойства-ссылки, RefOref – это ссылка OREF на объект, с которым устанавливается связь.

Например, для назначения автомобилю Car владельца, объекта типа Person, используем следующий код:

```
Set car.Owner = person
```

где car – ссылка OREF на объект Car и person ссылка OREF на объект Person.

Если объект хранится на диске и имеет значение ID, используется следующий синтаксис:

```
Do oref.PropertyNameSetObjectId(RefId)
```

где oref – ссылка OREF на определенный объект, PropertyName – имя свойства-ссылки на объект, RefId – это ID объекта, на который ссылаются. Для каждого свойства-ссылки существуют методы SetObject (использует значение OID) и SetObjectId (использует значение ID). Например, чтобы назначить владельца (ссылку на объект типа Person) конкретному автомобилю (объекту типа Car) используется следующий код:

```
Do car.OwnerSetObjectId(PersonId)
```

Где car ссылка OREF на объект Car и PersonId это значение ID сохраненного объекта Person.

Для извлечения значений свойств хранимых объектов используются методы GetObject и GetObjectId.

6.5.2. Изменение свойств объектов, на которые есть ссылка, с использованием точечного синтаксиса

Как только объект ассоциирован со ссылкой, он может быть изменен с использованием точечного синтаксиса:

```
Set oref.PropertyName.RefPropertyName = value
```

Где oref – ссылка OREF на исходный объект, PropertyName – имя свойства-ссылки на другой объект, RefPropertyName – имя свойства связываемого по ссылке объекта, которое требуется изменить, и value – новое значение.

Например, чтобы установить значение свойства Name объекта Owner через экземпляр объекта Car, используйте следующий код:

```
Set car.Owner.Name = "Иванов Александр "
```

Где car – это OREF-ссылка на объект класса Car и " Иванов Александр " значение свойства.

6.6. Работа со свойствами ссылками на встраиваемые объекты

Встраиваемые объекты существуют как самостоятельные объекты в оперативной памяти, но сохраняются как часть хранимых объектов на диске. Существует два способа изменения свойств встраиваемых объектов:

- Создать экземпляр встроенного объекта и связать этот экземпляр со значением свойства объекта-контейнера.
- С помощью точечного синтаксиса.

6.6.1. Связывание объекта со свойством встроенного объекта

Первый способ – это создать встроенный объект и заполнить его свойства:

1. Создать экземпляр встроенного объекта с помощью метода класса %New;
2. Связать OREF встроенного объекта со свойством-контейнером, используя следующий синтаксис:

```
Set oref.PropertyName = EmbeddedOref
```

где oref – ссылка OREF на объект-контейнер, PropertyName – имя свойства объекта контейнера типа встроенного объекта, EmbeddedOref – ссылка OREF на встраиваемый объект;

3. Заполнить значениями свойства объекта, используя основной синтаксис для установки значений.

Замечание: можно заполнять свойства встраиваемого объекта до связи его с объектом контейнером.

Например, создается новый объект типа Address:

```
Set address = ##class(Address).%New()
```

Можно связать его со свойством Home объекта Person:

```
Set person.Home = address
```

Где address – ссылка OREF на новый объект Address и person – это ссылка OREF на объект Person.

Установка значений свойств встраиваемого объекта выполняется с использованием основного синтаксиса:

```
Set person.Home.State = "MA"
```

6.6.2. Изменение встраиваемых объектов с использованием точечного синтаксиса

Второй способ – это задание свойств встраиваемого объекта без явного создания экземпляра:

```
Set oref.PropertyName.EmbPropertyName = value
```

где oref ссылка OREF на объект, PropertyName имя свойство объекта-контейнера типа встраиваемого объекта, EmbPropertyName имя свойства встраиваемого объекта, которое нужно изменить.

Например, изменить встроенный объект Home объекта Person можно так:

```
Set person.Home.Street = "One Memorial Drive"
Set person.Home.City = "Cambridge"
Set person.Home.State = "MA"
Set person.Home.Zip = 02142
```

где person – ссылка OREF на объект Person и его домашний адрес “One Memorial Drive, Cambridge, MA 02142”.

6.7. Работа со свойствами-списками

Список это упорядоченная порция информации. Каждый элемент имеет свою позицию в списке. Можно добавить элемент в список, либо изменить значение элемента списка. Добавление нового элемента в список как бы раздвигает его, т.е. вставка нового элемента во вторую позицию списка, приводит к тому, что второй элемент становится третьим, третий – четвертым и т.д.

Существует два типа списков свойств:

- Список типов данных
- Список объектов.

Список объектов может содержать либо встроенные объекты, либо хранимые объекты. Эти списки пополняются данными разными способами.

6.7.1. Работа со списком типов данных

Можно добавлять данные в список, используя следующий синтаксис:

```
Do oref.PropertyName.Insert(data)
```

Где oref – это ссылка OREF на объект, PropertyName – это имя свойства-списка, и data – это данные. Например, можно добавить значение “желтый” в конец списка любимых цветов, используя код:

```
Do person.FavoriteColors.Insert("желтый")
```

где person – ссылка OREF на объект Person.

Можно изменить значение элемента n, используя синтаксис:

```
Do oref.PropertyName.SetAt(data,n)
```

где oref – ссылка OREF на объект, PropertyName – это имя свойства-списка, data – это данные. Например, можно изменить список любимых цветов, например:

```
Do person.FavoriteColors.SetAt("желтый",2)
```

Т.е. вместо второго элемента будет вставлено значение "желтый".

Можно вставить данные в позицию n:

```
Do oref.PropertyName.InsertAt(data,n)
```

Вставка нового элемента списка как бы раздвигает список. Например, можно изменить список любимых цветов “красный”, “голубой”, “зеленый” следующим образом: “красный”, “желтый”, “голубой”, “зеленый” с помощью следующего оператора:

```
Do person.FavoriteColors.InsertAt("желтый",2).
```

6.7.2. Работа со списками встроенных объектов

Можно заполнять списки встроенных объектов несколькими способами. Можно создать новый объект, заполнить его данными, и затем добавить их в конец списка, используя следующий синтаксис:

```
Do oref.PropertyName.Insert(ItemOref)
```

где oref – ссылка OREF на объект-контейнер, PropertyName – имя свойства-списка контейнера, ItemOref – это ссылка OREF на встроенный объект. Например, можно добавить новую запись о вакцинации Vaccination к объекту Patient, используя следующий код:

```
Do pat.Vaccination.Insert(vac)
```

где pat – ссылка OREF на объект Patient, vac – это ссылка OREF на объект Vaccination.

Можно создать новый объект, заполнить его данными, и добавить его в позицию n, используя следующий синтаксис:

```
Do oref.PropertyName.SetAt(ItemOref,n)
```

где oref – ссылка OREF на объект-контейнер, PropertyName – это имя свойства-списка контейнера, ItemOref – это ссылка OREF на встроенный объект. Если элемент с номером n уже существует, он получает новое значение, если не существует, он вставляется в заданную позицию списка. Например, можно изменить второй элемент списка вакцинаций:

```
Do pat.Vaccination.SetAt(vac,2)
```

pat – ссылка OREF на объект Patient, vac – это ссылка OREF на новый объект Vaccination, который изменяет второй элемент списка.

Можно создать новый объект, заполнить его данными, затем вставить его в позицию n списка, используя след синтаксис:

```
Do oref.PropertyName.InsertAt(ItemOref,n)
```

Вставка нового элемента как бы раздвигает список. Например, можно добавить новую вакцинацию в третью позицию списка вакцинации Vaccination объекта Patient:

```
Do pat.Vaccination.InsertAt(vac,3)
```

где pat – ссылка OREF на объект Patient, vac ссылка OREF на объект Vaccination.

6.7.3. Работа со списками хранимых объектов

Синтаксис, описанный для заполнения списка встроенных объектов, также применим для списка хранимых объектов, если объекты находятся в памяти. К тому же, можно пополнять списки хранимых объектов объектами, не находящимися в памяти.

Работа со списками выполняется разными способами. Можно добавить объект в конец списка, используя следующий синтаксис:

```
Do oref.PropertyName.InsertObject(itemoid)
```

где oref – ссылка OREF на объект, PropertyName это имя свойства-списка, и itemoid – это OID объекта. Например, можно добавить информацию о новой собаке к списку любимцев:

```
Do per.Pets.InsertObject(DogOid)
```

Где per – это ссылка OREF на объект Person, Pets – свойство-список, DogOid это OID объекта класса Dog.

Можно добавить объект в позицию n, используя следующий синтаксис:

```
Do oref.PropertyName.SetObjectAt(ItemOid,n)
```

где oref это ссылка OREF на объект, PropertyName это имя свойства-списка, ItemOid это OID объекта. Например, можно добавить информацию о новой собаке в список, расположив ее в третьей позиции списка:

```
Do per.Pets.SetObjectAt(DogOid,3)
```

где per ссылка OREF на объект Person и DogOid это OID объекта класса Dog.

Можно вставить объект в позицию n списка:

```
Do oref.PropertyName.InsertObjectAt(ItemOid,n)
```

Вставка нового элемента как бы раздвигает список. Например, вставить информацию о новой собаке в начало списка любимцев:

```
Do per.Pets.InsertObject(DogOid,1)
```

Где per – ссылка OREF на объект Person и DogOid – это OID объекта класса Dog.

6.7.4. Изменение свойств объектов в списках

Так как объект связан с некоторой позицией в списке, можно изменить его свойства, используя следующий синтаксис:

```
Set oref.PropertyName.GetAt(n).ListPropertyName = data
```

где oref – ссылка OREF на объект, содержащий список; PropertyName это имя свойства-списка, метод GetAt находит и возвращает значение элемента в позиции n. ListPropertyName это имя свойства для изменения, data это новые данные свойства.

Например, чтобы изменить свойство name (имя) второй собаки из списка любимцев:

```
Set per.Pets.GetAt(2).Name = "Rover"
```

где per ссылка OREF на объект Person и Rover это новое имя любимца.

6.8. Работа со свойствами-массивами

Массив это неупорядоченный набор информации (в противоположность спискам, которые упорядочены). Массив содержит одну или несколько именованных пар, в которых имя элемента служит ключом, и значение элемента связано с этим ключом. Нет никакой синтаксической разницы между добавлением нового элемента и изменением существующего элемента массива.

Существует два типа свойств-массивов:

- Массивы типов данных
- Массивы объектов.

Массив объектов может содержать как встроенные объекты, так и хранимые.

6.8.1. Работа с массивами типов данных

Добавить элемент к массиву типов данных:

```
Do oref.PropertyName.SetAt(data,key)
```

где oref ссылка OREF на объект, PropertyName имя свойства-массива, data это данные, key это ключ, связанный с новым элементом.

Например, чтобы добавить новый цвет в массив Colors значений объекта Palette:

```
Do paint.Colors.SetAt("255,0,0","красный")
```

Где paint ссылка OREF на объект класса Palette, Colors это имя свойства-массива, и “красный” это ключ, позволяющий получить доступ к значению “255,0,0”.

6.8.2. Работа с массивами встроенных объектов

Чтобы добавить элемент к массиву встроенных объектов, создается новый объект, заполняется данными, затем используется следующий синтаксис для добавления объекта к массиву:

```
Do oref.PropertyName.SetAt(ElementOref,key)
```

где oref – ссылка OREF на объект, PropertyName это имя свойства массива, ElementOref это OREF нового элемента, и key это ключ, связанный с новым элементом.

Например, для добавления записи о вакцинации получаем доступ по дате к массиву Vaccination объекта Patient:

```
Do pat.Vaccination.SetAt(vac,"04/08/99")
```

где pat – ссылка OREF на объект Patient, Vaccination – свойство типа массив встроенных объектов, “04/08/99” это ключ, связанный с новым объектом, vac – это OREF ссылка на добавляемый объект.

6.8.3. Работа с массивами хранимых объектов

Синтаксис, описанный для наполнения массивов встроенных объектов, также применим к массивам хранимых объектов, если объекты уже находятся в памяти. К тому же, можно наполнять массивы хранимых объектов объектами, которые не находятся в памяти. Используется следующий синтаксис для добавления объекта к массиву:

```
Do oref.PropertyName.SetObjectAt(ElementOid,key)
```

где oref – ссылка OREF на объект, PropertyName – имя свойства массив, ElementOid это OID нового элемента, key – ключ, связанный с новым элементом. Например, для добавления информации о собаке к массиву, используем в качестве ключа имя собаки:

```
Do per.Pets.SetObjectAt(DogOid,"Джек")
```

Где per – ссылка OREF на объект Person, Pets – свойство типа массив хранимых объектов, "Джек" это ключ, используемый для доступа к собаке с OID, равным DogOid.

6.8.4. Изменение свойств объектов в массивах

Как только объект добавлен к массиву, можно изменить его свойства:

```
Set oref.PropertyName.GetAt(key).ArrayPropertyName = data
```

где oref – ссылка OREF на объект, содержащий массив, PropertyName это имя свойства массив, key идентифицирует реальные данные, ассоциированные со свойством. Например, чтобы установить тип вакцинации Vaccination, полученной пациентом на дату 02/23/98:

```
Set pat.Vaccination.GetAt("02/23/98").Type = "Polio"
```

где pat – ссылка OREF на объект Patient, метод GetAt позволяет получить доступ по ключу 2/23/98 к нужному объекту, и изменить его свойство Type на новое значение «Polio».

6.9. Обзор методов для работы с коллекциями

Кроме вышеупомянутых методов для работы с коллекциями List и Array могут использоваться методы, приведенные в табл.12.

Таблица 12

Методы коллекций

Метод	Пояснения	List	Array
Clear()	Удаляет все элементы коллекции	Да	Да
Count()	Возвращает количество элементов в коллекции	Да	Да
Define(key)	Проверяет наличие элемента с ключом key	Нет	Да
Find(element,key)	Осуществляет поиск заданного элемента коллекции, начиная с позиции ключа(key). Возвращает позицию (ключ) найденного элемента, либо «», если он не найден	Да	Да
GetAt(key)	Возвращает значение элемента в позиции с ключом key.	Да	Да
GetNext(. key)	Возвращает значение элемента, следующего после позиции ключа key, и обновляет key.	Да	Да
GetPrevious(. key)	Возвращает значение элемента, следующего перед позицией ключа key, и обновляет key.	Да	Да
Insert(element)	Добавляет элемент в конец списка	Да	Нет
InsertAt	Вставляет элемент в заданную позицию спи-	Да	Нет

(element,key)	ска. Все последующие элементы сдвигаются на одну позицию.		
InsertOrdered (element)	Вставляет элемент в позицию списка, соответствующую его значению в последовательности сортировки. Допустим только для списков со значениями константами (но не объектными ссылками и не встроенными объектами)	Да	Нет
Next(key)	Возвращает следующую позицию (ключ) после key	Да	Да
Previous(key)	Возвращает следующую позицию (ключ) перед key	Да	Да
RemoveAt(key)	Удаляет элемент в позиции с ключом key из коллекции	Да	Да
SetAt (element, key)	Присваивает новое значение элементу коллекции в позиции с ключом key.	Да	Да

6.10. Работа с потоками

Для изменения содержимого свойств потока используется следующий синтаксис:

```
Do oref.PropertyName.Write(data)
```

Где oref – ссылка OREF на объект содержащий поток, PropertyName это имя свойства типа поток, Write это метод класса %Stream (поток), запись в поток, data это данные, связанные со свойством. Например, для добавления нового значения к свойству Note объекта Visit используется следующий код:

```
Do Visit.Note.Write(note)
```

где visit – ссылка OREF на объект Visit, Note – это свойство типа поток, note это данные типа поток, например, вводимые доктором во время визита пациента.

В табл. 13 приведен ряд полезных методов для работы с потоками.

Таблица 13

Методы потоков данных

Метод	Пояснение
Write(data)	Записывает data в поток данных
Read(.length)	Считывает length символов или битов из потока данных и передает их в качестве возвращаемого значения метода. Аргумент передается по ссылке, после вызова метода он содержит количество прочитанных символов или битов. Если оно оказывается меньше запрошенного, значит, поток данных закончился.
WriteLine(data) (только для потоков символов)	Записывает data в поток данных в виде отдельной строки символов.
ReadLine(. length) (только для потоков символов)	Считывает строку символов из потока данных (вплоть до очередного ограничителя строки или до достижения длины length) и передает их в качестве возвращаемого значения метода.
Rewind()	Позиционирует указатель чтения-записи в начало потока данных.
GoToEnd()	Позиционирует указатель чтения-записи в конец потока данных.
CopyFrom(stream)	Копирует в «наш» поток данных другой поток, на который ссылается аргумент stream.
OutputToDevice()	Копирует все содержимое потока на текущее устройство.
LineTerminator	Ограничитель строки, используемый WriteLine() и ReadLine(). По умолчанию CR LF.

Свойства потока

Свойство	Пояснение
AtEnd	Истина(1), когда указатель чтения-записи находится в конце потока.

Пример. Пусть класс User.Person содержит свойство Memo, которое является свойством типа поток данных. Тогда в него можно записать текст произвольной длины.

```
Set pers=##class(User.Person).%OpenId(ID)
Do pers.Memo.WriteLine("это первая строка")
…
Do pers.Memo.WriteLine("это последняя строка")
Do pers.%Save()
Quit
```

Аналогично происходит чтение из потока

```
Set pers=##class(User.Person).%OpenId(ID)
Do pers.Memo.Rewind()
 While 'pers.Memo.AtEnd
   {  Set length=99
      write pers.Memo.ReadLine(.length),!
   }
  Quit
```

6.11. Сохранение объектов

Для сохранения хранимого объекта используется метод %Save() экземпляра:

```
Do oref.%Save()
```

где oref – ссылка oref объекта подлежащего сохранению.

Метод %Save возвращает значение, позволяющее проверить успешность выполнения метода.

```
Set sc=oref.%Save()
```

где sc – значение, возвращаемое методом %Save, oref – ссылка на объект сохранения.

Макро $$$ISOK возвращает 1, если метод отработал успешно, и 0 если неудачно. Наоборот, макро $$$ISERR возвращает 1, если неудачно, и 0 если успешно.

Если метод отработал неудачно, то вызов метода DisplayError() из библиотеки Caché $system.Status распечатает текст сообщения об ошибке.

Следующий код выполняет сохранение объекта Person со ссылкой OREF равной per и в случае ошибки выводит сообщение:

```
Set sc = per.%Save()
 If $$$ISERR(sc) {
     Do $System.Status.DisplayError()
 }
```

6.12. Удаление объектов

Можно удалить либо один объект, либо объекты одного экстента. Под экстентом понимается набор экземпляров класса и всех его подклассов.

6.12.1.Удаление одного объекта

Для удаления объекта из базы данных используются два метода %Delete и %DeleteId. Первый метод использует в качестве аргумента полный OID экземпляра, а второй ID экземпляра. Оба метода выдают информацию о том, как отработал метод.

Синтаксис метода %Delete:

```
Do ##class(Classname).%Delete(oid)
```

```
Set sc = ##class(Classname).%Delete(oid)
```

Синтаксис метода %DeleteId:

```
Do ##class(Classname).%DeleteId(id)
Set sc = ##class(Classname).%DeleteId(id)
```

Где classname это имя класса, oid это OID объекта удаления, id – ID экземпляра. В переменной sc – статус метода, информация о том, как он отработал.

Например, для удаления объекта класса Person используется код:

```
Set sc = ##class(Person).%Delete(oid)
```

где oid это OID объекта Person, который требуется удалить, переменная sc содержит код ошибки.

Для удаления объекта класса Person с ID=24 используется код:

```
Set sc = ##class(Person).%DeleteId(«24″)
```

переменная sc содержит код ошибки.

6.12.2. Удаление всех объектов экстента

Можно удалить все экземпляры класса и его подклассов, используя метод %Delete-Extent, который выдает информацию о том, как выполнилось удаление.

```
Do ##class(Classname).%DeleteExtent()
Set sc = ##class(Classname).%DeleteExtent()
```

где Classname это имя корневого класса экстента для удаления, sc это переменная, которая содержит код ошибки. Этот метод удаляет все экземпляры определенного класса, так же как и все экземпляры его подклассов.

Например, для удаления всех объектов Person, включая подклассы используется следующий синтаксис:

```
Do ##class(Person).%DeleteExtent()
```

Метод %DeleteExtent будет выполнять любые методы %OnDelete, если они присутствуют для каждого удаляемого объекта.

Можно также удалить все экземпляры класса и его подклассов, используя более быстрый, но более опасный метод %KillExtent:

```
Do ##class(Classname).%KillExtent()
```

Метод %KillExtent немедленно удаляет все данные, ассоциированные с экстентом класса, никакой откат не выполняется, и никакие ссылки не проверяются. Его следует использовать при разработке систем, не содержащих реальных данных.

6.13. Выполнение запросов

Caché обеспечивает объекты класса %ResultSet для выполнения запросов и получения результатов. Этот процесс состоит из следующих шагов:

1. Подготовка запроса
2. Выполнение запроса
3. Получение результатов запроса
4. Закрытие запроса

6.13.1. Методы запроса

Для использования запроса свяжите его с объектом класса %ResultSet. Методы объекта класса %ResultSet:

- GetParamCount() – возвращает число аргументов запроса
- GetParamName(n) – возвращает строку, имя n-ного аргумента запроса
- GetColumnCount() – возвращает целое, число столбцов в запросе
- GetColumnName(n) – возвращает строку, имя n-ного поля запроса
- GetColumnHeading(n) – возвращает строку, имя n-поля запроса.

- Prepare() – подготовить динамический запрос для выполнения, в качестве параметра методу передается строка, содержащая текст запроса.
- QueryIsValid() - проверяет допустим ли запрос, 1(true) – да, 0(false) – нет.
- Execute() – выполнить запрос, в качестве аргументов методу передаются значения параметров.
- Next() – переход к следующей строке запроса.
- Data(<имя поля>) – доступ к полям запроса по имени
- GetData(n) – доступ к полю запроса по номеру.
- Close() – закрыть запрос.

6.13.2. Подготовка запроса для выполнения

Для подготовки запроса, начнем с создания экземпляра объекта класса %ResultSet с помощью метода класса %New():

```
Set rset = ##class(%ResultSet).%New()
```

Созданный объект класса %ResultSet свяжем с запросом в классе:

```
Set rset.ClassName = class
Set rset.QueryName = query
```

где rset – ссылка OREF на объект типа %ResultSet, class это имя класса, содержащего запрос, и query – имя запроса для выполнения. Например:

```
Set rset = ##class(%ResultSet).%New(”%DynamicQuery:SQL”)
Set rset.ClassName = "Sample.Person"
Set rset.QueryName = "ByName"
Do rset.Execute()
While (rset.Next()) {
    Write rset.Data("Name"),!
}
```

Можно также использовать объект класса %ResultSet для подготовки динамического SQL-запроса, используя метод Prepare:

```
Set rset = ##class(%ResultSet).%New("%DynamicQuery:SQL")
 Do rset.Prepare("SELECT Name FROM Sample.Person WHERE Name
%STARTSWITH 'A'")
 Do rset.Execute()
 While (rset.Next()) {
     Write rset.Data("Name"),!
 }
```

6.13.3. Выполнение запросов

Для выполнения запроса используется следующий синтаксис:

```
Do rset.Execute(arglist)
```

где rset это ссылка OREF на объект класса %ResultSet, arglist это разделенный запятыми список параметров, передаваемых запросу.

6.13.4. Получение результатов запроса

Для навигации по результирующему набору данных можно использовать следующий синтаксис:

```
Do rset.Next()
```

где rset – ссылка OREF на объект класса%ResultSet.

Метод Next перемещает курсор на следующую строку данных запроса. Он возвращает 0, когда курсор достигает конца набора. Следующий код проверяет, имеет ли набор данные:

```
If ('rset.Next()) {
    Quit
```

```
}
```

Можно извлечь значение столбца текущей строки, используя свойство Data объекта %ResultSet. Data это многомерное свойство. Доступ к значению столбца можно получить по его имени. Запрос не должен иметь двух столбцов с одинаковыми именами.

```
Set code = rset.Data("Code")
```

Доступ также можно получить по номеру столбца:

```
Set data = rset.GetData(n)
```

где data – содержит данные, сохраненные в поле запроса с номером n.

Также можно читать данные определенного столбца по имени, используя метод GetDataByName:

```
Set data = rset.GetDataByName(fieldname)
```

где data содержит данные столбца с именем fieldname. Это менее эффективно, чем использовать свойство Data.

Когда все нужные данные будут извлечены из строки запроса, выполнение метода Next переводит указатель на следующую строку запроса.

6.13.5. Закрытие запроса

После извлечения нужных данных, закройте запрос:

```
Do rset.Close()
 Set sc = rset.Close()
```

sc это локальная переменная, в которой метод возвращает код выполнения. Метод Close освобождает ресурсы, занятые объектом класса %ResultSet. Как только результат предыдущего запроса закрыт, можно использовать объект класса %ResultSet для выполнения другого запроса, изменяя только значение свойств ClassName и QueryName

Вызов метода Close не является обязательным. Когда объект класса %ResultSet выходит из поля видимости, в этом случае он автоматически закрывается.

Пример использования запроса

Следующий код демонстрирует использование объекта класса %ResultSet для выполнения запроса и получения результатов:

```
// создать объект Result для запроса Sample.Person:ByName
Set rset =
##class(%ResultSet).%New("Sample.Person:ByName")
 Set columns = rset.GetColumnCount()
 // Выполнить запрос
 Set sc = rset.Execute("A")
 // Извлечение результатов
While (rset.Next()) {
     Write "------------------------",!
             // цикл по столбцам
             For col = 1:1:columns {
                     Write rset.GetColumnName(col),": "
                     Write rset.GetData(col),!
             }
 }
 Do rset.Close()
```

Можно также использовать ActiveX и Java версии объекта класса %ResultSet для выполнения этих же операций из компонент ActiveX или Java-приложений.

6.14. Примеры работы с объектами

Пример 1. Работа с хранимыми классами

Создать два хранимых класса:

1. Класс «Категории» (Category), содержит информацию о категориях, к которым могут относиться конкретные книги. Единственное свойство класса: CategoryName – название категории.

2. Класс «Книги» (Books) – содержит информацию о книгах. Где Title – название книги, Authors – список авторов, CountPage – количество страниц, Category – категория, ссылка на хранимый класс «Категории», Decsription – описание книги.

Задание: написать COS-программу, позволяющую в цикле вводить объекты класса Category и для каждого объекта Category вводить несколько объектов класса Books. Обеспечить выход из цикла.

Решение:

Определения классов:

```
Class User.Category Extends %Persistent [ ClassType = persistent, ProcedureBlock]
{
    Property CategoryName As %String;
}

Class User.Books Extends %Persistent [ ClassType = persistent, ProcedureBlock ]
{
Property Authors As %String [ Collection = list ];
Property Title As %String;
Property Category As User.Category;
Property Cena As %Numeric;
Property CountPage As %Integer;
Property Decsription As %String(MAXLEN = 100);
}
```

Программа:

```
for {
       read !, "Введите название категории:", cat
       Quit:cat=""
     set ct=##class(User.Category).%New()
     set ct.CategoryName =cat
     set xx=ct.%Save()
     for {
      read !, "Введите название книги ", title
      Quit:title=""
              set bk=##class(User.Books).%New()
              set bk.Title = title
      read !, "Введите автора: ",Author
        do bk.Authors.Insert(Author)
        set bk.Category =ct
      read !, "Введите описание: ",desc
        set bk.Decsription =desc
      read !, "Введите кол-во страниц: ",kol
        set bk.CountPage =kol
      read !, "Введите цену: ",price
        set bk.Cena =price
        set xx=bk.%Save()
        }
 }
```

Пример 2. Работа со свойствами списками

Для класса Books из предыдущего примера написать COS-программу ввода списка авторов для объекта с ID=58, учитывая, что свойство Authors – это список типа данных %String.

Решение

```
// открытие объекта с ID=58
 set bk=##class(Books).%OpenId(58)
 // очистить список авторов
 do bk.Authors.Clear()
 set i=1
 // ввести список авторов объекта с Id=58
for {
        read !,"Введите автора ",avt
        Quit:avt=""
   do bk.Authors.InsertAt(avt,i)
   set i=i+1
}
set xx=bk.%Save()

// вывести названия книг и списки авторов
Set r=##class(%ResultSet).%New("%DynamicQuery:SQL")
 Do r.Prepare("Select ID, Title from Books ")
 Do r.Execute()
  While(r.Next())
{  write !
  write ?5,r.Get("ID"),"   ",r.Get("Title")
  set id=r.Get("ID")
  set bk=##class(User.Books).%OpenId(id)
  set kol=bk.Authors.Count()

  write !,"kol=",kol

   for k=1:1:kol
  { write !,?15,bk.Authors.GetAt(k)
   }
}
```

Пример 3. Работа со встроенными классами

Создать хранимый класс Person (данные о человеке) со свойствами:

Fio – фамилия, имя, отчество человека

Adres – его адрес, это ссылка на встроенный класс Address

DataR – дата рождения.

Встроенный класс Address имеет следующие свойства:

Ind – индекс

City – город

Street – улица

House – дом

Flat – квартира.

Задание: написать COS-программу, позволяющую вводить объекты класса Person и объекты встроенного класса Address и выводить результаты на печать.

Решение:
Определение класса Person имеет вид:

```
Class User.Person Extends %Persistent [ ClassType = persistent, ProcedureBlock ]
{
Property Adres As Address;
Property DataR As %Date;
Property Fio As %String;
}
```

Определение класса Address имеет вид:

```
Class User.Address Extends %SerialObject [ ClassType = serial, ProcedureBlock ]
{
Property Flat As %Integer;
Property House As %String(MAXLEN = 5);
Property Street As %String;
Property City As %String(MAXLEN = 15);
Property Ind As %String(MAXLEN = 6);
}
```

Программа позволяет создать объект класса Person, заполнить значения его свойств и вывести его на печать. Обратите внимание: не требуется создание экземпляра встроенного класса Address. Объекты класса Person хорошо видны в SQL-менеджере.

```
// ввод объекта класса Person
 for {
          read !,"Введите ФИО ",fio
            Quit:fio=""
          set pr=##class(Person).%New()
          // ввод данных Person
          set pr.Fio=fio
           read !,"Введите дату рождения ",dataR
          set pr.DataR=$ZdateH(dataR)
           write !,"Введите адрес: "
            read !,"Индекс ",ind
              set pr.Adres.Ind=ind
             read !,"Город ",city
              set pr.Adres.City=city
             read !,"Улицу ",street
              set pr.Adres.Street=street
             read !,"Номер дома ",house
              set pr.Adres.House=house
           read !,"Номер квартиры ",flat
              set pr.Adres.Flat=flat
          set yy=pr.%Save()
 }
 // распечатать объекты класса Person
Pech Set r=##class(%ResultSet).%New("%DynamicQuery:SQL")
    Do r.Prepare("Select ID from Person")
    Do r.Execute()

   While(r.Next())
```

```
{  write !
   write ?5,"ID ",r.Get("ID")
   set id=r.Get("ID")
   set bk=##class(User.Person).%OpenId(id)
   write !,"FIO       ","DataR     ","Ind       ",
     "City      ","Street    ","House     ","Flat  "
     write !
   set k=1
   write bk.Fio
   set k=k+10
   write ?k,$ZDate(bk.DataR)
   set k=k+10
    write ?k,bk.Adres.Ind
   set k=k+10
    write ?k,bk.Adres.City
   set k=k+10
    write ?k,bk.Adres.Street
     set k=k+10
    write ?k,bk.Adres.House
     set k=k+10
    write ?k,bk.Adres.Flat
    }
```

Пример 4. Создание метода экземпляра

Написать метод экземпляра для класса Books, позволяющий изменять значения свойств заданного экземпляра. На вход методу подаются значения свойств объекта.

Решение:

```
Method AlterBook(title As %String, authors As %String, countpage As %String, desc As %String, categ As Category, cena As %Numeric)
{ // метод для изменения свойств объекта
    set ..Title=title
      do ..Authors.InsertAt(authors,1)
     do ..CategorySetObjectId(categ)
    set ..CountPage=countpage
    set ..Decsription=desc
    set ..Cena=cena
    set sc= ##this.%Save()
}
```

Вызов метода:

```
set bk=##class(Books).%OpenId(5)
do  bk.AlterBook("Три мушкетера", "Дюма А.", 560," ",1,350)
```

Пример 5. Создание метода класса

Написать метод класса для класса Books, позволяющий создавать новый объект класса и инициализировать значения его свойств. На вход методу подаются значения свойств нового объекта, на выходе OREF- ссылка на созданный экземпляр.

Решение:

```
ClassMethod AddBook(title As %String, author As %String, countpage As %String, desc As %String, categ As Category, cena As %Numeric) as Books
{ // метод класса для создания экземпляра объекта
   set book=##class(Books).%New()
```

```
    set book.Title=title
    Set book.CountPage=countpage
    Set book.Decsription=desc
    Set book.Cena=cena
     do book.CategorySetObjectId(categ)
     do book.Authors.InsertAt(author,1)
     set sc=book.%Save()
     Quit book
}
```

Вызов метода:

Set bk=##class(Books).AddBook(“Три мушкетера”, “Дюма А.”, 560,” ”,1,350)

В переменной bk – OREF-ссылка на созданный объект.

Пример 6. Использование запросов для вывода информации об экземплярах класса

Дан класс Human со свойствами ID, Fam – фамилия ,Im – имя, age – возраст, dat – дата рождения. В данном примере выполняется создание нового экземпляра объекта, ввод значений, сохранение экземпляра, а также вывод на печать значений всех объектов-экземпляров.

Решение:

Определение класса Human

```
Class User.Human Extends %Persistent [ ClassType = persistent, ProcedureBlock ]
{     Property Fam As %String;
      Property Im As %String;
      Property age As %Integer;
      Property dat As %Date;
}
```

Программа:

```
/ Создать новый объект
 write !,"Ввод новых объектов"
for {
  read !,"Введите фамилию ",fam
     Quit: fam=””
  read !," Введите имя ",im
  read !,"Возраст ",vozr
  // Создать экземпляр
  Set n=##class(User.Human).%New()
  Set n.Fam=fam, n.Im=im, n.age=vozr
  // Сохранить объект
   Set sc=n.%Save()
  }
  // Распечатать
  write !,"Объекты"
   Set r=##class(%ResultSet).%New(”%DynamicQuery:SQL”)
    Do r.Prepare("Select ID,Fam,Im,age from Human")
    Do r.Execute()
     While(r.Next())
  { write !,r.Data("ID")," ",
          r.Data("Fam")," ",r.Data("Im")," ",r.Data("age")
    }
```

Пример 7. Распечатка объектов класса, имя которого вводит пользователь

```
// Распечатать
 write "Распечатка объектов класса "
  read !,"Введите класс ", klass
 write !,"Объекты класса ", klass
 write !
  Set r=##class(%ResultSet).%New("%DynamicQuery:SQL")
   Do r.Prepare("Select * from "_klass)
   if 'r.QueryIsValid()
     { write !,"запрос не допустим"
       quit
     }
   Do r.Execute()
  // печать заголовков столбцов
  set k=1
    for i=1:1:r.GetColumnCount()
      { write ?k,r.GetColumnName(i)
        set k=k+10
      }
  // печать объектов
    While(r.Next())
  {  write !
   set k=1
    for i=1:1:r.GetColumnCount()
     { write ?k,r.Get( r.GetColumnName(i))
       set k=k+10
     }
  }
```

Пример 8. Использование запросов для вывода информации о свойствах списках

Дан класс Books – книги со свойствами Title – название книги, Description – описание, Authors – авторы. Свойство Authors является свойством типа список и описывается следующим образом:

```
Property Authors As %String [ Collection = list ];
```

Задание: распечатать все экземпляры класса Books, учитывая, что Authors это свойство типа список.

Решение

```
Set r=##class(%ResultSet).%New("%DynamicQuery:SQL")    // создать запрос
  Do r.Prepare("Select ID,Title,Authors from Books")
  if 'r.QueryIsValid()
     { write !,"запрос не допустим"
       quit
     }
  Do r.Execute()
// печать заголовков столбцов
set k=1
   for i=1:1:r.GetColumnCount()
     { write ?k, r.GetColumnName(i)
       set k=k+15
     }
// распечатать свойства ID, Title, Authors
```

```
  while (r.Next())
  {  Set IID= r.Data("ID")
     write !,IID, ?20,  r.Data("Title")
      set b=##class(Books).%OpenId(IID)
      set k=b.Authors.Count()       // так как свойство  Authors это список, метод Count
//возвращает количество элементов списка.
      for i=1:1:k
      { write ?40, b.Authors.GetAt(i),!  // доступ к элементам списка
      }
       }
```

Глава 7. Прямой доступ и глобалы

7.1. Индексированные переменные

Переменные как локальные, так и глобальные могут существовать в виде простых или индексированных структур. Глобальные переменные или глобалы, являясь хранимыми данными, создают основу так называемого прямого доступа.

Многомерность данных реализована через индексы, поэтому говорят об индексированных переменных. Табл. 14 демонстрирует различия между переменными:

Таблица 14

Тип переменной	Локальная	Глобальная
Скалярная	Name=Иванов	^Name=Иванов
Многомерная	Book(Nomer)=Солярис|Лемм С.	^Book(Nomer)=Солярис|Лемм С.

Переменная Name – скалярная, она не хранится в базе данных. Переменная ^Name – глобальная, она сохраняется в базе данных.

Переменные Book и ^Book – это индексированные переменные, в отличие от переменных Name и ^Name. Переменная ^Book является глобальной, т.е. сохраняется в базе данных.

7.2. Массивы со строковыми индексами

Индексированные переменные создаются с помощью команды Set. При этом нет необходимости в предварительном объявлении массива и его размерности. Например:

Set A=1, A(3)=2, A(3,7)=”Monday”, A(3,7,25)=”holiday”.

Будут созданы лишь элементы A, A(3), A(3,7), A(3,7,25). Это становится возможным благодаря разреженным массивам. Массив разреженный – это значит, что исходя из наличия узла A(3,7) невозможно автоматически предполагать наличие узла, скажем A(3,4). Если он был ранее явно создан, то он существует и занимает некий объем памяти. Кроме того, в качестве индекса могут использоваться числа любого типа, строки и выражения. Например:

Set month(“Сентябрь”)=9 // одномерная переменная

Set Fio=Univer(«математический», 104) // двумерная переменная

Примеры демонстрируют смешанное применение числовых и строковых индексов. Длина имени переменной плюс число всех скобок, число всех ее индексов и запятых не должно превышать 1023 символов.

7.3. Сортировка индексированных переменных

Индексированные переменные сортируются в Caché автоматически. Последовательность сортировки определяется с помощью:

- Используемого набора символов
- Правил сортировки символов этого набора.

Независимо от используемого набора символов для сортировки всегда действует правило: первыми идут канонические числовые индексы. Под каноническим числом подразумевается число без избыточных нулей и предшествующих знаков. Так числа + 1, 0.7 и 2.40 не являются каноническими. Следом за числовыми индексами сортируются все другие индексы, а также индексы представленные неканоническими числами. Например:
-100 -15 0 5 6.1 11 “!” “AA” “ZZ” “a” “z” “А” “Ш” “у” “я”.

7.4. Глобалы

Глобалы это структуры данных, как правило, многомерные, которые хранятся в базе данных и могут обрабатываться в многопользовательской среде различными процессами. Обработка глобалов в Caché весьма проста.

С помощью команды Set глобалу присваивается значение. Например:

```
Set ^Univer(“математический”)=$ListBuild(“Волков Ю.Н.”, “22-33-45”)
```

Так как значением глобала является список, для его обработки следует использовать списковые функции. Например:

```
For i =1:1:$ListLength(^Univer(“математический”))
{
 Write !, $List ( ^ Univer(“математический”) , i)
}
```

Удаление отдельных значений глобала выполняется с помощью функции Kill. Например: Kill ^ Univer(“математический”).

Потомки удаляемого узла также будут удалены. Если это не желательно, нужно применить команду ZKill, удаляющую лишь данное конкретное значение. Поле глобала можно изменить списковой функцией $List.

Рассмотрим пример прототипа системы «Структура университета». Для этого введем глобальную переменную ^Univer, которая на нулевом уровне имеет название университета:

^Univer= “N-ский Государственный университет”

На первом уровне имеет название факультета в качестве индекса, в качестве значения – ФИО декана и телефон деканата:

^Univer(<факультет>)=<ФИО декана>|<телефон деканата>

Примеры записей:

^Univer(“математический”) = Волков Ю.Н.| 22-33-45

^Univer(“физический”) = Семенов А.К.| 34-32-12

^Univer(“исторический”) = Фофанов М.Н.| 45-90-78

Значением каждой записи является список, который обрабатывается с помощью списковых функций. Выборка записей осуществляется с использованием уникального ключа, индекса.

На следующем (втором) уровне записываются название факультета и номер группы в качестве индекса и ФИО куратора в качестве значения:

^Univer(<факультет>,<группа>)=ФИО куратора.

Примеры записей:

^Univer(“математический”, 104) = Иванов А.П.

^Univer(“математический”, 105) = Розанов С.И.

^Univer(“математический”, 106) = Потапова Т.А.

(…)

Информацию о студентах сохраняем на третьем уровне:

^Univer(<факультет>,<группа>, <№ зачетки>)= ФИО студента| Адрес| Телефон

Примеры записей:

^Univer(“математический”, 104, 91232) = Иванов П.И.|Ленина 10,3| 67-56-32

^Univer(“математический”, 104, 91233) = Петров И.М.|Мира 45,9| 78-56-34

^Univer(“математический”, 104, 91234) = Сидоров В.А.|Русская 10,78| 33-45-67

Программа создания такого глобала приведена ниже:

```
Set ^Univer= "N-ский Государственный университет"
Set ^Univer("математический") = $ListBuild("Волков Ю.Н.","22-33-45")
Set ^Univer("физический") = $ListBuild("Семенов А.К.","34-32-12")
```

```
Set ^Univer("исторический") = $ListBuild("Фофанов М.Н.","45-90-78")
Set ^Univer("математический", 104) = "Иванов А.П."
Set ^Univer("математический", 105) = "Розанов С.И."
Set ^Univer("математический", 106) = "Потапова Т.А."
Set ^Univer("математический", 104, 91232) = $ListBuild("Иванов П.И.","Ленина 10,3","67-56-32")
Set ^Univer("математический", 104, 91233) = $ListBuild("Петров И.М.","Мира 45,9","78-56-34")
Set ^Univer("математический", 104, 91234) = $ListBuild("Сидоров В.А.","Русская 10,78","33-45-67")
```

Здесь необходимо сделать несколько замечаний:

- Первые индексы в каждом случае одинаковы (разумеется, нет необходимости их каждый раз сохранять). Различие проявляется на третьем уровне, сортировка выполняется в соответствие со строковой последовательностью сортировки.
- Появляется модель данных, которую можно представить в виде многомерного куба данных: название факультета, ФИО декана и телефон деканата располагаются на первом уровне, номер группы и ФИО куратора на втором, а № зачетки и информация о студенте на третьем.
- Caché предоставляет все инструменты для анализа этой модели данных.

Интересный пример прототипа экономической системы торговли текстильными товарами, приведен также в книге «СУБД Caché, объектно-ориентированная разработка приложений» [2].

7.5. Работа с глобалами

Работа с глобалами выполняется в проводнике Caché. Глобальные данные сохраняются в B*-деревьях. Дерево, которое имеет одинаковое число подуровней в каждом своем поддереве, называется сбалансированным (balanced tree, отсюда и B-дерево). Дерево, у которого каждый ключ указывает на блок данных, содержащий требуемую запись, называется B*-деревом. Оно делает возможной интеграцию области указателей и области данных.

7.6. Функции для работы с многомерными массивами

7.6.1. Функция $Data

$Data (коротко - $D) – может работать с глобальной, локальной, скалярной или индексированной переменной. Функция определяет, существует ли заданная в виде аргумента переменная и какая у нее структура данных. Функция может возвращать четыре значения. Синтаксис:

```
$DATA(variable,target)
$D(variable,target)
```

Где variable – переменная, target – необязательный, возвращает текущее значение переменной. Если переменная не определена, то остается неизменным. В табл. 15 приведены значения, возвращаемые функцией $DATA.

Таблица 15

Значение	Описание
0	Переменная с заданным индексом не существует
10 (десять)	Переменная указывает на элемент массива, который имеет потомков, но сам не имеет значения. Любая прямая ссылка на такую переменную будет вызывать ошибку <UNDEFINED>. Например, если $DATA(y) воз-

	вращает 10, то set x=y будет вызывать ошибку <UNDEFINED>.
1	Переменная существует и содержит значение, но не имеет потомков. Заметим, что пустая строка «» рассматривается как имеющая значение.
11(одиннадцать)	Переменная имеет потомков и имеет значение. Такие переменные можно использовать в выражениях.

Замечание. Значение 1 и 11 только указывают на существование значения, но не на его тип.

С помощью функции $Data можно с уверенностью сказать существует ли переменная и имеет ли она значение. Например:

If $Data(^Univer(“математический”, 104))#10 write “Такая группа есть на факультете”

If $Data((^Univer(“математический”, 110))#10=0 write “Такой группы нет”.

При делении по модулю 10 оба результата функции $Data сигнализирующие о существовании, а именно 1 или 11, преобразуются в значение 1, интерпретируемое как логическая ИСТИНА.

Аналогичного результата достигают путем целочисленного деления на 10, когда хотят установить присутствует ли значение на более высоком уровне. Как 10, так и 11 при целочисленном делении на 10 дают в результате 1, а 0 и 1 дают в результате 0.

Часто вопрос стоит так: хотелось бы знать, имеется ли значение глобала и, если оно имеется, то с этим значением работать дальше. Пример:

```
If   ($Data(^Univer(“математический”, 104))#10
{    Set Fio=^Univer(“математический”, 104)
      write Fio
 }
```

Такая проверка позволяет избежать ошибки <UNDEFINED>, если переменная отсутствует. Эта часто встречающаяся проверка может быть значительно упрощена с помощью функции $Get (сокращенно $G). Пример:

Set Fio=$Get(^Univer(“математический”, 110),””)

Если глобал существует, то значением Fio становится результат функции, если нет то пустая строка – “ ”.

Пример. В этом примере выбраны некоторые записи массива ^client, это разреженный массив, имеющий три уровня. Первый уровень содержит имя клиента, второй – адрес клиента, а третий – тип счета, номер счета, остаток по счету. Каждый клиент может иметь до четырех разных счетов. Так как ^client это разреженный массив, то могут быть неопределенные элементы на всех трех уровнях. Примеры записей массива приведены ниже:

```
^client(5)        John Jones // первый уровень
^client(5,1)      23 Bay Rd./Boston/MA 02049 // второй уровень
^client(5,1,1)    Checking/45673/1248.00   // третий
^client(5,1,2)    Savings/27564/3270.00    // третий
^client(5,1,3)    Reserve Credit/32456/125.00  // третий
^client(5,1,4)    Loan/81263/460.00          // третий
```

Программа для создания такого глобала приведена ниже:

```
Set ^client(5)="John Jones" // первый уровень
 Set ^client(5,1)=$ListBuild("23 Bay Rd.","Boston","MA 02049") // второй уровень
Set ^client(5,1,1)=$ListBuild("Checking","45673","1248.00")   // третий
Set ^client(5,1,2)=$ListBuild("Savings","27564","3270.00")    // третий
Set ^client(5,1,3)=$ListBuild("Reserve Credit","32456","125.00")  // третий
Set ^client(5,1,4)=$ListBuild("Loan","81263","460.00")          // третий
```

Нижеследующий код это подпрограмма для проверки данных каждого уровня. Она использует функцию $DATA для проверки существования текущего элемента. Если $DATA=0 на 1,2, и 3 уровнях, это означает, что текущий элемент не существует. Функция $DATA=10 если на 1 и 2 уровнях есть потомки, но нет данных. Команда WRITE на уровнях 2 и 3 использует функцию $List для извлечения информации о текущем элементе массива.

```
    Start Read !,"Введите количество записей: ",n
 Read !,"Начать с записи номер: ",s
For i=s : 1 : s + n {
// первый уровень
If $Data(^client(i)) {
If $Data(^client(i))=10 {
Write !," Имя : Данных нет"}Else {Write !," Имя : " ,^client(i)}

If $Data(^client(i,1)) {
// второй уровень
If $Data(^client(i,1))=10 {Write !,"Адрес: Нет данных" }
Else { // вывод адреса
Write !,"Адрес: ",$List(^client(i,1),1)
Write " , ",$List(^client(i,1),2)
Write " , ",$List(^client(i,1),3)
}
}

For j=1:1:4 {// третий уровень
If $Data(^client(i,1,j)) { Write !,"Тип счета: ",$List(^client(i,1,j),1)
Write ?30," Номер #: ",$List(^client(i,1,j),2)
Write ?50," Остаток: ",$List(^client(i,1,j),3)
}
}
}
}
Write !,"Конец"
Quit
```

7.6.2. Функция $Get

Функция $Get возвращает значение глобала, если он существует и пустую строку, если его значение отсутствует. Синтаксис:

$Get(<имя глобала> [,<новое значение глобала>])

В форме со вторым аргументом глобалу будет присвоено значение, заданное вторым аргументом.

Например:

```
Set Fio=$Get(^Univer(“математический”, 110),””)
```

7.6.3. Функция $Order

Функция – $Order ($O) служит для получения очередного в последовательности сортировки индекса локальной или глобальной индексированной переменной. С помощью цикла в соответствии с порядком сортировки могут быть получены все существующие

индексы данного уровня. Рассмотрим простой пример, ориентируясь на одномерную версию глобала ^Univer.

^Univer(“математический”) = Волков Ю.Н.| 22-33-45

^Univer(“физический”) = Семенов А.К.| 34-32-12

^Univer(“исторический”) = Фофанов М.Н.| 45-90-78

Команда write $Order(^Univer(“математический”)) выдаст: “физический”. Аргументом функции является конкретный узел глобала ^Univer(“математический”), при этом в качестве результата возвращается следующий существующий индекс, а именно “физический”. Если поместить его в функцию, то получим следующий индекс в порядке сортировки и т.д. вплоть до последнего. Если задать и его, получим пустую строку, которую можно использовать в качестве критерия окончания цикла. Например, следующий цикл позволяет получить список всех факультетов университета:

```
Set x=””
For {
Set x=$Order(^Univer(x))
  Quit: x=””
  Write !,x
}
```

Замечания

1. Индекс, используемый при обращении к функции, может не существовать, при этом выдается очередной существующий индекс, например:

```
> write $Order(^Univer(“математический”, 004))
> 104
```

2. Часто возникает потребность в получении не всех значений глобала, а в пределах интервала, например, мы хотели бы получить список всех групп на математическом факультете в пределах от 104 до 115.

```
Set x=104
    For  {
     Set x=$Order(^Univer(“математический”, x))
      Quit:x=””!(x>115)
    Write !, x }
```

3. Часто требуется двигаться в обратном порядке последовательности сортировки:

```
Set x=””
For  {
Set x=$Order(^Univer(x), -1)
  Quit:x=””
  Write !, x}
```

Получим:

физический
математический
исторический

4. Каким образом получают к уже заданному индексу “математический” первый индекс на втором уровне? Это делается путем использования пустой строки на втором уровне индексации:

```
> write $Order(^Univer(“математический”, ””))
> 104       // получаем первый индекс второго уровня
```

Теперь можем получить все индексы второго уровня:

```
Set Fac="математический", x=""
  For  {
  Set x=$Order(^Univer(Fac,x))
```

```
        Quit:x=""
       Write !, x
     }
```

Задача: выдать список всех студентов математического факультета группы 104.

Решение:

```
 Set Fac="математический",  Gr=104, x=""
  For  {
   Set x=$Order(^Univer(Fac,Gr,x))
  Quit:x=""
  Write "№ зачетки ",x,!
  Write "ФИО студента :",$List(^Univer(Fac,Gr,x),1) ,!
  Write "Адрес :",$List(^Univer(Fac,Gr,x),2) ,!
  Write "Телефон :",$List(^Univer(Fac,Gr,x),3) ,!
}
```

7.6.4. Функция $Query

Функция $Query возвращает имя следующего узла глобала в виде строки символов, позволяет обойти глобал сверху вниз. В отличие от функции $Order получаем полную ссылку. Например, пусть имеем глобал:

```
^Univer=”N-ский Государственный Университет”
^Univer(“математический”) = Волков Ю.Н.| 22-33-45
^Univer(“физический”) = Семенов А.К.| 34-32-12
^Univer(“исторический”) = Фофанов М.Н.| 45-90-78
^Univer(“математический”, 104) = Иванов А.П.
^Univer(“математический”, 105) = Розанов С.И.
^Univer(“математический”, 106) = Потапова Т.А.
^Univer(“математический”, 104, 91232) = Иванов П.И.|Ленина 10,3| 67-56-32
^Univer(“математический”, 104, 91233) = Петров И.М.|Мира 45,9| 78-56-34
^Univer(“математический”, 104, 91234) = Сидоров В.А.|Русская 10,78| 33-45-67
```

Тогда

```
> write $Query(^Univer(“математический”))
выдает   ^Univer("математический",104)

> write $Query(^Univer("математический",104))
выдает ^Univer("математический",104,91232)

> write $Query(^Univer("математический",104,91232))
выдает $Query(^Univer("математический",104,91233))

> write $Query(^Univer("математический",104,91233))
выдает $Query(^Univer("математический",104,91234))
```

Таким образом, обходим дерево в глубь, каждый раз получая уровень с большей размерностью, если таковой существует. Так как результатом выполнения функции является строка символов, то можно применить оператор косвенности для получения значения:

```
set x="^Univer(""математический"")"
 for { set x=$Query(@x)
   Quit:x=""
   write !,x
    }
```

Общность функций $Query и $Order состоит в том, что для получения самого первого индекса на заданном уровне используется пустая строка в позиции данного индекса, а путем задания – 1 (минус 1) в качестве второго аргумента можно «развернуть вспять» исходную последовательность сортировки.

7.6.5. Функции $QLength, $QSubscript

Функция $Qlength ($QL) отвечает на вопрос, сколько уровней имеет индексированная переменная. Т.е. если индексированную переменную записать как Name (s1, s2,…, sn), то функция вернет n. Например:

```
> write $QL("(^Univer(""математический"",104,91232)")
3
>write $QL("A")
0
```

Аргумент функции должен заключаться в кавычки. Для неиндексированной переменной вернет 0. Пустая строка получается, если аргумент не является значением имени переменной, например: write $QL(1).

Функция $QSubscript имеет два аргумента:

$QS[ubscript] (<namevalue>,<intexpr>)

namevalue – исследуемое значение имени, intexpr – число, которое задает какой индекс следует извлечь.

Если значение имени имеет форму Name (s1, s2,…, sn), то результат функции $QS(Name (s1, s2,…, sn), m) равен значению m-того индекса, если m<=n. Пример:

```
>write $QS("^Univer(""математический"",104,91232)", 2)
104
```

7.6.6. Команда Merge

Команда Merge (сокращенно M) создает полные копии индексированных переменных, независимо от того, являются ли они локальными или глобальными. Синтаксис:

Merge <Конечная переменная>=<Исходная переменная >

Где <Конечная переменная> и <Исходная переменная> – это имена локальной или глобальной, индексированной или не индексированной переменной. Конечная переменная может не существовать на момент начала работы команды. Например:

```
Merge copy=^Univer
```

В данном примере выполняется полное копирование всей структуры глобальной переменной ^Univer в локальную переменную copy. Узлы copy(i) получают значения ^Univer(i).

Если требуется скопировать лишь поддерево, то это осуществляется, например, командой:

```
Merge ^Math(“математика”)=^Univer(“математический”)
```

Для больших глобалов копирование занимает некоторое время.

7.7. Использование многомерных структур для хранения данных

Каждый хранимый класс, который использует класс %CacheStorage, может сохранять свои экземпляры внутри базы данных, используя один или несколько узлов многомерных данных (глобалей). При создании класса компилятором класса автоматически

создаются определения двух глобалей: глобали данных и индексной глобали. При этом справедливо следующее:

- Данные сохраняются в глобале, имя которого начинается с полного имени класса, т.е. пакет.класс, при этом справа к имени глобала данных добавляется буква “D”, к имени глобала индекса добавляется буква “I”.
- Данные каждого экземпляра сохраняются внутри единственного узла глобала данных, при этом значения свойств размещаются в списковых структурах.
- Каждый узел глобала данных имеет объектный идентификатор ID. По умолчанию, значения ID это целые числа, значения которых обеспечиваются вызовом функции $Increment.

При этом для работы с объектами можно использовать либо объектный доступ, либо чрезвычайно эффективный прямой доступ с помощью глобалей. Например, предположим мы определили простой хранимый класс MyApp.Person с двумя литеральными свойствами.

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
    Property Name As %String;
    Property Age As %Integer;
}
```

Если мы создадим и сохраним два экземпляра этого класса, будет создан следующий глобал данных:

```
^MyApp.PersonD = 2  // номер последнего узла  данного уровня (счетчик узлов)
 ^MyApp.PersonD(1) = $LB("",19,"Иванов")
 ^MyApp.PersonD(2) = $LB("",20,"Петров")
```

Заметим, что первый элемент списковой структуры, хранимой в каждом узле, пустой. Это зарезервировано для имени подкласса. Если определить подклассы класса Person, этот элемент будет содержать имя подкласса. Метод %OpenId, обеспечиваемый классом %Persistent, использует эту информацию для полиморфного открытия корректного типа объекта, когда несколько объектов сохраняются внутри подобных экстентов.

7.7.1. Механизм IDKEY

С помощью механизма IDKEY можно задать значение объектного ID, основываясь на значениях свойств объекта. Для этого, к определению класса добавляется определение индекса IDKEY и задаются свойство или свойства, которые обеспечивают значение ID. Заметим, что после сохранения объекта, его объектный ID не может быть изменен. Это означает, что после сохранения объекта, использующего механизм IDKEY, нельзя изменять те его свойства, на которых базируется ID.

Например, можно изменить класс **Person** из предыдущего примера, используя индекс IDKEY, который базируется на свойстве Name.

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Index IDKEY On Name [ Idkey ];
Property Name As %String;
Property Age As %Integer;
}
```

После создания и сохранения двух экземпляров класса **Person,** результирующий глобал будет следующим:

```
^MyApp.PersonD("Иванов") = $LB("",19)
 ^MyApp.PersonD("Петров") = $LB("",20)
```

Заметим, что здесь не определен счетчик узлов. Также заметим, что, так как объектный ID основывается на свойстве *Name*, предполагается, что это свойство должно быть уникальным идентификатором для каждого объекта.

Если индекс IDKEY основывается на нескольких свойствах, то каждый узел данных будет иметь составной идентификатор.

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Index IDKEY On (Name,Age) [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

В этом случае результирующий глобал будет подобен следующему:

```
^MyApp.PersonD("Иванов",19) = $LB("")
^MyApp.PersonD("Петров",20) = $LB("")
```

7.7.2. Подклассы

По умолчанию, любые поля, входящие в подкласс хранимого класса, хранятся в дополнительном узле. Имя подкласса используется как дополнительный идентификатор узла. Например, предположим, мы определили простой хранимый класс MyApp.Person с двумя литеральными свойствами:

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
    Property Name As %String;
    Property Age As %Integer;
}
```

Потом определили хранимый подкласс, MyApp.Student, который включает два дополнительных литеральных свойства:

```
Class MyApp.Student Extends Person [ClassType = persistent]
{
    Property Subject As %String;
    Property Mark As %Float;
}
```

Если мы создадим и сохраним два экземпляра класса MyApp.Student, результирующий глобал будет подобен следующему:

```
^MyApp.PersonD = 2  // счетчик узлов
^MyApp.PersonD(1) = $LB("Student",19,"Иванов")
^MyApp.PersonD(1,"Student") = $LB(3.2,"Физика")
^MyApp.PersonD(2) = $LB("Student",20,"Петров")
^MyApp.PersonD(2,"Student") = $LB(3.8,"Химия")
```

Свойства, унаследованные от класса **Person,** сохраняются в главном узле, в то время как свойства, входящие в класс **Student** сохранятся в дополнительных подузлах. Эта структура гарантирует, что данные класса **Student** могут быть использованы равнозначно и как данные класса **Person**. Например, SQL запрос, выдающий список имен всех объектов класса **Person,** будет корректно выбирать данные как **Person,** так и **Student**. Такая структура также облегчает работу компилятора классов по поддержке совместимости свойств, добавляемых как из классов, так и из подклассов.

Заметим, что первый элемент узла содержит строку “Student”, указывающую на то, что это данные класса **Student.**

7.7.3. Индексы

Хранимый класс может иметь один или несколько индексов, которые используются для выполнения операций, таких как сортировка или условный поиск. Caché SQL использует индексы для выполнения запросов. Caché Object и SQL автоматически поддерживает корректные значения индексов при операциях вставки, изменения и удаления.

7.7.4. Структуры хранения стандартных индексов

Стандартный индекс связан с упорядоченным набором одного или нескольких значений объектного ID. Например, определим хранимый класс **MyApp.Person** с двумя литеральными свойствами и индексом по свойству *Name*.

```
Class MyApp.Person Extends %Persistent [ClassType = persistent]
{
Index NameIdx On Name;

Property Name As %String;
Property Age As %Integer;
}
```

Если создать и сохранить несколько экземпляров класса **Person**, глобали данных и индексов будут следующими:

```
// глобал данных
 ^MyApp.PersonD = 3  // счетчик узлов
 ^MyApp.PersonD(1) = $LB("",34,"Jones")
 ^MyApp.PersonD(2) = $LB("",22,"Smith")
 ^MyApp.PersonD(3) = $LB("",45,"Jones")

 // глобал индекса
 ^MyApp.PersonI("NameIdx"," JONES",1) = ""
 ^MyApp.PersonI("NameIdx"," JONES",3) = ""
 ^MyApp.PersonI("NameIdx"," SMITH",2) = ""
```

Заметим следующее для индексных глобалей:

1. По умолчанию, индексный глобал размещается в глобале с именем, которое состоит из имени класса плюс буква “I” справа.
2. По умолчанию, первый элемент в таком глобале это имя индекса, это позволяет нескольким индексам храниться в подобных структурах глобалей без конфликтов.
3. Второй элемент содержит упорядоченное значение данных. По умолчанию, данные упорядочиваются в соответствии с функцией SQLUPPER, которая преобразует все символы значения в верхний регистр, и добавляет пробел в начале. Для того, чтобы данные не преобразовывались в верхний регистр, следует использовать функцию Exact. В этом случае определение индекса может выглядеть так:
   ```
   Index NameIdx On Name As Exact;
   ```
4. Третий элемент содержит объектный ID того объекта, который содержит эти данные;
5. Значения узлов пустые, все необходимые данные содержатся внутри индексов индексированного глобала. Заметим, что если определение индекса задает, что данные должны быть сохранены вместе с индексом, они будут размещены в узлах индексного глобала.

Вышеприведенный индекс содержит достаточно информации, для того чтобы удовлетворить большинство запросов, например, таких как выбрать список всех объектов **Person,** отсортированных по свойству *Name*.

Поиск в индексе может выполняться с помощью функции $Order. Например:

```
Write $Order(^MyApp.PersonI("NameIdx"," JONES",-1))
```

Т.к. такого узла нет, то будет выдан первый индекс третьего уровня, это 1. Таким образом, функция вернет значение объектного Id соответствующее значению " JONES". Следующий оператор:

```
Write $Order(^MyApp.PersonI("NameIdx"," JONES",1))
```

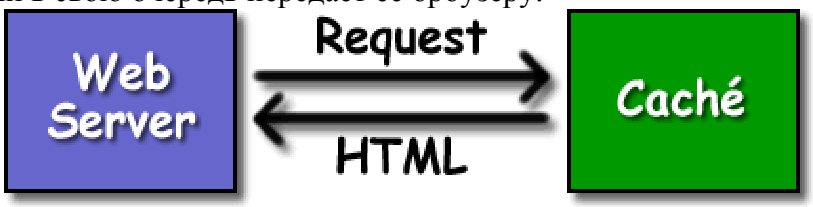
выдаст 3, это значение объектного Id второго узла, соответствующее значению " JONES", т.к. это следующий после 1 узел третьего уровня.

Глава 8. Основы технологии CSP

Технология CSP (Caché Server Pages) основной инструмент создания Web-интерфейса для информационных приложений, написанных на Caché. Технология CSP предлагает изящные средства создания быстродействующих, хорошо масштабируемых Web-приложений за короткое время. Она также упрощает дальнейшее сопровождение и развитие таких приложений.

CSP-страницы хранятся в CSP-файлах. При обращении к CSP-файлу происходит его трансляция в класс CSP, который затем компилируется с помощью компилятора Caché Server Pages. Этот процесс прозрачен для разработчика и пользователя.

Когда броузер, используя HTTP, запрашивает CSP-страницу на Web-сервере, последний, в свою очередь, запрашивает содержание страницы из базы данных Caché. Caché обрабатывает запрос, динамически генерируя HTML-страницу и передавая ее Web-серверу, который в свою очередь передает ее броузеру.



При создании CSP-файла используются:

- Выражения;
- Скрипты, выполняющие код Caché или код JavaScript;
- Методы стороны сервера;
- CSP-теги;
- Стандартные теги HTML.

8.1. Выражения Caché

Выражения Caché это заключенные в #(…)# выражения, которые заменяются вычисленными значениями в процессе генерации страниц. Например:

<html>
…
Добро пожаловать на наш сайт!
Сегодня #($Zdate($Horolog,4,” январь февраль март апрель май июнь июль август сентябрь октябрь ноябрь декабрь“)# года!
…
</html>

Конструкция #(…)# - фундаментальная часть технологии CSP. Обработка содержимого внутри круглых скобок выполняется Caché.

8.2. Скрипты, выполняющие код Caché

<script language=”Cache” runat={”server”|”compiler”}>

…
</script>

Тег <script> - стандартный тег HTML, который вызывает скрипт. Он имеет два параметра:

Language = “Cache” – определяет язык скрипта, другими значениями могут быть JavaScript, VBScript, SQL.

Runat = ”server” – определяет, что скрипт выполняется на сервере, когда страница загружена в броузер. Другим значением может быть «compiler», который заставляет выполнить код скрипта во время компиляции страницы, кроме того, может не быть никакого значения для этого параметра, что предполагает выполнение кода скрипта на клиенте, когда страница загружена в броузер.

Содержанием скрипта является текст программы на Caché Object Script, JavaScript или VBScript. Программный код выполняется каждый раз при загрузке страницы.

Пример 1. Счетчик посещений

```
<script language=Cache runat=server>
        Set ^Visit=$Get(^Visit)+1
</script>
<br>
Вы наш  #(^Visit)#  посетитель
```

Каждый раз при загрузке страницы на сервере при генерации страницы будет выполняться сценарий(скрипт), который увеличивает текущее значение глобальной переменной ^Visit на 1.

Пример 2. Вывести на Web-странице свойства объекта с id=2 класса Human:

```
<html>
 <body>
   <script language=Cache runat=server>
    set obj=##class(Human).%OpenId(2)
       write obj.Fam,"<br>"
       write obj.Im,"<br>"
    </script>
   </body>
</html>
```

Каждый раз при загрузке страницы будет открываться объект класса Human с ID равным 2. На страницу будут выводиться свойства объекта: Fam и Im, после чего объект закрывается.

8.3. Подпрограммы, вызываемые на стороне сервера #server(…)#

Пример 1.

Разработать страницу Login.csp, позволяющую проверять ввод пароля. Внешний вид страницы приведен на рис. 10.

По кнопке “Ввод” проверяется ввод пароля, при этом работает серверный метод. Если пароль неверный, то выдается сообщение. На событие onClick кнопки «Ввод» назначен обработчик события – серверный метод “CheckPassword”. Код страницы имеет вид:

```
<HTML>
<HEAD> </HEAD>
<BODY>
```

```
 <form name="Log">
  Ваше имя :<input type="TEXT" name="Name"> <br>
  Пароль   :<input type="password" name="psw"> <br>

  <input type="button" value="Ввод"  onClick="#server(..CheckPassword(
                                  self.document.Log.Name.value,
                                  self.document.Log.psw.value))#">
 </form>
</BODY>
</HTML>
 <!--   Cache-код  -- >
 <script language="Cache" Method="CheckPassword"
                  arguments="name:%String, psw:%String">
      if psw="password"
        { &javascript<alert("пароль верен");>}
     else
        { &javascript<alert("пароль не верен");>}
 </script>
```

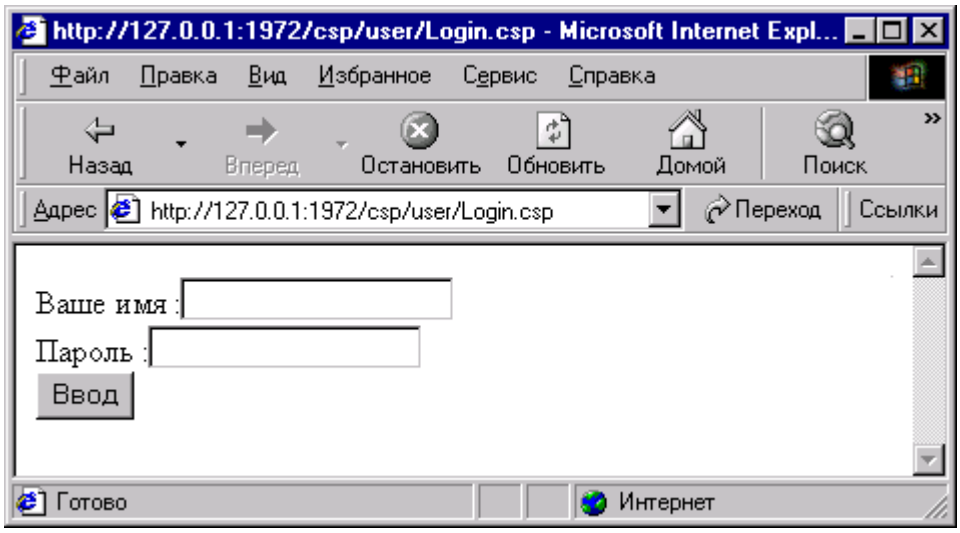


Рисунок 10. Login.csp.

Пример 2. Калькулятор

Требуется создать простой калькулятор, рассчитанный на четыре арифметические операции. Примерный вид страницы приведен на рис. 11.

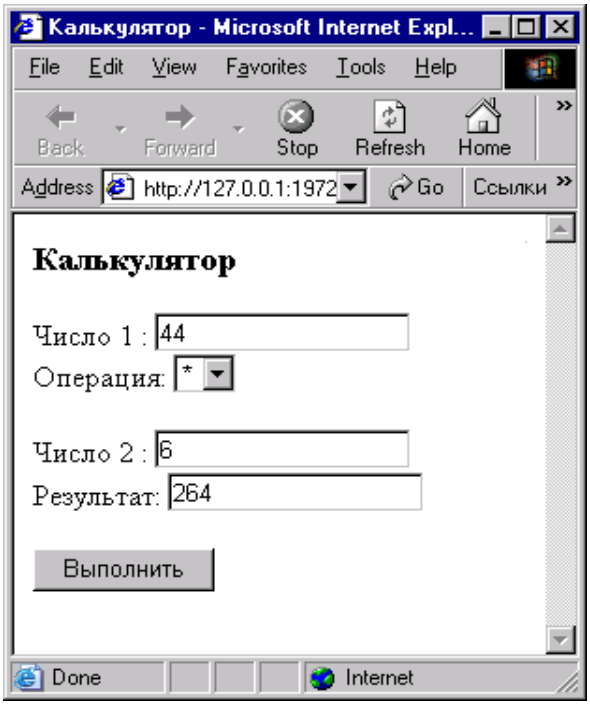


Рисунок 11. Калькулятор.

На событие onClick кнопки «Сумма» назначен обработчик события, серверный метод «Calculate». Код данной страницы имеет вид:

```
<HTML>
<HEAD>
<TITLE>          Калькулятор </TITLE>
</HEAD>
<BODY>
 <h3>  Калькулятор </h3>
  <form name="Calc">
    Число 1 : <input type="text" name="d1" value="0"><br>
    Операция: <select name="sp" size="1" >
      <option value="+">+
      <option value="-">-
      <option value="*">*
      <option value="/">/
    </select> <br> <br>
    Число 2 : <input type="text" name="d2" value="0"><br>
    Результат: <input type="text" name="Rez"><br><br>
  <input type="Button" value="Выполнить"
 onClick="#server(..Calculate(self.document.Calc.sp.value,
                 self.document.Calc.d1.value,
                 self.document.Calc.d2.value))#">
   </form>
 </BODY>
</HTML>
 <script language="Cache" Method="Calculate"
    Arguments="sp:%String, d1:%Integer, d2:%Integer">
    if sp="/",d2=0
      { &javascript<alert('нельзя делить на ноль' );>
        Quit
       }
     else
```

```
    {set sum=$Select(sp="+":d1+d2, sp="-":d1-d2,
   sp="*":d1*d2,sp="/"&& d2'=0:d1/d2)}
   &javascript<self.document.Calc.Rez.value=#(sum)#;>
 </script>
```

8.4. Теги CSP

Теги Caché имеют следующий общий синтаксис:

<CSP:XXX …>

Где XXX – это имя тега. Как встроенные, так и заказываемые теги обеспечивают разнообразные функциональные возможности.

В табл. 16 приведен перечень основных тегов:

Таблица 16

Основные теги CSP

<CSP:Object>	Связывает объект Caché с CSP-страницей
<CSP:Query>	Выполняет из CSP-страницы предопределенный запрос и именует результат
<CSP:Method>	Создает метод при определении класса CSP-страницы
<CSP:If>	Формирует содержание, основанное на условии
<CSP:Loop>	Циклическая обработка фрагмента страницы
<CSP:While>	Циклическая обработка фрагмента страницы

8.4.1. Тег <CSP:Object>

При генерации HTML-кода объектная технология требует явной установки указателя на объект, с которым будут производиться операции. На CSP-странице это выполняет тег **<CSP:Object>.** Используемый объект открывается только на время генерации страницы и в момент окончания генерации закрывается. Если этот объект потребуется в процессе работы в броузере, его необходимо вновь открыть, используя код тега **<CSP:Method>.** Атрибуты тега приведены в таблице:

Атрибут	Обязателен	Описание
CLASSNAME	Да	Имя класса Caché
NAME	Да	Имя объекта
OBJID	Нет	OID открываемого объекта. Если OBJID=” ”, создается новый объект. Если атрибут опущен, то не используется никакой объект.

Парного тега не имеет.

Пример использования тега. Открывается объект класса Human с id=34. Доступ к свойствам объекта осуществляется через атрибут name = "Hum”

```
<HTML>
<HEAD>
<!-- Пример использования тега csp:object -->
<TITLE>    использование тега csp:object  </TITLE>
</HEAD>
<BODY>
 <csp:object classname="Human"  name="Hum"  OBJID="34">
   Фамилия  : #(Hum.Fam)#   <br>
   Имя          : #(Hum.Im)#     <br>
    Возраст    : #(Hum.age)#    <br>
</BODY>
```

</HTML>

Связывание объектов с формами: CSPBIND в формах

Тег <csp:object> позволяет связать данные объекта с формой HTML через атрибут формы **CSPBind.** Таблица атрибутов связывания, которые могут появиться в теге <Form>:

Атрибут	Обязателен	Описание
CSPBind	Да	Объект, с которым связывается форма. Он должен соответствовать атрибуту NAME тега <CSP:Object>

Таблица атрибутов связывания, которые могут появиться в теге <Input>:

Атрибут	Обязателен	Описание
CSPBind	Да	Свойство или метод, к которому привязывается INPUT элемент. Должен соответствовать свойству или методу класса Caché, указанному в теге <CSP:Object>
ClassName	Может быть	Класс, из которого будут получены значения, чтобы заполнить список выбора (только для SELECT). Должен присутствовать, если присутствует CSPField или CSPQuery.
CSPField	Нет	Свойство или метод (класса, определенного **ClassName)** чьи значения появятся в списке выбора (только для SELECT). Значение по умолчанию ID.
CSPQuery	Может быть	Запрос, который будет выполнен, чтобы заполнить список выбора (только для SELECT). Должен присутствовать, если присутствует CSPField или CSPQuery.
CSPRequired	Нет	Если представлен, указывается, что значение должно быть обеспечено.
CSPValid	Нет	JavaScript код, чтобы проверить правильность содержимого элемента.

Например, требуется создать форму ввода для класса Human со свойствами Id – идентификатор объекта, Fam – фамилия ,Im – имя, age – возраст. При отображении страницы поля формы отображают текущие значения свойств конкретного объекта класса Human.

```
<HTML>
<HEAD>
<!-- Пример на использование CSPBind -->
<TITLE></TITLE>
</HEAD>
<BODY>
 <csp:object classname="Human" name="Hum" OBJID="34">
 <!--  Форма с помощью тега  CSPBind связывается с классом Human  -->
  <form CSPBind=Hum name="Vvod" >
 <!--  Каждое поле формы связывается  с конкретным свойством объекта с помощью тега CSPBind  -->
   ID          :<input CSPBind="%Id()"  Name="id"   readonly type="Text"> <br>
   Фамилия :<input CSPBind="Fam"    Name="fam"  type="Text"><br>
   Имя         :<input CSPBind="Im"     Name="im"   type="Text"><br>
```

```
 Возраст   :<input CSPBind="age"    Name="age"   type="Text"><br>
  <input type="button" value="Обновить" onclick="#server(..Alter(
    self.document.Vvod.id.value,
    self.document.Vvod.fam.value,
    self.document.Vvod.im.value,
    self.document.Vvod.age.value))#">
  </form>
</BODY>
</HTML>
 <script language="Cache" Method="Alter"
                arguments="id:%Integer, fam:%String, im:%String, age:%Integer">
        set hum=##class(Human).%OpenId(id)
        Set hum.Fam=fam,hum.Im=im,hum.age=age
        set sc=hum.%Save()
  </script>
```

8.4.2. Тег <CSP:Query>

Один из наиболее важных CSP-тегов. Позволяет найти данные, используя встроенные SQL-запросы класса. Выполняет следующие функции:

- Соединяется с определенным классом;
- Выполняет встроенный SQL-запрос;
- Возвращает результирующий набор на страницу.

Тег **<CSP:Query>** выполняет запрос и именует результирующий набор. Затем можно использовать этот набор внутри страницы по своему усмотрению. Имеет атрибуты:

Атрибут	Обязателен	Описание
CLASSNAME	Нет	Определяет имя класса Caché. Требуется, если используется предопределенный запрос, но не динамический запрос.
NAME	Да	Задает имя переменной, которая позволяет обращаться к результирующему набору на CSP-странице
QUERYNAME	Нет	Определяет имя предопределенного запроса Caché. Этот атрибут требуется, если используется предопределенный запрос, но не динамический запрос.
P1,P2,…P16	Нет	Параметры, передаваемые запросу, в порядке определения их в запросе класса.

Пример 1. Вывести все экземпляра класса Human на HTML-странице, отсортированные по свойству Fam.

Для решения этой задачи требуется создать запрос byHuman следующего вида:

```
SELECT %ID,age,dat,Fam,Im FROM Human  ORDER BY Fam.
```

В данном примере также используется тег <csp:while>, который позволяет пройти по всем записям результирующего набора. Оператор hum.Get(<имя поля>) – позволяет извлечь данные конкретного поля запроса.

```
<csp:query name="hum" classname="Human" queryname="byHuman">
  <csp:while Condition=hum.Next()>
  <tr>
    <td width="5%"> <a href=VvodHuman.csp?Id=#(hum.Get("ID"))#>
                    #(hum.Get("ID"))#</a></td>
    <td width="25%">#(hum.Get("Fam"))#</td>
```

```
  <td width="20%">#(hum.Get("Im"))#</td>
  <td width="15%">#(hum.Get("dat"))#</td>
  <td width="15%">#(hum.Get("age"))#</td></br>
 </tr>
 </csp:while>
```

В результате на странице будут отображены все объекты класса Human, отсортированные по полю Fam.

8.4.3. Тег <CSP:While >

Используется для циклической обработки фрагмента страницы. Этот тег, например, может быть использован при визуализации результатов запросов. Атрибуты тега приведены в таблице:

Атрибут	Описание
CONDITION	Логическое выражение, которое проверяется после каждой итерации. Если выражение истина, цикл продолжается.
COUNTER	Задает имя переменной-счетчика цикла, начальное значением равно 1, значение счетчика автоматически увеличивается на 1 с каждой итерацией цикла. Счетчик можно не задавать.
CURSOR	Используется только для встроенных SQL-запросов. Задает имя SQL-курсора.
INTO	Используется только для встроенных SQL-запросов. Задает разделенный запятыми список переменных, из которых будет состоять запрос.

Имеет закрывающий тег </CSP:While>.

Пример 1.

Данный пример демонстрирует использование переменной счетчика цикла.

```
<UL>
<CSP:WHILE COUNTER="i" CONDITION="i<10">
   <LI>Значение счетчика -  #(i)#.
</CSP:WHILE>
</UL>
```

Будет выведено:

- Значение счетчика - 1.
- Значение счетчика - 2.
- Значение счетчика - 3.
- Значение счетчика - 4.
- Значение счетчика - 5.
- Значение счетчика - 6.
- Значение счетчика - 7.
- Значение счетчика - 8.
- Значение счетчика - 9.
- Значение счетчика - 10.

Пример 2. Использование курсора в цикле while.

Данный пример показывает, что для вывода результатов запроса можно не использовать тег <CSP:Query>. Для задания запроса используется скрипт: <SCRIPT LANGUAGE=SQL CURSOR="query">. Для доступа к полям запроса используются переменные, задаваемые в теге <CSP:While> в атрибуте INTO.

```
<SCRIPT LANGUAGE=SQL CURSOR="query">
    SELECT Fam,Im,age FROM Human ORDER BY Fam
```

```
    </SCRIPT>
<CSP:WHILE CURSOR="query" INTO="Fam,Im,age">
    Фамилия: #(Fam)#, Имя: #(Im)#
    Возраст: #(age)# <BR>
</CSP:WHILE>
```

Пример 3. Вывести названия всех книг, которые описываются классом Books(Id, Author,Title).

Данный пример показывает использование цикла while для вывода результатов запроса:

```
<BODY>
 <script language=SQL name="query">
      SELECT * FROM Books
</script>
 <table border=1 bgcolor="">

  <csp:while  condition=query.Next()>
  <tr>
       <td>#(query.Get("Title"))#</td>
  </tr>
  </csp:while>
</table>
</BODY>
```

8.4.4. Тег <CSP:Loop>

Используется для циклической обработки фрагмента CSP-страницы. Этот тег эффективно используется при визуализации результатов запроса. Атрибуты тега приведены в таблице:

Атрибут	Обязателен	Описание
COUNTER	Да	Задает переменную счетчик цикла
STEP	Нет	Шаг цикла
FROM	Нет	Задает начальное значение переменной
TO	Нет	Задает конечное значение переменной

Имеет парный тег </CSP:LOOP>

Пример.

```
<UL>
<CSP:LOOP COUNTER="i" FROM="0" TO="6" STEP="1.5">
    <LI>Значение счетчика  #(i)#.
</CSP:LOOP>
</UL>
```

На странице будет напечатано:

- Значение счетчика 0.
- Значение счетчика 1.5.
- Значение счетчика 3.
- Значение счетчика 4.5.
- Значение счетчика 6.

8.4.5. Тег <CSP:If>

Тег <csp:IF> вместе с тегами <csp:ELSE> и <csp:ELSEIF> используется для определения направления генерации CSP-страницы. Тег имеет единственный атрибут CONDITION, значение которого является выражением ObjectScript или Basic и вычисляется на сервере во время выполнения. Если оно равно true, то содержимое тега <csp:IF> выполняется, если нет, то выполняется содержимое тега <csp:ELSE>.

Атрибут	Обязателен	Описание
CONDITION	Да	задает условие

Имеет парный тег </csp:if>

Пример 1:

```
<script language=Cache runat=server>
    set parol=%request.Get("psw")
</script>

<csp:if condition='parol="password"'>
  Пароль верен
 <csp:else>
  Пароль не верен
 </csp:if>
```

Пример 2: формирование списка категорий книг в HTML теге <select> с помощью тега <csp:if>

```
Категория:
  <select name="categ" size="1">
   <csp:while condition=Cat.Next()>
     <option value='#(Cat.Get("ID"))#'>
          <csp:if condition='book.CategoryGetObjectId()=Cat.Get("ID")'>
              <option selected>
          </csp:if>
                #(Cat.Get("CategoryName"))#
   </csp:while>
```

8.4.6. Тег <CSP:Method>

Часто, работая в броузере клиента, приходится выполнять определенные действия на сервере. Эту работу можно выполнять с помощью собственных методов классов базы данных Caché или с помощью методов, встроенных в CSP-страницы. Тег **<CSP:Method>** предоставляет возможность написать метод на Caché Object Script, который будет выполняться на сервере. Таблица атрибутов метода:

Атрибут	Обязателен	Описание
ARGUMENTS	Да	Задает формальные параметры метода
NAME	Да	Имя метода
RETURNTYPE	Нет	Тип возвращаемого методом значения

Пример. Посчитать сумму двух чисел.

```
<HTML>
<HEAD>
<!-- Пример использования тега <csp:Method> -->
<TITLE>          </TITLE>
```

```
</HEAD>
<BODY>
 Посчитать сумму двух чисел
 <form name="forma">
  ЧИСЛО 1: <input type="Text" name="C1" Value="0">
  ЧИСЛО 2:<input type="Text" name="C2" Value="0">
  РЕЗУЛЬТАТ: <input type="Text" name="rez" Value="0">
   <input type ="button" name="but" Value="Сумма"
   onclick="#server(..Summa2(self.document.forma.C1.value,
   self.document.forma.C2.value))#" >
 </form>
 <csp:method arguments="c1:%Integer,c2:%Integer" name="Summa2">
  &Javascript<alert("начало");>
    set sum=c1+c2
    &Javascript<self.document.forma.rez.value=#(sum)# ;>
 </csp:method>
</BODY>
</HTML>
```

8.4.7. Использование JavaScript-кода и кода HTML в коде Caché Object Script (COS)

В коде COS можно использовать JavaScript-код. Синтаксис использования:

```
&JavaScript < код на JavaScript ;>
&JS < код на JavaScript ;>
```

Пример:

```
&JS< alert(“начало”);>
```

Также можно использовать HTML-код. Синтаксис:

```
&HTML< код-HTML >
```

Пример:

```
If (t=0) ! (t=6)
{   &HTML< <font color=red >>
    Write ^day(t)
}
else
{ Write ^day(t)
}
```

8.5. Доступ к полям формы. Класс %CSP.Request

При получении CSP-запроса CSP-сервер создает экземпляры класса %CSP.Request, которые доступны через переменную %request.

Каждая форма имеет поля ввода, которые в соответствии со стандартом HTML оформляются в виде: «имя/значение». Переменная %request позволяет получить доступ к полю ввода формы по его имени. Например, для формы с полями ввода и именами соответственно: ID, Fam, Im, переменная %request позволит получить значения этих полей. Переменная %request это объект со своими свойствами и методами, которые приведены в следующей таблице:

Имя	Возвращаемое значение	Функция	Аргументы
Count	%Library.Integer	Число значений для заданного имени поля	Имя поля ввода (%Library.String)
Get	Соответствует аргументу	Извлекает данные, посланные в запросе	Имя поля ввода (%Library.String), необязательный второй параметр задает значение по умолчанию для поля
Kill	Нет	Удаляет поле ввода из объекта %request	Имя поля (%Library.String)
Next	%Library.String	Перебрать все поля формы, вернуть имя следующего поля	Имя предыдущего поля или «» (%Library.String)
Set	Элемент данных	Присваивает значение элементу данных поля ввода или создает новый элемент	Имя поля (%Library.String)

Например:

1) %request.Get(“Fam”) – вернет значение поля Fam.
2) %request.Set(“Length”,12) – установить значение поля ввода Length равным 12.
3) %request.Kill(“Length”) – удалить из объекта %request поле ввода Length со всеми его значениями. Если задан второй параметр, то удалятся только отдельные экземпляры поля ввода, если был множественный ввод.
4) %request.Next(“CustomerID”) – вернет имя следующего за CustomerID поля данных. %request.Next(“”) – имя первого поля.
5) %request.Count(“Authors”) – вернет количество всех значений поля Authors.

```
for I=1:1: %request.Count(“Authors”)
{   Set x=%request.Get(“Authors”, , i )
    …                          }
```

Примеры использования объекта %request

Пример 1. Дан класс Human(ID,Fam,Im, dat, age), содержащий информацию о клиентах, где Fam – фамилия, Im – имя, dat – дата рождения, age – возраст. Создать web-страницу для просмотра всех клиентов и web-страницу для изменения свойств клиентов.

Решение. Создать страницу BrowHuman.csp, которая позволяет просматривать информацию обо всех клиентах в виде таблицы. Внешний вид страницы приведен на рис. 12.

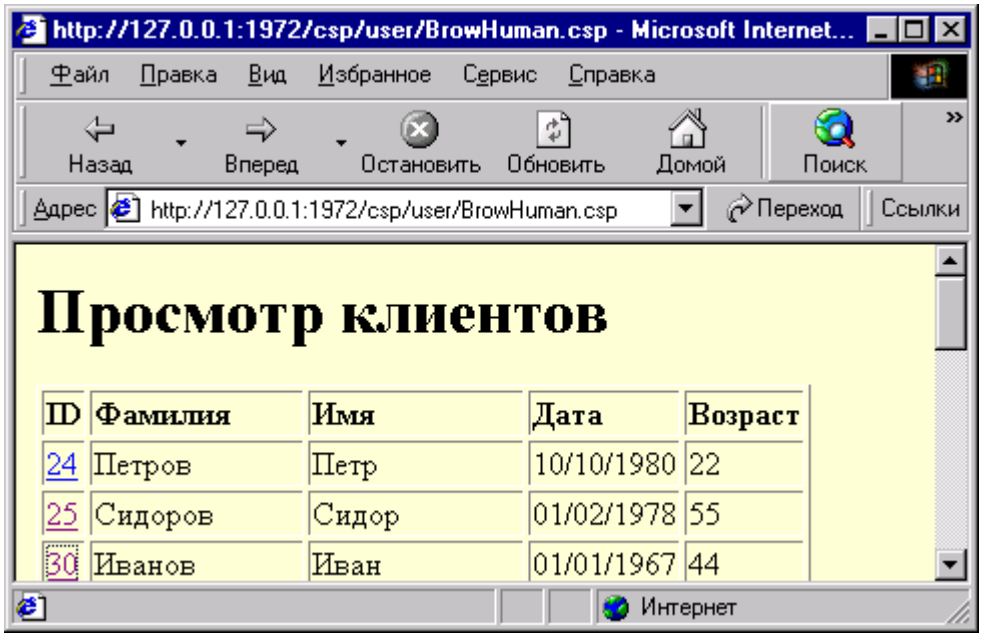


Рисунок 12. BrowHuman.csp

В поле ID формы имеется гипер-ссылка, по которой открывается другая web-страница – CSPBind1.csp. CSPBind1.csp является формой отображения и изменения полей нужного объекта. Она должна получить от BrowHuman.csp значение ID редактируемого объекта, см. рис 13.

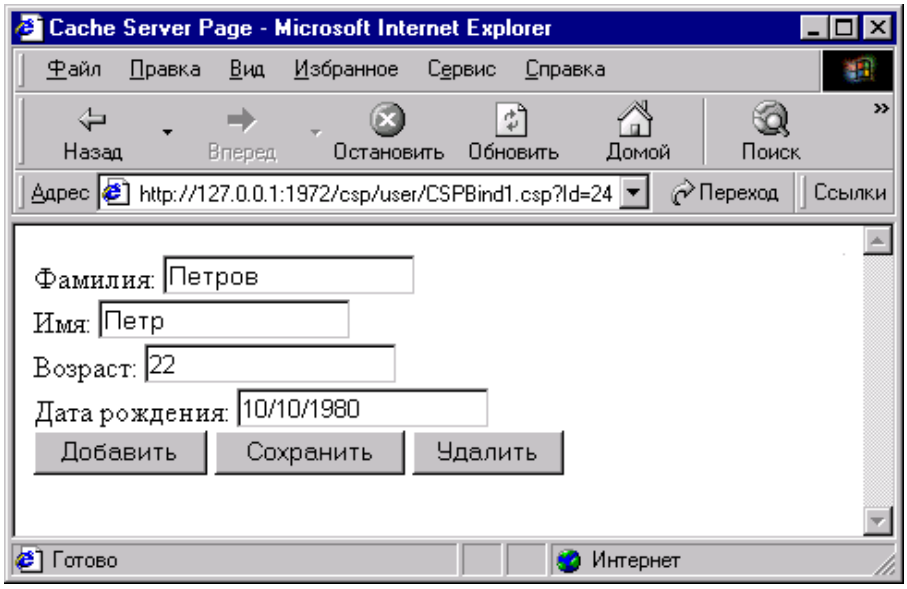


Рисунок 13. Форма CSPBind1.csp.

На странице BrowHuman.csp значение ID передается странице CSPBind1.csp с помощью следующего кода:

<a href=CSPBind1.csp?Id=#(hum.Get("ID"))#>

В свою очередь на странице CSPBind1.csp доступ к переданному значению ID выполняется с помощью объекта %request, код приведен ниже:

<csp:object classname="Human" name="obj" objid=#(%request.Get("Id"))#>

Исходный код страниц приводится ниже.

Код формы BrowHuman.csp:

```
<head> </head>
 <body bgcolor="#FFFDD">
  <h1>Просмотр клиентов</h1>
```

```
            <table border="1" width="70%">
 <tr>
   <td width="5%">  <b>ID</b> </td>
   <td width="25%"> <b>Фамилия</b> </td>
   <td width="20%">  <b>Имя</b> </td>
   <td width="15%">  <b>Дата</b> </td>
   <td width="15%">  <b>Возраст</b> </td>
 </tr>
     <csp:query name="hum" classname="Human" queryname="byHuman">
 <csp:while Condition=hum.Next()>
 <tr>
 <td width="5%">
 <a href=CSPBind1.csp?Id=#(hum.Get("ID"))#>
                    #(hum.Get("ID"))#</a></td>
   <td width="25%">#(hum.Get("Fam"))#</td>
   <td width="20%">#(hum.Get("Im"))#</td>
   <td width="15%">#(hum.Get("dat"))#</td>
   <td width="15%">#(hum.Get("age"))#</td>
 </tr>
 </csp:while>
 </table>
 </body>
```

Код формы CSPBind1.csp:

```
<HTML>
<HEAD>
<TITLE>  </TITLE>
</HEAD>
<BODY>
 <csp:object classname="Human" name="obj"
     objid=#(%request.Get("Id"))#>
<form name=Person cspbind="obj">
   Фамилия:  <input cspbind="Fam" type="text" name="Fam" ><br>
   Имя:        <input cspbind="Im"  type="text" name="Im" ><br>
   Возраст:   <input cspbind="age" type="text" name="age" ><br>
   Дата рождения: <input cspbind="dat" type="text" name="dat" ><br>
<input name="New" type="Button" value="Добавить"
                                    OnClick='Person_new();'>
<input name="Save" type="Button" value="Сохранить"
                                    OnClick='Person_save();'>
<input name="Del" type="Button" value="Удалить"
     OnClick="#server(..Del(#(%request.Get("Id"))#))#">
 </form>
</BODY>
</HTML>
 <script language="Cache" Method="Del" arguments="id:%Integer">
    set cc=##class(Human).%DeleteId(id)
 </script>
```

С помощью элемента objid=#(%request.Get("Id"))# форма CSPBind1.csp получает значение поля ID и правильно отображает поля нужного объекта.

Форма CSPBind1.csp также интересна тем, что она использует атрибут CSPBind тега Form, т.е. связывает поля формы с полями объекта. Если осуществляется привязка объекта к форме CSP-страницы тегом <CSP:Object>, Caché автоматически создает методы formname_new() и formname_save(), где formname – это имя формы, методы используются для добавления и сохранения объектов формы.

Пример 2. Обработка запросов пользователей с использованием %request

Например, требуется выполнить поиск клиента по фамилии, для этого создается страница «Search.csp». Внешний вид страницы «Search.csp» приведен на рис. 14.

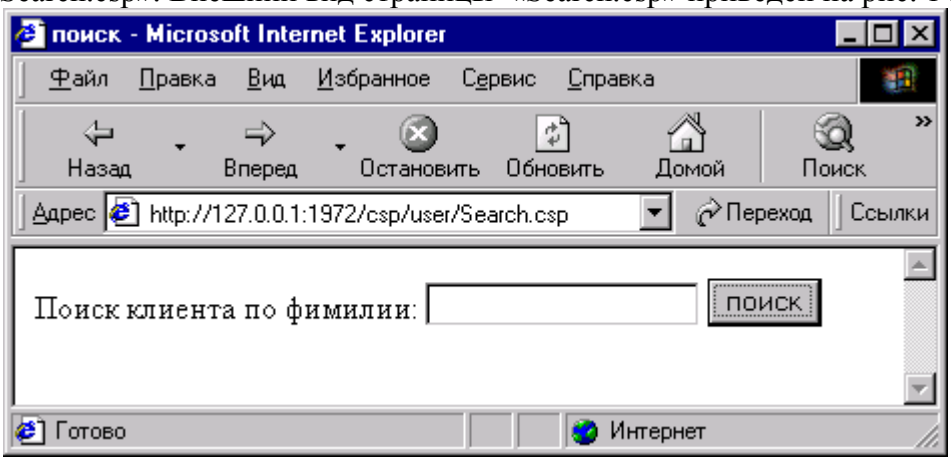


Рисунок 14. Форма поиска Search.csp.

Код страницы Search.csp приведен ниже:

```
<HTML>
<HEAD>
<TITLE>          поиск </TITLE>
</HEAD>
<BODY bgcolor="#FFFFFF" >
  <form name="Search" Action="PersonSearch.csp">
    Поиск клиента по фимилии:
  <input type="Text" name="SearchFor">
  <input type="submit" value="поиск">
  </form>
</BODY>
</HTML>
```

При нажатии на кнопку "submit" («поиск») открывается web-страница PersonSearch.csp, которой передается объект %request, метод Get() которого позволяет получить значение поля "SearchFor". При загрузке страницы выполняется запрос Q2, содержащий параметр. Текст запроса Q2 приведен ниже:

```
Query Q2(P1 As %Integer = 44) As %SQLQuery(CONTAINID = 1)
{
SELECT %ID,age,dat,Fam,Im FROM Human
 WHERE (Fam = :P1)
 ORDER BY Fam
}
```

Код страницы PersonSearch.csp приведен ниже:

```
<HTML>
<HEAD>
<TITLE>          </TITLE>
```

```
</HEAD>
<BODY bgcolor="#BBFFFF">
 <csp:query name="query" classname="Human"
                              queryname="Q2"      P1='#(%request.Get("SearchFor"))#'>
  req=  #(%request.Get("SearchFor"))#  <br>
  <csp:while Condition="query.Next()">
   <a href='Person.csp?oid=#(query.Get("ID"))#'></a>
            Фамилия :  #(query.Get("Fam"))# <br>
            Имя     :  #(query.Get("Im"))#  <br>
            Возраст :  #(query.Get("age"))# <br>
            Дата рождения :#(query.Get("dat"))# <br>
    <br>
  </csp:while>

</BODY>
</HTML>
```

8.6. Объект %session

При активизации сеанса работы с пользователем создается переменная %session, объект класса %CSP.Session. В объекте %session могут храниться переменные, ассоциированные с пользователем, используемые для управления сеансом. Например, с помощью сессии можно передавать значения переменных между страницами. Дело в том, что одна из основных трудностей, при разработке Web-приложений состоит в том, что при работе по протоколу HTTP соединение между броузером и сервером прекращается сразу после окончания вывода очередной страницы. Таким образом, мы не можем определить, какие действия выполнял пользователь на предыдущих страницах нашего Web-приложения, то есть сохранять контекст приложения. Благодаря использованию объекта %session появляется возможность передать информацию от одного запроса к другому, от одной страницы к другой.

Необходимая информация сохраняется в объекте %session в виде пар «имя/значение» и может использоваться на любой другой странице. Например, сохранение значения ”Admin” переменной “Name” в объекте %session выполняется с помощью метода Set объекта session:

Do %session.Set(“Name”,”Admin”)

Позже, в ходе обработки страницы, можно получить значение Name следующим образом: %session.Get(“Name”).

Объект %session содержит ряд свойств, методов и параметров, которые помогают разработчику управлять сессией. Некоторые полезные свойства и методы объекта %session приведены в табл. 17. Ознакомиться с документацией объекта %session класса %CSP.Session можно по адресу:

(http://127.0.0.1:1972/apps/documatic?CLASSNAME=%25CSP.Session).

Таблица 17

Имя	Возвращаемое значение	Функция	Аргумент(ы)
Count	%Library.Integer	Кол-во значений под заданным именем (поля данных)	Имя поля данных
Get	Соответствующий аргумент	Извлекает из объекта %session значение заданного поля данных (или пустую строку, если значение отсутствует)	Имя поля данных

Kill	нет	Удаляет поле из объекта	Имя поля данных
Next	%Library.String	Навигация по полям объекта, возвращает имя следующего поля	Имя предыдущего поля данных
OnAppTimeOut	нет	Стандартный метод для обработки тайм-аута, устанавливает свойство EndSession в 1	нет
onStart	%Library.Boolean	Вызывается сразу после создания и инициализации объекта %session, возвращает ИСТИНУ, если должен последовать вызов страницы	нет
IsDefined	%Library.Boolean	Возвращает ИСТИНУ, если значение поля данных передано странице	Имя поля данных
Set	нет	Сохраняет измененное значение поля данных, либо создает новое поле	Имя поля данных

Для иллюстрации возможностей CSP по поддержке сессии выполним следующее простое упражнение. Создадим CSP-страницу Session.csp, задающую значение сессионной переменной Page. Страница Session1.csp вызывается со страницы Session.csp и отображает значение сессионной переменной Page. Код страницы Session.csp:

```
<HTML>
<HEAD>
<TITLE>         Пример использования session  </TITLE>
</HEAD>
<BODY>
  <script language="Cache" runat="server">
    // назначить переменной значение
    do %session.Set("Page",45)
  </script>
 Страница 1
 Назначает значение #(%session.Get("Page"))#
 сессионной переменной Page
 <a href="Session1.csp">Страница 2</a>
 </BODY>
</HTML>
```

Код страницы Session1.csp:

```
<HTML>
<HEAD>
<TITLE>         Пример использования session  </TITLE>
</HEAD>
<BODY>
 Страница 2
 Получает значение сессионной переменной Page
     равное #(%session.Get("Page"))#

</BODY>
</HTML>
```

8.7. Пример разработки форм ввода и просмотра объектов класса

Задание: Разработать формы просмотра, добавления, изменения и удаления объектов класса Books, содержащего информацию о книгах. Описание классов, необходимых для этого, приведены ниже.

Класс «Книги» (Books) – содержит информацию о книгах. Где Title – название книги, Authors – список авторов, CountPage – количество страниц, Category – категория, ссылка на хранимый класс «Категории», Decsription – описание книги. В классе созданы два метода: метод класса AddBook для создания объекта и метод экземпляра AlterBook для изменения свойств объекта. Определение класса:

```
Class User.Books Extends %Persistent [ ClassType = persistent, ProcedureBlock ]
{
Property Authors As %String [ Collection = list ];
Property Category As User.Category;
Property CountPage As %Integer;
Property Decsription As %String(MAXLEN = 100);
Property Title As %String;
Property Cena As %Numeric;

Method AlterBook(title As %String, authors As %String, countpage As %String, desc As %String, categ As Category, cena As %Numeric)
{ // метод для изменения свойств объекта
    set ..Title=title
      do ..Authors.InsertAt(authors,1)
     do ..CategorySetObjectId(categ)
    set ..CountPage=countpage
    set ..Decsription=desc
    set ..Cena=cena
    set sc= ##this.%Save()

}

ClassMethod AddBook(title As %String, author As %String, countpage As %String, desc As %String, categ As Category, cena As %Numeric) as Books
{ // метод класса для создания экземпляра объекта
   set book=##class(Books).%New()
   set book.Title=title
   Set book.CountPage=countpage
   Set book.Decsription=desc
   Set book.Cena=cena
    do book.CategorySetObjectId(categ)
    do book.Authors.InsertAt(author,1)
   set sc=book.%Save()
  Quit book
}
Index IndBooks On Title As Exact;
}
```

Класс «Категории» (Category), содержит информацию о категориях, к которым могут относиться конкретные книги. Определение класса:

```
Class User.Category Extends %Persistent [ ClassType = persistent, ProcedureBlock ]
```

```
    {
    Property CategoryName As %String;
    Index InxCat On CategoryName As Exact [ Data = CategoryName ];

    Method AltCateg(categoryname As %String) As %Status
    {  // метод экземпляра: изменить название категории
      Set ..CategoryName=categoryname
      Quit ..%Save()
    }
}
```

Решение:

Создадим три csp-страницы:

1. BookBrowse.csp – для просмотра всех объектов.
2. AddBooks1.csp – для добавления нового объекта
3. AlterDel1.csp – для изменения и удаления существующего объекта

Со страницы BookBrowse.csp по гиперссылке «Добавить книгу» вызывается страница AddBooks1.csp, которая является формой ввода новых объектов для класса «Книги». В поле ID страницы BookBrowse.csp по гиперссылке вызывается страница AlterDel1.csp, которая обеспечивает изменение удаление выбранного объекта.

На страницах AddBooks1.csp и AlterDel1.csp находятся кнопки типа submit, по которым страница закрывается, и значения полей редактирования передаются форме BookBrowse.csp, определенной в параметре Action оператора Form. Страница BookBrowse.csp каждый раз обновляется. В начале страницы BookBrowse.csp располагается программный код Cache, который выполняет добавление, изменение или удаление объекта.

BookBrowse.csp – форма просмотра.

Внешний вид формы приведен на рис. 15.

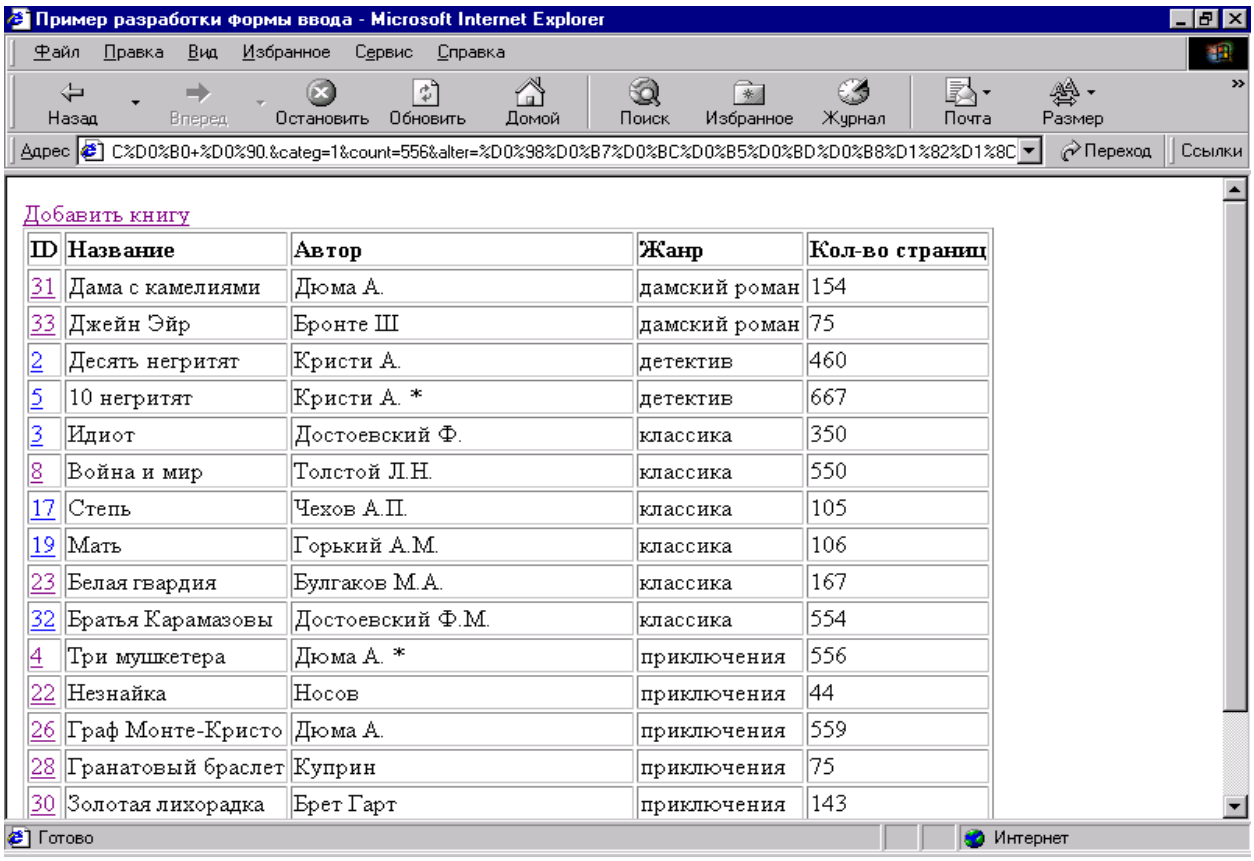


Рисунок 15. BookBrowse.csp

Программный код страницы BookBrowse.csp:

```
<HTML>
<HEAD>
<TITLE>       Пример разработки формы просмотра </TITLE>
</HEAD>
<BODY>
 <script language="Cache" runat="server">
   if %request.Get("add")'=""
   { // добавить объект
 do ##class(Books).AddBook(%request.Get("title"),%request.Get("author"),
                           %request.Get("count"),"",%request.Get("categ"),%request.Get("cena"))
   }
   if %request.Get("alter")'=""
   { // изменить объект
   set book=##class(Books).%OpenId(%request.Get("id"))
     do
book.AlterBook(%request.Get("title"),%request.Get("author"),%request.Get("count"),"",%reque
st.Get("categ"),
     %request.Get("cena"))
   }
   if %request.Get("del")'=""
   { // удалить объект
     Set sc=##class(Books).%DeleteId(%request.Get("id"))
   }
 </script>
  <script language=SQL name="query">
        SELECT ID,Category->CategoryName,CountPage,Decsription,Title,Authors
```

```
        FROM Books ORDER BY  Category->CategoryName
  </script>

<a href="AddBooks1.csp">Добавить книгу</a>

  <table border=1 bgcolor="">
  <tr>
   <td> <b>ID</b></td>
   <td><b> Название</b> </td>
   <td><b> Автор</b> </td>
   <td> <b>Жанр</b></td>
    <td><b>Кол-во страниц</b></td>
  </tr>
  <tr>
        <csp:while  condition=query.Next()>
                        <td>
                        <a href=AlterDel1.csp?ID=#(query.Get("ID"))#>
                                #(query.Get("ID"))# </a></td>
                        <td>#(query.Get("Title"))#</td>
                        <td>#(query.Get("Authors"))#</td>
                        <td>#(query.Get("CategoryName"))#</td>
                        <td>#(query.Get("CountPage"))#</td>

        </tr>
        </csp:while>
</table>
</BODY>
</HTML>
```

AddBooks1.csp – добавление нового объекта

Внешний вид страницы **AddBooks1.csp** приведен на рис. 16.

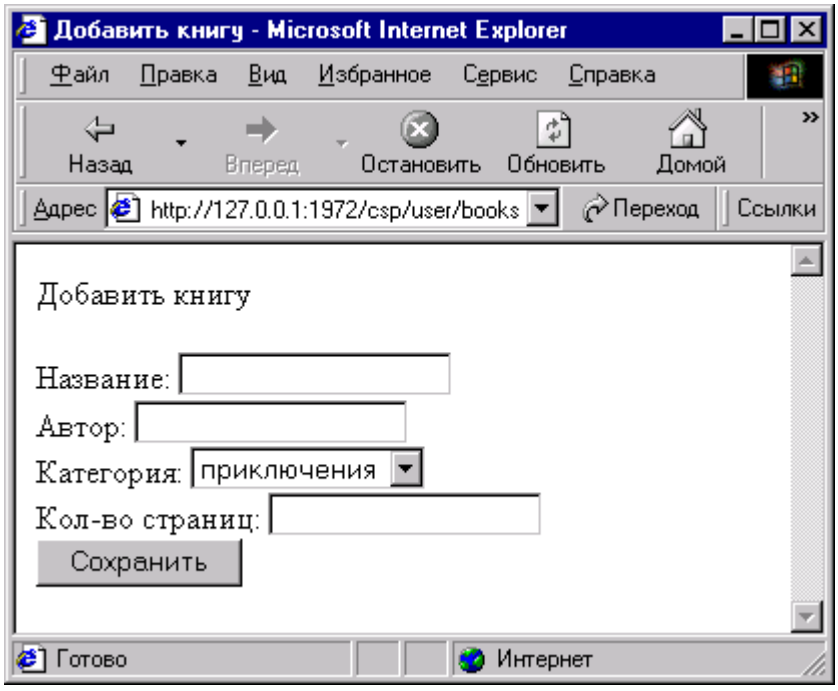


Рисунок 16. Страница AddBooks1.csp

Программный код страницы AddBooks1.csp:

```
<HTML>
<HEAD>
<TITLE>       Добавить книгу </TITLE>
</HEAD>
<BODY>
 <script language=SQL name="Cat">
   Select * from Category
 </script>
   Добавить книгу
  <form name="add" action="BookBrowse.csp">
   Название:  <input type="text" name="title" value=""> <br>
   Автор:  <input type="text" name="author" value=""><br>
   Категория:
   <select name="categ" size="1">
    <csp:while condition=Cat.Next()>
      <option value='#(Cat.Get("ID"))#'>#(Cat.Get("CategoryName"))#
    </csp:while>
   </select><br>
   Кол-во страниц:  <input type="text" name="count" value=""><br>
   <input type="submit" name="add" value="Сохранить"> <br>
  </form>
</BODY>
</HTML>
```

AlterDel1.csp – форма изменения и удаления существующего объекта

Внешний вид страницы AlterDel1.csp приведен на рис. 17.

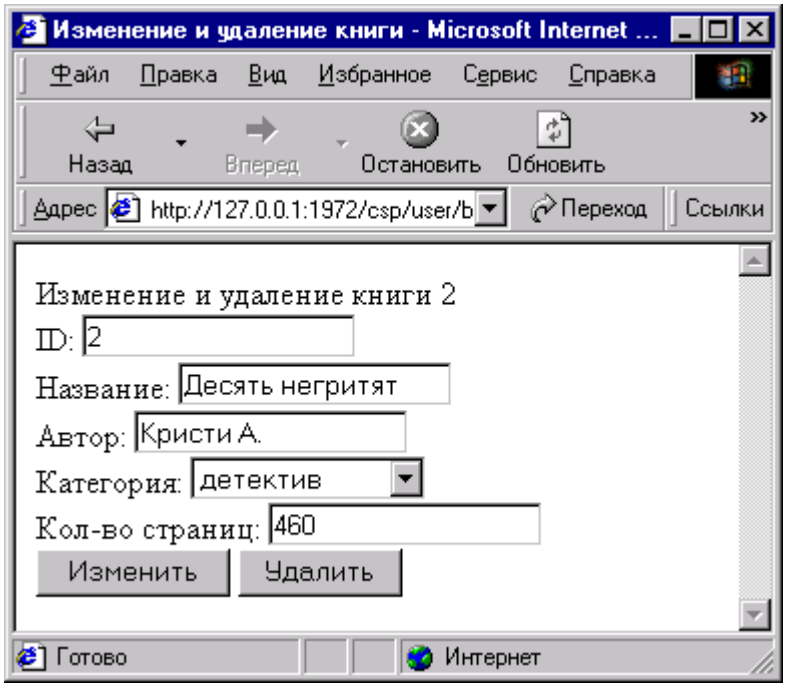


Рисунок 17. Страница AlterDel1.csp

Программный код страницы AlterDel1.csp:

```
<HTML>
<HEAD>
<!-- Изменение и удаление книги -->
<TITLE>          Изменение и удаление книги </TITLE>
</HEAD>
<BODY>
 <script language=SQL name="Cat">
   Select * from Category
 </script>

  <csp:object classname="Books" name="book"
       objid=#(%request.Get("ID"))#>
<form name="add" Action="BookBrowse.csp">
  Изменение и удаление книги #(%request.Get("ID"))# <br>
ID:              <input type="text" name="id" value="#(%request.Get("ID"))#"> <br>
Название:   <input type="text" name="title" value="#(book.Title)#"> <br>
Автор:       <input type="text" name="author" value="#(book.Authors.GetAt(1))#"><br>
   Категория:
   <select name="categ" size="1">
    <csp:while condition=Cat.Next()>
      <option value='#(Cat.Get("ID"))#'>
            <csp:if condition='book.CategoryGetObjectId()=Cat.Get("ID")'>
                 <option selected>
            </csp:if>
    #(Cat.Get("CategoryName"))#
     </csp:while>
   </select><br>
   Кол-во страниц:  <input type="text" name="count" value="#(book.CountPage)#"><br>
  <input type="submit" name="alter" value="Изменить">
  <input type="submit" name="del" value="Удалить">
 </form>
</BODY>
</HTML>
```

Решение с использованием серверного метода на событие OnClick:

onclick = "#server (..Method(…)

предлагается выполнить самостоятельно.

8.8. Пример организации поиска экземпляра класса с использованием индексного глобала

Пусть имеем класс Abonent со свойствами Fio – фамилия абонента, Login – его логическое имя, Password – пароль. Требуется создать интерфейс для поиска нужного экземпляра класса Abonent по значениям его свойств Login и Password.

Такую задачу можно решить с использованием параметрического Select-запроса языка SQL, что и предлагается читателю сделать самостоятельно в качестве полезного упражнения.

Мы же будем использовать для организации поиска индексный глобаль. Для этого создадим в классе Abonent индекс для свойства Login:

```
Index LoginIndex On Login As Exact;
```

Для непосредственного поиска требуется создать метод класса Abonent, назовем его, скажем, CheckPassw, который по заданным значениям Login и Password осуществляет поиск экземпляра в классе. В случае удачного поиска метод возвращает id найденного экземпляра, и 0(ноль) в случае, если экземпляр не найден. Для поиска по индексу используется функция $Order (см. главу 7, п.7.6.3). Структура хранения стандартных индексов в виде глобалей приведена в п.7.7.4 главы 7. Код метода приведен ниже:

```
ClassMethod CheckPassw(log As %String, passw As %String) As %Status
{ set id=$Order(^User.AbonentI("LoginIndex",log,-1))
if id="" quit 0
Set obj=##class(User.Abonent).%OpenId(id)
if obj.Password=passw quit  id
quit 0
}
```

Создадим также две csp-страницы. Первая – это форма ввода значений Login и Password. Назовем ее VvodParol.csp. Внешний вид формы VvodParol.csp приведен на рис.18.

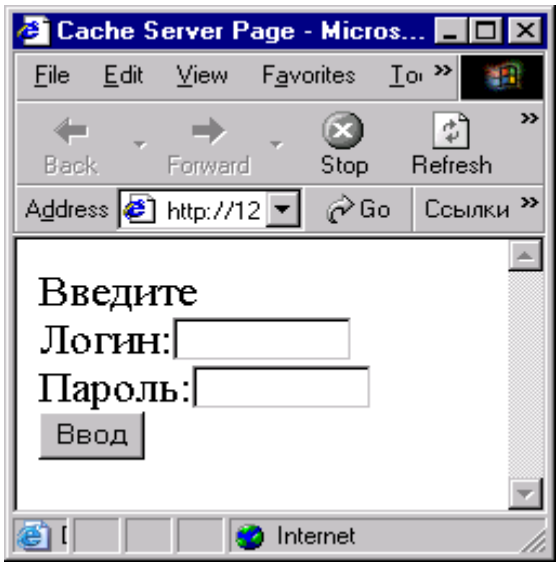


Рисунок 18

По кнопке «Ввод», которая является кнопкой типа submit, вызывается вторая страница, с именем CheckParol.csp, которая используется для обработки того, что введено в форму VvodParol.csp. Внешний вид страницы CheckParol.csp приведен на рис. 19.

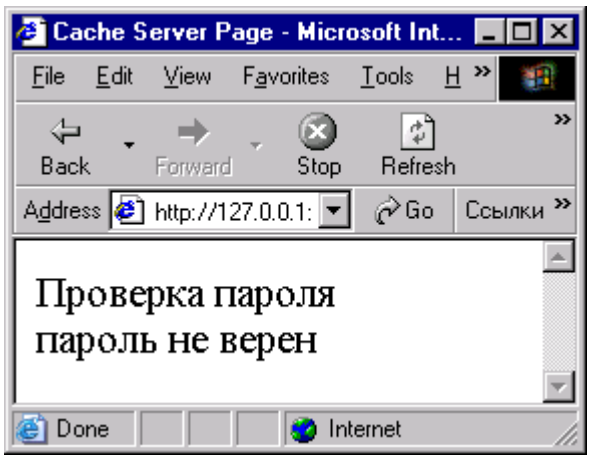


Рисунок 19

Каждый раз при загрузке страницы CheckParol.csp работает Cache-код, который выполняет поиск нужного экземпляра и информирует пользователя о результатах поиска.

Код страницы VvodParol.csp приведен ниже:

```
        <HTML>
<HEAD>
<TITLE>      Cache Server Page </TITLE>
        </HEAD>
        <BODY>
<form name="parol" action="CheckParol.csp">
        Введите <br>
        Логин:<INPUT TYPE="text" NAME="login" SIZE="10"><br>
        Пароль:<INPUT TYPE="password" NAME="passw" SIZE="10"><br>
        <INPUT TYPE="submit" NAME="vvodpar" VALUE="Ввод">
        </form>
    </BODY>
    </HTML>
```

Код страницы CheckParol.csp имеет вид:

```
    <TITLE> Cache Server Page </TITLE>
    </HEAD>
      <BODY>
    Проверка пароля <br>
    <script language=Cache runat=server>
    set obj=##class(User.Abonent).CheckPassw(
                                        %request.Get("login"),%request.Get("passw"))
      </script>
      <csp:if condition="obj'=0">
            пароль верен
      <csp:else>
            пароль не верен
      </csp:if>
      </BODY>
    </HTML>
```

Список литературы

1. Документация по СУБД Caché 5.0.
2. СУБД Caché, объектно-ориентированная разработка приложений, Кирстен В., Иринger М., Рёриг В., Шульте П., С-Пб, 2001
3. Технологический справочник по СУБД Caché 5.0.
4. Постреляционная технология Caché для реализации объектных приложений, Кречетов Н.Е., Петухов Е.А., Скворцов В.И., Умников А.В., Щукин Б.А., МИФИ, 2001
5. «Внутренний мир объектно-ориентированных СУБД», А. Андреев, Д. Березкин, Р. Самарев, Открытые системы, 2001
6. «Постреляционная СУБД Caché», Сиротюк О. В.