Конспект лекций

по дисциплине **«Программирование сетевых приложений»**
для студентов специальности
1-40 05 01 Информационные системы и технологии (по направлениям)

Тема 1. Основные принципы, методы и перспективы разработки объектно-ориентированных программ и сетевых приложений на основе технологий Microsoft и Oracle

Предмет курса и содержание дисциплины, ее связь с другими дисциплинами. Две парадигмы программирования. Основные направления в программировании. Возникновение объектно-ориентированного подхода (ООП). Базовые принципы ООП. Реализации основных концепций объектно-ориентированного программирования – полиморфизма, наследования в языках высокого уровня. Технология разработки программ на основе ООП. Обзор возможностей языков и технологий, разработанных компаниями Microsoft и Oracle

Предмет курса и содержание дисциплины, ее связь с другими дисциплинами

Развитие современных технических устройств не мыслим без внедрения технологий компьютерных сетей. Компьютерные сети являются необходимым элементом интеграции различного рода устройств и комплексов, в том числе и устройств промышленной электроники. Поэтому программирование компьютерных сетей является важным элементом подготовки современных инженеров и магистров, специализирующихся на проектировании сложных систем промышленной электроники. Данный курс «Разработка сетевых приложений» предназначен для изучения основ и принципов создания сетевого программного обеспечения, базирующегося на клиент/серверной модели. Данный курс базируется на курсах «Информатика», «Программирование», «Технологии программирования», «Базы данных», «Компьютерные сети», «Основы вычислительной техники», «Операционные системы», «Введение в интернет»

Две парадигмы программирования

В настоящее время существуют два основных подхода к распараллеливанию вычислений. Это параллелизм данных и параллелизм задач. В англоязычной литературе соответствующие термины - data parallel и message passing. В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера. При этом приходится решать разнообразные проблемы. Прежде всего, это достаточно равномерная загрузка процессоров, так как если основная вычислительная работа будет ложиться на один из процессоров, мы приходим к случаю обычных последовательных вычислений и тогда никакого выигрыша за счет распараллеливания задачи не будет. Сбалансированная работа процессоров -

это первая проблема, которую следует решить при организации параллельных вычислений. Другая и не менее важная проблема - скорость обмена информацией между процессорами. Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую, на ожидание информации, необходимой для дальнейшей работы данного процессора. Рассматриваемые парадигмы программирования различаются методами решения этих двух основных проблем. Разберем более подробно параллелизм данных и параллелизм задач.

Основная идея подхода, основанного на параллелизме данных, заключается в том, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции - перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию опций векторной или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений. Наиболее распространенными языками для параллельных вычислений являются высокопроизводительный ФОРТРАН (High Performance FORTRAN) и параллельные версии языка С (это, например, С*).

Более детальное описание рассматриваемого подхода к распараллеливанию содержит указание на следующие его основные особенности:

1. Обработкой данных управляет одна программа.
2. Пространство имен является глобальным, то есть для программиста существует одна единственная память, а детали структуры данных, доступа к памяти и межпроцессорного обмена данными от него скрыты.
3. Слабая синхронизация вычислений на параллельных процессорах, то есть выполнение команд на разных процессорах происходит, как правило, независимо и только лишь иногда производится согласование выполнения циклов или других программных конструкций - их синхронизация. Каждый процессор выполняет один и тот же фрагмент программы, но нет гарантии, что в заданный момент времени на всех процессорах выполняется одна и та же машинная команда.
4. Параллельные операции над элементами массива выполняются одновременно на всех доступных данной программе процессорах.

Таким образом, в рамках данного подхода от программиста не требуется больших усилий по векторизации или распараллеливанию вычислений. Даже при программировании сложных вычислительных алгоритмов можно использовать библиотеки подпрограмм, специально разработанных с учетом конкретной архитектуры компьютера и оптимизированных для этой архитектуры.

Подход, основанный на параллелизме данных, базируется на использовании при разработке программ базового набора операций:

1) операции управления данными;
2) операции над массивами в целом и их фрагментами;
3) условные операции;
4) операции приведения;
5) операции сдвига;
6) операции сканирования;
7) операции, связанные с пересылкой данных.

Рассмотрим эти базовые наборы операций.

Управление данными

В определенных ситуациях возникает необходимость в управлении распределением данных между процессорами. Это может потребоваться, например, для обеспечения равномерной загрузки процессоров. Чем более равномерно загружены работой процессоры, тем более эффективной будет работа компьютера.

Операции над массивами

Аргументами таких операций являются массивы в целом или их фрагменты (сечения), при этом данная операция применяется одновременно (параллельно) ко всем элементам массива. Примерами операций такого типа являются вычисление поэлементной суммы массивов, умножение элементов массива на скалярный или векторный множитель и т.д. Операции могут быть и более сложными, например, вычисление функций от массива.

Условные операции

Эти операции могут выполняться лишь над теми элементами массива, которые удовлетворяют какому-то определенному условию. В сеточных методах это может быть четный или нечетный номер строки (столбца) сетки или неравенство нулю элементов матрицы.

Операции приведения

Операции приведения применяются ко всем элементам массива (или его сечения), а результатом является одно единственное значение, например, сумма элементов массива или максимальное значение его элементов.

Операции сдвига

Для эффективной реализации некоторых параллельных алгоритмов требуются операции сдвига массивов. Примерами служат алгоритмы обработки изображений, конечно-разностные алгоритмы.

Операции сканирования

Операции сканирования еще называются префиксными / суффиксными операциями. Префиксная операция, например, суммирование, выполняется следующим образом. Элементы массива суммируются последовательно, а результат очередного суммирования заносится в очередную ячейку нового, результирующего массива, причем номер этой ячейки совпадает с числом просуммированных элементов исходного массива.

Операции пересылки данных

Это, например, операции пересылки данных между массивами разной формы (то есть имеющими разную размерность и разную протяженность по каждому измерению).

При программировании на основе параллелизма данных часто используются специализированные языки - CM FORTRAN, C*, FORTRAN+, MPP FORTRAN, Vienna FORTRAN, а также HIGH PERFORMANCE FORTRAN (HPF). HPF основан на языке программирования ФОРТРАН 90, что связано с наличием в последнем удобных операций над массивами [39].

Стиль программирования, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Компьютер при этом представляет собой MIMD - машину. Аббревиатура MIMD обозначает в известной классификации архитектур ЭВМ компьютер, выполняющий одновременно множество различных операций над множеством, вообще говоря, различных и разнотипных данных. Для каждой подзадачи пишется своя собственная программа на обычном языке программирования, обычно это ФОРТРАН или С. Чем больше подзадач, тем большее число процессоров можно использовать, тем большей эффективности можно добиться. Важно то, что все эти программы должны обмениваться результатами своей работы, практически такой обмен осуществляется вызовом процедур специализированной библиотеки. Программист при этом может контролировать распределение данных между процессорами и подзадачами и обмен данными. Очевидно, что в этом случае требуется определенная работа для того, чтобы обеспечить эффективное совместное выполнение различных программ. По сравнению с подходом, основанным на параллелизме данных, данный подход более трудоемкий, с ним связаны следующие проблемы:

1) повышенная трудоемкость разработки программы и ее отладки;
2) на программиста ложится вся ответственность за равномерную загрузку процессоров параллельного компьютера;
3) программисту приходится минимизировать обмен данными между задачами, так как пересылка данных - наиболее "времяемкий" процесс;
4) повышенная опасность возникновения тупиковых ситуаций, когда отправленная одной программой посылка с данными не приходит к месту назначения.

Привлекательными особенностями данного подхода являются большая гибкость и большая свобода, предоставляемая программисту в разработке программы, эффективно использующей ресурсы параллельного компьютера и, как следствие, возможность достижения максимального быстродействия. Примерами специализированных библиотек являются библиотеки MPI (Message Passing Interface) [61] и PVM (Parallel Virtual Machines) [62]. Эти библиотеки являются свободно распространяемыми и существуют в исходных кодах. Библиотека MPI разработана в Аргоннской национальной лаборатории

(США), а PVM - разработка Окриджской национальной лаборатории, университетов штата Теннеси и Эмори (Атланта).

Литература

Процедурное и объектно-ориентированное программирование. Режим доступа: https://www.examclouds.com/ru/java/java-core-russian/metodiki-programmirovanija.

Тема 2. Фундаментальные методы и свойства сетевой архитектуры и механизмы ее программной реализации в Windows и web-приложениях

Понятие о сетевой архитектуре. Общие представления о процессе передачи данных по сети. Понятие протокола и механизмы взаимодействия системы клиент-сервер. Обзор общих принципов построения многоуровневых приложений. Работа с сетью на уровне сокетов и потоков. Способы доступа к ресурсам сети из программных приложений при помощи URL

Понятие о сетевой архитектуре.

Архитектура сети определяет основные элементы сети, характеризует ее общую логическую организацию, техническое обеспечение, программное обеспечение, описывает методы кодирования. Архитектура также определяет принципы функционирования и интерфейс пользователя.

В данном курсе будет рассмотрено три вида архитектур:

1. архитектура терминал – главный компьютер;
2. одноранговая архитектура;
3. архитектура клиент – сервер.

Архитектура терминал – главный компьютер

Архитектура *терминал – главный компьютер* (terminal – host computer architecture) – это концепция информационной сети, в которой вся обработка данных осуществляется одним или группой главных компьютеров.

Рассматриваемая архитектура предполагает два типа оборудования:

1. Главный компьютер, где осуществляется управление сетью, хранение и обработка данных;
2. Терминалы, предназначенные для передачи главному компьютеру команд на организацию сеансов и выполнения заданий, ввода данных для выполнения заданий и получения результатов.

Главный компьютер через мультиплексоры передачи данных (МПД) взаимодействуют с терминалами. Классический пример архитектуры сети с главными компьютерами – системная сетевая архитектура (System Network Architecture – SNA).

Одноранговая архитектура

Одноранговая архитектура (peer-to-peer architecture) – это концепция информационной сети, в которой ее ресурсы рассредоточены по всем системам. Данная архитектура характеризуется тем, что в ней все системы равноправны.

К одноранговым сетям относятся малые сети, где любая рабочая станция может выполнять одновременно функции файлового сервера и рабочей станции. В одноранговых ЛВС дисковое пространство и файлы на любом компьютере могут быть общими. Чтобы ресурс стал общим, его необходимо отдать в общее пользование, используя службы удаленного доступа сетевых одноранговых операционных систем. В зависимости от того, как будет установлена защита данных, другие пользователи смогут пользоваться файлами сразу же после их создания. Одноранговые ЛВС достаточно хороши только для небольших рабочих групп.

Одноранговые ЛВС являются наиболее легким и дешевым типом сетей для установки. Они на компьютере требуют, кроме сетевой карты и сетевого носителя, только операционной системы Windows 95 или Windows for Workgroups. При соединении компьютеров, пользователи могут предоставлять ресурсы и информацию в совместное пользование.

Одноранговые сети имеют следующие преимущества:

они легки в установке и настройке;

отдельные ПК не зависят от выделенного сервера;

пользователи в состоянии контролировать свои ресурсы;

малая стоимость и легкая эксплуатация;

минимум оборудования и программного обеспечения;

нет необходимости в администраторе;

хорошо подходят для сетей с количеством пользователей, не превышающим десяти.

Проблемой одноранговой архитектуры является ситуация, когда компьютеры отключаются от сети. В этих случаях из сети исчезают виды сервиса, которые они предоставляли. Сетевую безопасность одновременно можно применить только к одному ресурсу, и пользователь должен помнить столько паролей, сколько сетевых ресурсов. При получении доступа к разделяемому ресурсу ощущается падение производительности компьютера. Существенным недостатком одноранговых сетей является отсутствие централизованного администрирования.

Использование одноранговой архитектуры не исключает применения в той же сети также архитектуры "терминал – главный компьютер" или архитектуры "клиент – сервер".

Архитектура клиент – сервер

Архитектура клиент – сервер (client-server architecture) – это концепция информационной сети, в которой основная часть ее ресурсов сосредоточена в серверах, обслуживающих своих клиентов. Рассматриваемая архитектура определяет два типа компонентов: серверы и клиенты.

Сервер - это объект, предоставляющий сервис другим объектам сети по их запросам. Сервис- это процесс обслуживания клиентов.

Сервер работает по заданиям клиентов и управляет выполнением их заданий. После выполнения каждого задания сервер посылает полученные результаты клиенту, пославшему это задание.

Сервисная функция в архитектуре клиент – сервер описывается комплексом прикладных программ, в соответствии с которым выполняются разнообразные прикладные процессы.

Процесс, который вызывает сервисную функцию с помощью определенных операций, называется клиентом. Им может быть программа или пользователь.

Клиенты – это рабочие станции, которые используют ресурсы сервера и предоставляют удобные интерфейсы пользователя. Интерфейсы пользователя это процедуры взаимодействия пользователя с системой или сетью. Клиент является инициатором и использует электронную почту или другие сервисы сервера. В этом процессе клиент запрашивает вид обслуживания, устанавливает сеанс, получает нужные ему результаты и сообщает об окончании работы.

В сетях с выделенным файловым сервером на выделенном автономном ПК устанавливается серверная сетевая операционная система. Этот ПК становится сервером. Программное обеспечение (ПО), установленное на рабочей станции, позволяет ей обмениваться данными с сервером.

Наиболее распространенные сетевые операционная системы:

NetWare фирмы Novel;

Windows NT фирмы Microsoft;

UNIX фирмы AT&T;

Linux.

Помимо сетевой операционной системы необходимы сетевые прикладные программы, реализующие преимущества, предоставляемые сетью.

Сети на базе серверов имеют лучшие характеристики и повышенную надежность. Сервер владеет главными ресурсами сети, к которым обращаются остальные рабочие станции.

В современной клиент – серверной архитектуре выделяется четыре группы объектов: клиенты, серверы, данные и сетевые службы. Клиенты располагаются в системах на рабочих местах пользователей. Данные в основном хранятся в серверах. Сетевые службы являются совместно используемыми серверами и данными. Кроме того службы управляют процедурами обработки данных.

Сети клиент – серверной архитектуры имеют следующие преимущества:

позволяют организовывать сети с большим количеством рабочих станций;

обеспечивают централизованное управление учетными записями пользователей, безопасностью и доступом, что упрощает сетевое администрирование;

эффективный доступ к сетевым ресурсам;

пользователю нужен один пароль для входа в сеть и для получения доступа ко всем ресурсам, на которые распространяются права пользователя.

Наряду с преимуществами сети клиент – серверной архитектуры имеют и ряд недостатков:

неисправность сервера может сделать сеть неработоспособной, как минимум потерю сетевых ресурсов;

требуют квалифицированного персонала для администрирования;

имеют более высокую стоимость сетей и сетевого оборудования.

Выбор архитектуры сети

Выбор архитектуры сети зависит от назначения сети, количества рабочих станций и от выполняемых на ней действий.

Следует выбрать одноранговую сеть, если:

количество пользователей не превышает десяти;

все машины находятся близко друг от друга;

имеют место небольшие финансовые возможности;

нет необходимости в специализированном сервере, таком как сервер БД, факс-сервер или какой-либо другой;

нет возможности или необходимости в централизованном администрировании.

Следует выбрать клиент серверную сеть, если:

количество пользователей превышает десяти;

требуется централизованное управление, безопасность, управление ресурсами или резервное копирование;

необходим специализированный сервер;

нужен доступ к глобальной сети;

требуется разделять ресурсы на уровне пользователей.

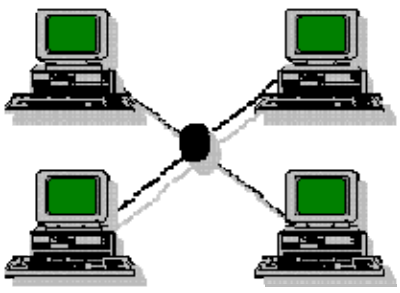
Топология сети

Топология сети описывает схему физического соединения компьютеров. Существуют 3 основных типа сетевой топологии:

Общая шина.

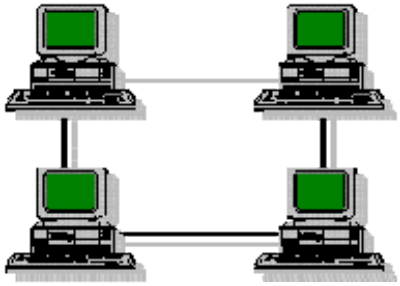
При использовании шинной топологии компьютеры соединяются в одну линию, по концам которой устанавливают терминаторы. Преимущества шинной топологии заключаются в простоте организации сети и низкой стоимости. Недостатком является низкая устойчивость к повреждениям - при любом обрыве кабеля вся сеть перестает работать, а поиск повреждения весьма затруднителен.

Звезда.



При использовании топологии "звезда", каждый компьютер подключается к специальному концентратору (хабу). Преимуществом этой топологии является ее устойчивость к повреждениям кабеля - при обрыве перестает работать только один из узлов сети и поиск повреждения значительно упрощается. Недостатком является более высокая стоимость.

Кольцо.



При такой топологии узлы сети образуют виртуальное кольцо (концы кабеля соединены друг с другом). Каждый узел сети соединен с двумя соседними. Эту топологию активно продвигает фирма IBM (сети Token Ring). Преимуществом кольцевой топологии является ее высокая надежность (за счет избыточности), однако стоимость такой сети достаточно высока за счет расходов на адаптеры, кабели и дополнительные приспособления.

Спецификации IEEE

Каждая сеть должна следовать определенным правилам (протоколам) при передачи данных от одного компьютера к другому. Протокол определяет способ доступа узла к передающей среде (кабелю) и способ передачи информации от одного узла к другому.

В настоящее время используется два типа протоколов доступа к среде:

передача маркера (token) используется в сетях IBM Token Ring и FDDI;

множественный доступ с детектированием несущей (CSMA) используется в сетях Ethernet.

Как многие компьютерные и сетевые технологии, которыми мы пользуемся, сетевая архитектура Ethernet разработана в научно-исследовательском центре Pa1o A1to ResearchCenter (PARC) компании Xerox в 1972 г. Коммерческая версия Ethernet была выпущена в 1975 г. и обеспечивала скорость передачи данных на уровне 3 Мбит/с.

Ethernet получила всеобщее признание, и компании Xerox, Intel и DigitalEquipmentCorporation (DEC) объединили свои усилия, чтобы улучшить технические характеристики Ethernet и довести скорость передачи данных до 10 Мбит/с. Именно эта версия Ethernet обеспечивающая скорость передачи данных на уровне 10 Мбит/с, прошла стандартизацию в институте IEEE, и ей была присвоена спецификация 802.3.Это самая популярная сетевая архитектура вмире.

Основные особенности; 1. Разделяемая среда и широковещательность ПД. 2. Метод доступа – CSMA/CD (carrier sense multiply access with collision detection) – коллективный доступ с опознаванием несущей и обнаружением

коллизий. 3. Физическая реализация Ethernet. Спецификации: — 10 Base 5 – Thick Ethernet (на толстом коаксиальном кабеле); — 10 Base 2 – Thin Ethernet (на тонком коаксиальном кабеле); — 10 Base T – Twisted Ethernet (на витой паре); — 10 Broad 36 (на широкополосном коаксиальном кабеле); — 100 Base T (100 Base T4, 100 Base TX, 100 Base FX) — Fast Ethernet, реализуемый на различных физических средах. — 1000 Base-T,SX,LX — 10 Gigabit 4 — Топология – линейная шина или звезда. 5 — Скорость ПД Мбит/с.

Понятие протокола и механизмы взаимодействия системы клиент-сервер

Основной принцип технологии "клиент-сервер" заключается в разделении функций приложения на три группы:

ввод и отображение данных (взаимодействие с пользователем);

прикладные функции, характерные для данной предметной области;

функции управления ресурсами (файловой системой, базой даных и т.д.)

Поэтому, в любом приложении выделяются следующие компоненты:

компонент представления данных

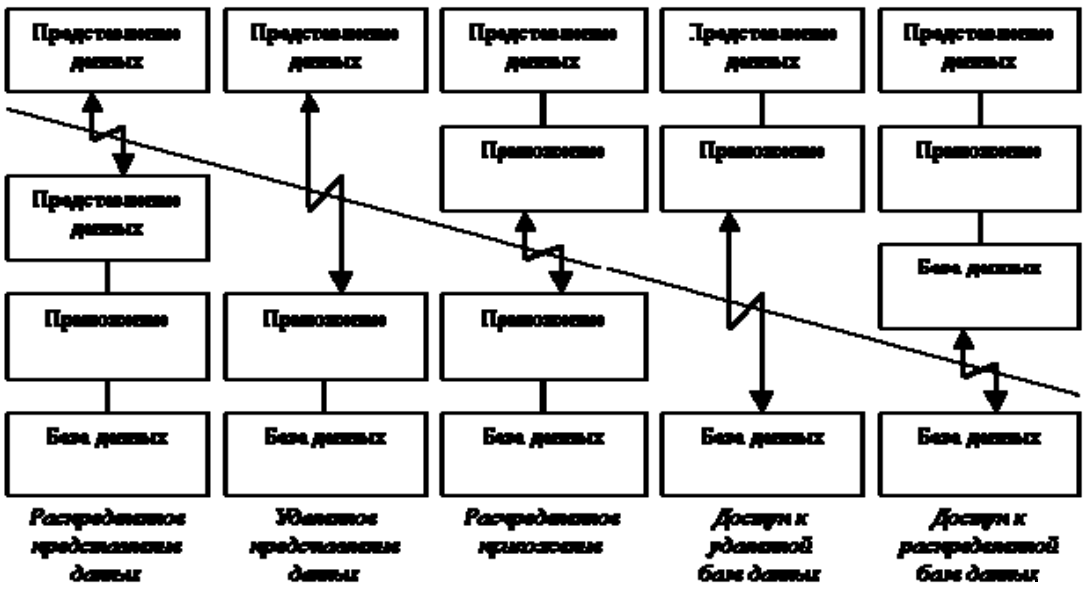
прикладной компонент

компонент управления ресурсом

Связь между компонентами осуществляется по определенным правилам, которые называют "протокол взаимодействия".

Модели взаимодействия клиент-сервер.

Компанией Gartner Group, специализирующейся в области исследования информационных технологий, предложена следующая классификация двухзвенных моделей взаимодействия клиент-сервер (двухзвенными эти модели называются потому, что три компонента приложения различным образом распределяются между двумя узлами):



Исторически первой появилась модель распределенного представления данных, которая реализовывалась на универсальной ЭВМ с подключенными к ней неинтеллектуальными терминалами. Управление данными и

взаимодействие с пользователем при этом объединялись в одной программе, на терминал передавалась только "картинка", сформированная на центральном компьютере.

Затем, с появлением персональных компьютеров (ПК) и локальных сетей, были реализованы модели доступа к удаленной базе данных. Некоторое время базовой для сетей ПК была архитектура файлового сервера. При этом один из компьютеров является файловым сервером, на клиентах выполняются приложения, в которых совмещены компонент представления и прикладной компонент (СУБД и прикладная программма). Протокол обмена при этом представляет набор низкоуровненых вызовов операций файловой системы. Такая архитектура, реализуемая, как правило, с помощью персональных СУБД, имеет очевидные недостатки - высокий сетевой трафик и отсутствие унифицированного доступа к ресурсам.

С появлением первых специализированных серверов баз данных появилась возможность другой реализации модели доступа к удаленной базе данных. В этом случае ядро СУБД функционирует на сервере, протокол обмена обеспечивается с помощью языка SQL. Такой подход по сравнению с файловым сервером ведет к уменьшению загрузки сети и унификации интерфейса "клиент-сервер". Однако, сетевой трафик остается достаточно высоким, кроме того, по прежнему невозможно удовлетворительное администрирование приложений, поскольку в одной программе совмещаются различные функции.

Позже была разработана концепция активного сервера, который использовал механизм хранимых процедур. Это позволило часть прикладного компонента перенести на сервер (модель распределенного приложения). Процедуры хранятся в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, что и SQL-сервер. Преимущества такого подхода: возможно централизованное администрирование прикладных функций, значительно снижается сетевой трафик (т.к. передаются не SQL-запросы, а вызовы хранимых процедур). Недостаток - ограниченность средств разработки хранимых процедур по сравнению с языками общего назначения (C и Pascal).

На практике сейчас обычно используются смешанный подход:

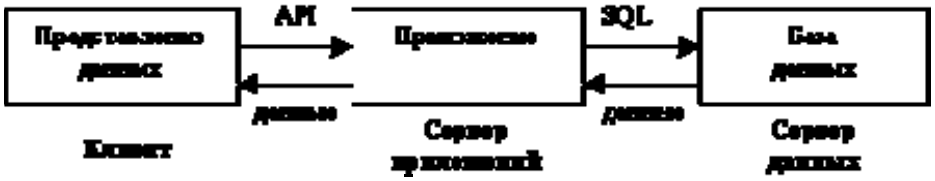
простейшие прикладные функции выполняются хранимыми процедурами на сервере

более сложные реализуются на клиенте непосредственно в прикладной программе

Сейчас ряд поставщиков коммерческих СУБД объявило о планах реализации механизмов выполнения хранимых процедур с использованием языка Java. Это соответствует концепции "тонкого клиента", функцией которого остается только отображение данных (модель удаленного представления данных).

В последнее время также наблюдается тенденция ко все большему использованию модели распределенного приложения. Характерной чертой таких приложений является логическое разделение приложения на две и более

частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате. В этом случае двухзвенная архитектура клиент-сервер становится трехзвенной, а к некоторых случаях, она может включать и больше звеньев.



Клиент и сервер взаимодействую друг с другом в сети Интернет или в любой другой компьютерной сети при помощи различных сетевых протоколов, например, IP протокол, HTTP протокол, FTP и другие. Протоколов на самом деле очень много и каждый протокол позволяет оказывать ту или иную услугу. Например, при помощи HTTP протокола браузер отправляет специальное HTTP сообщение, в котором указано какую информацию и в каком виде он хочет получить от сервера, сервер, получив такое сообщение, отсылает браузеру в ответ похожее по структуре сообщение (или несколько сообщений), в котором содержится нужная информация, обычно это HTML документ.

Сообщения, которые посылают клиенты получили названия HTTP запросы. Запросы имеют специальные методы, которые говорят серверу о том, как обрабатывать сообщение. А сообщения, которые посылает сервер получили название HTTP ответы, они содержат помимо полезной информации еще и специальные коды состояния, которые позволяют браузеру узнать то, как сервер понял его запрос.

Сейчас мы схематично описали, как взаимодействуют клиент и сервер на седьмом уровне модели OSI, но, на самом деле это взаимодействие происходит на всех семи уровнях. Когда клиент отправляет запрос, сообщение упаковывается, можно представить, что сообщение заворачивается в семь оберток (хотя их может быть намного больше или же меньше), а когда сообщение получает сервер, он начинает эти обертки разворачивать.

Также стоит заметить, что **в основе взаимодействия клиент-сервер лежит принцип того, что такое взаимодействие начинает клиент**, сервер лишь отвечает клиенту и сообщает о том может ли он предоставить услугу клиенту и если может, то на каких условиях. Клиентское программное обеспечение и серверное программное обеспечение обычно установлено на разных машинах, но также они могут работать и на одном компьютере.

Данная концепция взаимодействия была разработана в первую очередь для того, чтобы разделить нагрузку между участниками процесса обмена информацией, а также для того, чтобы разделить программный код поставщика и заказчика. Ниже вы можете увидеть упрощенную схему взаимодействия клиент-сервер.

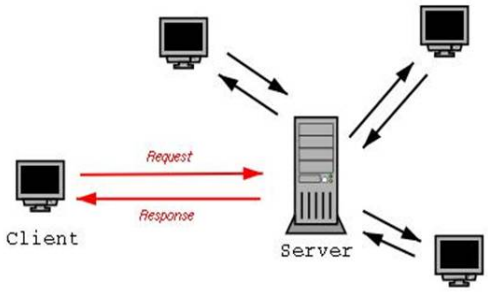


Рисунок – Простая схема взаимодействия клиент-сервер

Мы видим, что к одному серверу может обращаться сразу несколько клиентов (действительно, на одном сайте может находиться несколько посетителей). Также стоит заметить, что количество клиентов, которые могут одновременно взаимодействовать с сервером зависит от мощности сервера и от того, что хочет получить клиент от сервера.

Многие сетевые протоколы построены на архитектуре клиент-сервер, поэтому в их основе обычно лежат одинаковые или схожие принципы взаимодействия, а разницу мы видим лишь в деталях, которые обусловлены особенностями и спецификой области, для которой разрабатывался тот или иной сетевой протокол.

Почему веб-мастеру нужно понимать модель взаимодействия клиент-сервер

Давайте теперь ответим на вопрос: «**зачем веб-мастеру или веб-разработчику понимать концепцию взаимодействия клиент-сервер?**». Ответ, естественно, очевиден. Чтобы что-то делать своими руками нужно понимать, как это работает. Чтобы сделать сайт и, чтобы он правильно работал в сети Интернет или хотя бы просто работал, нам нужно понимать, как работает сеть Интернет.

Мы уже упоминали, что большая часть сетевых протоколов имеют архитектуру клиент-сервер. Например, веб-мастеру или веб-разработчику будут интересны протоколы седьмого и шестого уровня эталонной модели. Сетевым администраторам важно понимать, как происходит взаимодействие на уровнях с пятого по второй. Для инженеров связи наибольший интерес представляют протоколы с четвертого по первый уровень модели OSI.

Поэтому если вы действительно хотите быть профессионалом в сфере web, то сперва вам необходимо понимать, как происходит взаимодействии в сети (именно на седьмом уровне), а уже потом начинать изучать инструменты, которые позволят создавать сайты.

Архитектура «клиент-сервер»

Архитектура клиент-сервер определяет лишь общие принципы взаимодействия между компьютерами, детали взаимодействия определяют различные протоколы. Данная концепция нам говорит, что нужно разделять машины в сети на клиентские, которым всегда что-то надо и на серверные,

которые дают то, что надо. При этом взаимодействие всегда начинает клиент, а правила, по которым происходит взаимодействие описывает протокол.

Существует два вида архитектуры взаимодействия клиент-сервер: первый получил название **двухзвенная архитектура клиент-серверного взаимодействия**, второй – **многоуровневая архитектура клиент-сервер**(иногда его называют трехуровневая архитектура или трехзвенная архитектура, но это частный случай).

Принцип работы двухуровневой архитектуры взаимодействия клиент-сервер заключается в том, что обработка запроса происходит на одной машине без использования сторонних ресурсов. Двухзвенная архитектура предъявляет жесткие требования к производительности сервера, но в тоже время является очень надежной. Двухуровневую модель взаимодействия клиент-сервер вы можете увидеть на рисунке ниже.

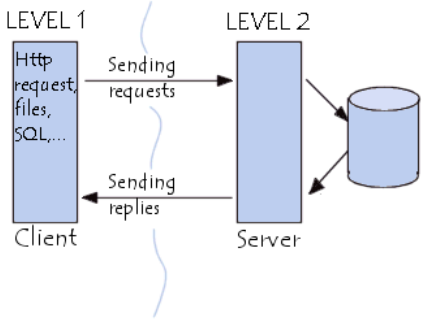


Рисунок – Двухуровневая модель взаимодействия клиент-сервер

Здесь четко видно, что есть клиент (1-ый уровень), который позволяет человеку сделать запрос, и есть сервер, который обрабатывает запрос клиента.

Если говорить про **многоуровневую архитектуру взаимодействия клиент-сервер**, то в качестве примера можно привести любую современную СУБД (за исключением, наверное, библиотеки SQLite, которая в принципе не использует концепцию клиент-сервер). Суть многоуровневой архитектуры заключается в том, что запрос клиента обрабатывается сразу несколькими серверами. Такой подход позволяет значительно снизить нагрузку на сервер из-за того, что происходит распределение операций, но в то же самое время данный подход не такой надежный, как двухзвенная архитектура. На рисунке ниже вы можете увидеть пример многоуровневой архитектуры клиент-сервер.

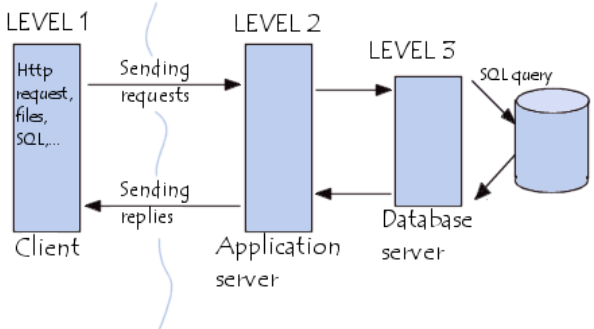


Рисунок – Многоуровневая архитектура взаимодействия клиент-сервер

Типичный пример трехуровневой модели клиент-сервер. Если говорить в контексте систем управления базами данных, то первый уровень – это клиент, который позволяет нам писать различные SQL запросы к базе данных. Второй уровень – это движок СУБД, который интерпретирует запросы и реализует взаимодействие между клиентом и файловой системой, а третий уровень – это хранилище данных.

Если мы посмотрим на данную архитектуру с позиции сайта. То первый уровень можно считать браузером, с помощью которого посетитель заходит на сайт, второй уровень – это связка Apache + PHP, а третий уровень – это база данных. Если уж говорить совсем просто, то PHP больше ничего и не делает, кроме как, гоняет строки и базы данных на экран и обратно в базу данных.

Преимущества и недостатки архитектуры клиент-сервер

Преимуществом модели взаимодействия клиент-сервер является то, что программный код клиентского приложения и серверного разделен. Если мы говорим про локальные компьютерные сети, то к преимуществам архитектуры клиент-сервер можно отнести пониженные требования к машинам клиентов, так как большая часть вычислительных операций будет производиться на сервере, а также архитектура клиент-сервер довольно гибкая и позволяет администратору сделать локальную сеть более защищенной.

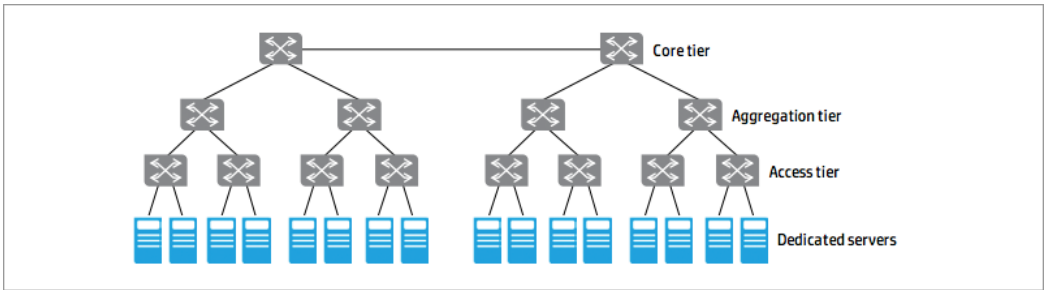
К недостаткам модели взаимодействия клиент-сервер можно отнести то, что стоимость серверного оборудования значительно выше клиентского. Сервер должен обслуживать специально обученный и подготовленный человек. Если в локальной сети ложится сервер, то и клиенты не смогут работать (в качестве частного случая можно привести пример: мощности сервера не всегда хватает, чтобы удовлетворит запросы клиентов, если вы хоть раз работали с биллинговыми системами, то понимаете о чем я: время ожидания ответа от сервера может быть очень большим).

Обзор общих принципов построения многоуровневых приложений. Новые сетевые архитектуры

Традиционная архитектура корпоративной сети включает в себя три уровня: уровень доступа, агрегирования/распределения и ядра. На каждом из них выполняются специфические сетевые функции.

Уровень ядра – основа всей сети. Для достижения максимальной производительности функции маршрутизации и политики управления трафиком выносятся на уровень агрегирования/распределения. Именно он отвечает за надлежащую маршрутизацию пакетов, политики трафика. Задачей уровня распределения является агрегирование/объединение всех коммутаторов уровня доступа в единую сеть. Это позволяет существенно уменьшить количество соединений. Как правило, именно к коммутаторам распределения подключаются самые важные сервисы сети, другие ее модули.

Уровень доступа служит для подключения клиентов к сети. По аналогичной схеме строились и сети ЦОД (центр обработки данных data center).

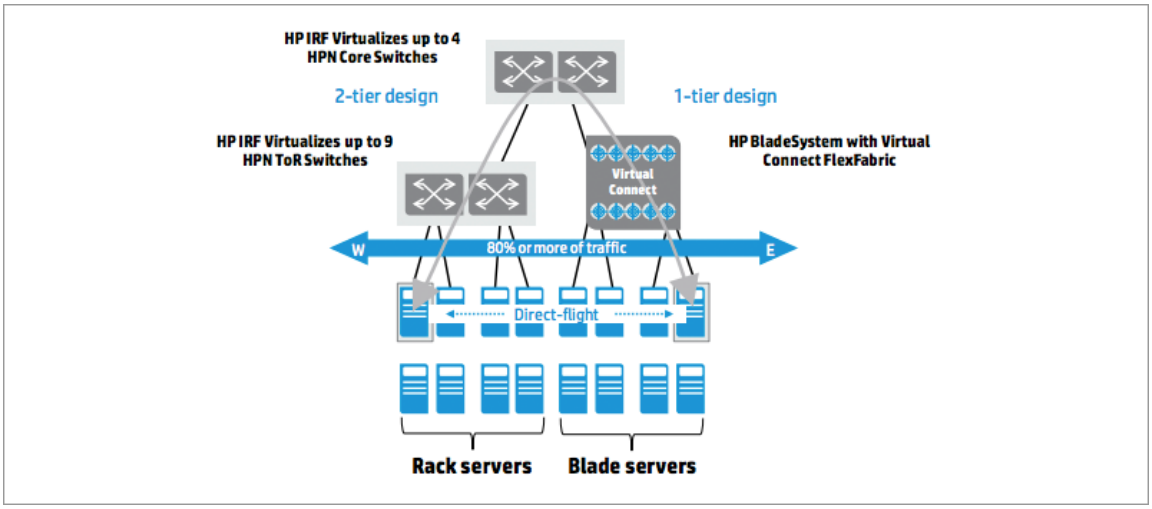


Устаревшая архитектура трехуровневой сети в центре обработки данных.

Традиционные трехуровневые архитектуры ориентированы на клиент-серверную парадигму сетевого трафика. С дальнейшим развитием технологий виртуализации и интеграции приложений возрастает поток сетевого трафика между серверами. Аналитики говорят о смене парадигмы сетевого трафика с направления «север—юг», на «восток—запад», т.е. на существенное преобладание трафика между серверами в отличие от обмена между сервером и клиентами.

При рассмотрении сетевой архитектуры ЦОД, уровень доступа соответствует границе серверной фермы. Трехуровневая архитектура сети в данном случае недостаточно оптимизирована для передачи трафика между отдельными физическими серверами, поскольку вместо сокращения пути передачи пакетов до одного (или максимум двух) сетевых уровней, пакет передается по всем трем, увеличивая задержки за счет паразитного трафика в обоих направлениях.

То есть трафик между серверами проходит через уровни доступа, агрегации, ядра сети и обратно неоптимальным образом, за счет необоснованного увеличения общей длины сетевого сегмента и количества уровней обработки пакетов сетевыми устройствами. Иерархические сети недостаточно приспособлены для обмена данными между серверами, не вполне отвечают требованиям современных ЦОД с высокой плотностью серверных ферм и интенсивным межсерверным трафиком. В такой сети обычно используются традиционные протоколы защиты от петель, резервирования устройств и агрегированных соединений. Ее особенности: существенные задержки, медленная сходимость, статичность, ограниченная масштабируемость и т.п. Вместо традиционной древовидной топологии сети необходимо использовать более эффективные топологии (CLOS/ Leaf-Spine/ Collapsed), позволяющие уменьшить количество уровней и оптимизировать пути передачи пакетов.



HP упрощает архитектуру сети с трёхуровневой (характерной для традиционных сетевых архитектур Cisco) до двух- или одноуровневой. Сейчас тенденция такова, что все больше заказчиков при построении своих сетей ориентируются на построение сетей передачи данных второго уровня (L2) с плоской топологией. В сетях ЦОД переход к ней стимулируется увеличением числа потоков «сервер – сервер» и «сервер – система хранения». Такой подход упрощает планирование сети и внедрение, а также снижает операционные расходы и общую стоимость вложений, делает сеть более производительной.

В ЦОД плоская сеть (уровня L2) лучше отвечает потребностям виртуализации приложений, позволяяэффективно перемещать виртуальные машины между физическими хостами. Еще одно преимущество, которое реализуется при наличии эффективных технологий кластеризации/стекирования – отсутствие необходимости в протоколах STP/RSTP/MSTP. Такая архитектура в сочетании с виртуальными коммутаторами обеспечивает защиту от петель без использования STP, а в случае сбоев сеть сходится на порядок быстрее, чем при использовании традиционных протоколов семейства STP.

Архитектура сети современных ЦОД должна обеспечивать эффективную поддержку передачи больших объемов динамического трафика. Динамический трафик обусловлен существенным ростом количества виртуальных машин и уровня интеграции приложений. Здесь необходимо отметить все возрастающую роль различных технологий виртуализации информационно-технологической (ИТ) инфраструктуры на базе концепции программно-определяемых сетей (SDN).

Концепция SDN в настоящее время широко распространяется не только на уровень сетевой инфраструктуры отдельных площадок, но и на уровни вычислительных ресурсов и систем хранения как в рамках отдельных, так и географически-распределенных ЦОД (примерами последних являются HP Virtual Cloud Networking – VCN и HP Distributed Cloud Networking – DCN).

Ключевой особенностью концепции SDN является объединение физических и виртуальных сетевых ресурсов и их функционала в рамках

единой виртуальной сети. При этом важно понимать, что несмотря на то, что решения сетевой виртуализации (overlay) могут работать поверх любой сети, производительность/доступность приложений и сервисов в значительной степени зависят от работоспособности и параметров физической инфраструктуры (underlay). Таким образом, объединение преимуществ оптимизированной физической и адаптивной виртуальной сетевых архитектур, позволяет строить унифицированные сетевые инфраструктуры для эффективной передачи больших потоков динамического трафика по запросам приложений.

Работа с сетью на основе сокетов и потоков

Процессы

Термин «процесс» впервые появился при разработке операционной системы Multix и имеет несколько определений, которые используются в зависимости от контекста, согласно которым процесс — это:

программа на стадии выполнения

«объект», которому выделено процессорное время

асинхронная работа

Примером контекстной зависимости применения термина «процесс» является ОС Android. В этой системе процесс и приложение не являются тесно связанными сущностями: программы для андроидмогут выглядеть выполняющимися при отсутствии процесса; несколько программ могут использовать один процесс; либо одно приложение может использовать несколько процессов; процесс может присутствовать в системе даже если его приложение бездействует. Отметим, что в этом нет никаких противоречий с теорией операционных систем — объектные библиотеки и загружаемые модули тоже являются процессами.

Для описания состояний процессов используется несколько моделей. Самая простая — модель трех состояний (рис. 1). Она определяет следующие состояния процесса:

состояния выполнения

состояния ожидания

состояния готовности

Выполнение — это *активное состояние*, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.

Ожидание — это *пассивное состояние*, во время которого процесс заблокирован и не может быть выполнен, потому что ожидает какое-то событие, например, ввода данных или освобождения нужного ему устройства.

Готовность — это тоже пассивное состояние, процесс тоже заблокирован, но в отличие от состояния ожидания, он заблокирован не по внутренним причинам (ведь ожидание ввода данных — это внутренняя, «личная» проблема

процесса — он может ведь и не ожидать ввода данных и свободно выполняться — никто ему не мешает), а по внешним, независящим от процесса, причинам.

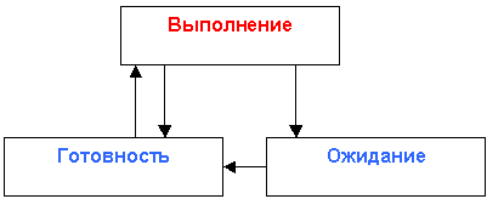


Рис. 1. Модель трех состояний

Когда процесс может перейти в *состояние готовности*? Предположим, что наш процесс выполнялся до ввода данных. До этого момента он был в *состоянии выполнения*, потом перешел в *состояние ожидания*— ему нужно подождать, пока мы введем нужную для работы процесса информацию. Затем процесс хотел уже перейти в состояние выполнения, так как все необходимые ему данные уже введены, но не тут-то было: так как он не единственный процесс в системе, пока он был в состоянии ожидания, его «место под солнцем» занято — процессор выполняет другой процесс. Тогда нашему процессу ничего не остается как перейти в состояние готовности: ждать ему нечего, а выполняться он тоже не может.

Из *состояния готовности* процесс может перейти только в *состояние выполнения*. В состоянии выполнения может находится только один процесс на один процессор. Если у вас n-процессорная машина, у вас одновременно в состоянии выполнения могут быть n процессов.

Из *состояния выполнения* процесс может перейти *либо* в состояние ожидания, либо в состояние готовности. Почему процесс может оказаться в состоянии ожидания, мы уже знаем — ему просто нужны дополнительные данные или он ожидает освобождения какого-нибудь ресурса, например, устройства или файла. В состояние готовности процесс может перейти, если во время его выполнения, квант времени выполнения «вышел». Другими словами, в операционной системе есть специальная программа — планировщик, которая следит за тем, чтобы все процессы выполнялись отведенное им время. Например, у нас есть три процесса. Один из них находится в состоянии выполнения. Два других — в состоянии готовности. Планировщик следит за временем выполнения первого процесса, если «время вышло», планировщик переводит процесс 1 в состояние готовности, а процесс 2 — в состояние выполнения. Затем, когда, время отведенное, на выполнение процесса 2, закончится, процесс 2 перейдет в состояние готовности, а процесс 3 — в состояние выполнения.

Более сложная модель — это модель, состоящая из пяти состояний. В этой модели появилось два дополнительных состояния: *рождение процесса* и *смерть процесса*.

Рождение процесса — это пассивное состояние, когда самого процесса еще нет, но уже готова структура для появления процесса.

Смерть процесса — самого процесса уже нет, но может случиться, что его «место», то есть структура данных , осталась в списке процессов. Такие процессы называются зобми.

Диаграмма модели пяти состояний представлена на рис. 2.

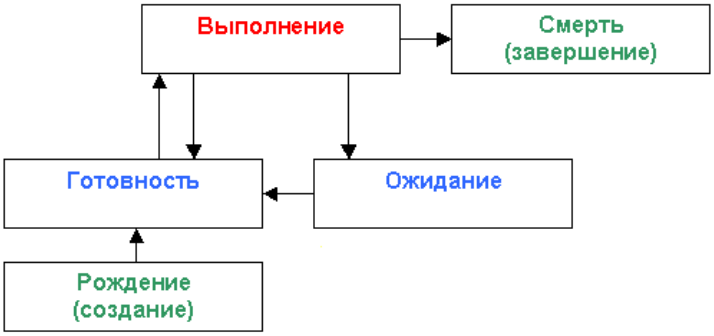


Рис.2. Модель пяти состояний

В ОС РВ время перехода процесса из одного состояния в другое должно быть детерминированно. Функции контроля за временем (deadline) возлагаются на планировщика (о планировании будет сказано далее).

Операции над процессами

Над процессами можно производить следующие операции:

Создание процесса — это переход из состояния рождения в состояние готовности

Уничтожение процесса — это переход из состояния выполнения в состояние смерти

Восстановление процесса — переход из состояния готовности в состояние выполнения

Изменение приоритета процесса — переход из выполнения в готовность

Блокирование процесса — переход в состояние ожидания из состояния выполнения

Пробуждение процесса — переход из состояния ожидания в состояние готовности

Запуск процесса (или его выбор) — переход из состояния готовности в состояние выполнения

Для создания процесса операционной системе нужно:

Присвоить процессу имя

Добавить информацию о процессе в список процессов

Определить приоритет процесса

Сформировать блок управления процессом

Предоставить процессу нужные ему ресурсы

Иерархия процессов

Процесс не может взяться из ниоткуда: его обязательно должен запустить какой-то процесс. Процесс, запущенный другим процессом, называется *дочерним (child) процессом* или *потомком*. Процесс, который запустил новый процесс называется *родительским (parent), родителем* или просто — *предком*. У каждого процесса есть два атрибута — PID (Process ID) -

идентификатор процесса и PPID (Parent Process ID) — идентификатор родительского процесса.

Процессы создают иерархию в виде дерева. Самым «главным» предком, то есть процессом, стоящим на вершине этого дерева, является процесс init (PID=1).

Потоки

Концепция процесса, пришедшая из мира UNIX, плохо реализуется в многозадачной системе, поскольку процесс имеет тяжелый контекст. Возникает понятие потока (thread), который понимается как подпроцесс, или *легковесный процесс (light-weight process)*, выполняющийся в контексте полноценного процесса.

С помощью процессов можно организовать параллельное выполнение программ. Для этого процессы клонируются вызовами `fork`() или `exec`(), а затем между ними организуется взаимодействие средствами IPC. Это довольно дорогостоящий в отношении ресурсов способ.

С другой стороны, для организации параллельного выполнения и взаимодействия процессов можно использовать механизм многопоточности. Основной единицей здесь является поток, который представляет собой облегченную версию процесса. Чтобы понять, в чем состоит его особенность, необходимо вспомнить основные характеристики процесса.

Процесс располагает определенными ресурсами. Он размещен в некотором виртуальном адресном пространстве, содержащем образ этого процесса. Кроме того, процесс управляет другими ресурсами (файлы, устройства ввода/вывода и т.д.).

Процесс подвержен диспетчеризации. Он определяет порядок выполнения одной или нескольких программ, при этом выполнение может перекрываться другими процессами. Каждый процесс имеет состояние выполнения и приоритет диспетчеризации.

Если рассматривать эти характеристики независимо друг от друга (как это принято в современной теории ОС), то:

Владельцу ресурса, обычно называемому процессом или задачей, присущи:

виртуальное адресное пространство;

индивидуальный доступ к процессору, другим процессам, файлам, и ресурсам ввода — вывода.

Модулю для диспетчеризации, обычно называемому потоком или облегченным процессом, присущи:

состояние выполнения (активное, готовность и т.д.);

сохранение контекста потока в неактивном состоянии;

стек выполнения и некоторая статическая память для локальных переменных;

доступ к пространству памяти и ресурсам своего процесса.

Все потоки процесса разделяют общие ресурсы. Изменения, вызванные одним потоком, становятся немедленно доступны другим.

При корректной реализации потоки имеют определенные преимущества перед процессами. Им требуется:

меньше времени для создания нового потока, поскольку создаваемый поток использует адресное пространство текущего процесса;

меньше времени для завершения потока;

меньше времени для переключения между двумя потоками в пределах процесса;

меньше коммуникационных расходов, поскольку потоки разделяют все ресурсы, и в частности адресное пространство. Данные, продуцируемые одним из потоков, немедленно становятся доступными всем другим потокам.

Преимущества многопоточности

Если операционная система поддерживает концепции потоков в рамках одного процесса, она называется многопоточной. Многопоточные приложения имеют ряд преимуществ:

Улучшенная реакция приложения — любая программа, содержащая много не зависящих друг от друга действий, может быть перепроектирована так, чтобы каждое действие выполнялось в отдельном потоке. Например, пользователь многопоточного интерфейса не должен ждать завершения одной задачи, чтобы начать выполнение другой.

Более эффективное использование мультипроцессирования — как правило, приложения, реализующие параллелизм через потоки, не должны учитывать число доступных процессоров. Производительность приложения равномерно увеличивается при наличии дополнительных процессоров. Численные алгоритмы и приложения с высокой степенью параллелизма, например перемножение матриц, могут выполняться намного быстрее.

Улучшенная структура программы — некоторые программы более эффективно представляются в виде нескольких независимых или полуавтономных единиц, чем в виде единой монолитной программы. Многопоточные программы легче адаптировать к изменениям требований пользователя.

Эффективное использование ресурсов системы — программы, использующие два или более процессов, которые имеют доступ к общим данным через разделяемую память, содержат более одного потока управления. При этом каждый процесс имеет полное адресное пространство и состояние в операционной системе. Стоимость создания и поддержания большого количества служебной информации делает каждый процесс более затратным, чем поток. Кроме того, разделение работы между процессами может потребовать от программиста значительных усилий, чтобы обеспечить связь между потоками в различных процессах или синхронизировать их действия.

Литература

Гуриков, С.Р. Информатика : учебник / С.Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2021. – 463 с. – (Высшее образование: Бакалавриат)

Тема 3. Обзор базовых конструкций и основных элементов языка

Переменные и типы. Преобразование и приведение типов, расширение типов. Динамическая инициализация, область действия и время жизни переменных. Циклы и логика. Массивы и строки

Правила описания и использования переменных в выражениях

Для хранения данных в программе предназначены переменные. Переменная представляет именованную область памяти, которая хранит значение определенного типа. Каждая переменная имеет тип, имя и значение. Тип определяет, какую информацию может хранить переменная или диапазон допустимых значений.

Переменные объявляются следующим образом:

```
1    тип_данных имя_переменной;
```

Например, определим переменную, которая будет называться x и будет иметь тип int:

```
1    int x;
```

В этом выражении мы объявляем переменную x типа int. То есть x будет хранить некоторое число не больше 4 байт.

В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

– имя может содержать любые алфавитно-цифровые символы, а также знак подчеркивания, при этом первый символ в имени не должен быть цифрой

– в имени не должно быть знаков пунктуации и пробелов

– имя не может быть ключевым словом языка Java

Кроме того, при объявлении и последующем использовании надо учитывать, что Java – регистрозависимый язык, поэтому следующие объявления int num; и int NUM; будут представлять две разных переменных.

Объявив переменную, мы можем присвоить ей значений:

```
1    int x;    // объявление переменной
2    x = 10;    // присвоения значения
3    System.out.println(x);  // 10
```

Ключевое слово var

Начиная с Java 10 в язык было добавлено ключевое слово var, которое также позволяет определять переменную:

```
1    var x = 10;
2    System.out.println(x);  // 10
```

Слово var ставится вместо типа данных, а сам тип переменной выводится из того значения, которое ей присваивается. Например, переменой x приваривается число 10, значит, переменная будет представлять тип int.

Но если переменная объявляется с помощью var, то мы обязательно должны инициализировать ее, то есть предоставить ей начальное значение, иначе мы получим ошибку, как, например, в следующем случае:

```
1    var x;     // ! Ошибка, переменная не инициализирована
2    x = 10;
```

Инициализация переменных

Также можно привоить значение переменной при ее объявлении. Этот процесс называется инициализацией:

```
1    int x = 10;    // объявление и инициализация переменной
2    System.out.println(x);  // 10
```

Если мы не присвоим переменной значение до ее использования, то мы можем получить ошибку, например, в следующем случае:

```
1    int x;
2    System.out.println(x);
```

Через запятую можно объявить сразу несколько переменных одного типа:

```
1    int x, y;
2    x = 10;
3    y = 25;
4    System.out.println(x);  // 10
5    System.out.println(y);  // 25
```

Также можно их сразу инициализировать:

```
1    int x = 8, y = 15;
2    System.out.println(x);  // 8
3    System.out.println(y);  // 15
```

Отличительной особенностью переменных является то, что мы можем в процессе работы программы изменять их значение:

```
1    int x = 10;
2    System.out.println(x);  // 10
```

```
3     x = 25;
4     System.out.println(x);  // 25
```

Время жизни и область видимости переменных

В различных языках программирования существуют различные типы или классы переменных — локальные, глобальные, статические и т.п. В Java только один тип переменных — локальные переменные. Время жизни переменной в Java определяется правилом:

Переменная создается в точке ее описания и существует до момента окончания того блока, в котором находится данное описание.

В Java блок — это то, что начинается открывающей фигурной скобкой '{' и заканчивается закрывающей фигурной скобкой '}'.

Областью видимости переменной (scope) является фрагмент программы от точки ее описания до конца текущего блока.

Область видимости — это статическое понятие, имеющее отношение к какому-то фрагменту текста программы. Время жизни, в отличие от области видимости, — это понятие динамики выполнения программы. Время жизни переменных в Java совпадает с их областью видимости с учетом отличия самих этих понятий.

Если в блоке, где описана данная переменная, вложены другие блоки, то переменная доступна в этих блоках (обычная практика языков программирования). Но, в отличие от многих других языков, в Java запрещено переопределять переменную во вложенных блоках (т.е. описывать другую переменную с тем же именем).

Рассмотрим примеры, демонстрирующие эти понятия.

Пример 1

```
. . .  // предшествующая часть программы
{     // начало блока
   . . .  // какие-то операторы внутри блока
   int x = 1;   // 1-я точка описания
   {          // еще один блок
      . . .   // какие-то операторы внутри блока
      int y;   // 2-я точка описания
      . . .   // какие-то операторы внутри блока
```

```
  } // переменная y уничтожается и она больше не видна
  . . . // какие-то операторы внутри блока
} // переменная x уничтожается и она больше не видна
. . . // последующая часть программы
```

Пример 2

```
{    // начало блока
  int x = 1;
  long y;
  {        // еще один блок
    int x;  // !!! ошибка
    int y;  // !!! ошибка
    . . .  // какие-то операторы внутри блока
  }
  . . . // какие-то операторы внутри блока
}
```

Смысл примеров указан в комментариях в самих примерах. Первый пример просто показывает, в каких пределах текста видна та или иная переменная (область видимости) и, когда она создается и уничтожается (время жизни). Второй пример содержит ошибки. Он демонстрирует запрет переопределять переменные во вложенных блоках.

Константы и их типы в программе

Кроме переменных, в Java для хранения данных можно использовать константы. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется также, как и переменная, только вначале идет ключевое слово final:

```
1    final int LIMIT = 5;
2    System.out.println(LIMIT);  // 5
3    // LIMIT=57; // так мы уже не можем написать, так как LIMIT - константа
```

Как правило, константы имеют имена в верхнем регистре.

Константы позволяют задать такие переменные, которые не должны больше изменяться. Например, если у нас есть переменная для хранения числа pi, то мы можем объявить ее константой, так как ее значение постоянно.

Типы данных (по способу хранения в памяти): типы-значения, ссылочные типы

Java как и все языки программирования имеет ссылочные типы. Эти типы передаются по ссылкам. Кажется, что это противоречит Java и это правда. Ведь в Java передается копия ссылки, а не сама ссылка.

Рассмотрим ссылочные типы в Java. К таким типам относятся массивы, классы и String. Рассмотрим поподробнее.

Эти типы хранятся в памяти, как и все, но переменные хранят ссылки на ячейки памяти в которых хранятся эти типы, и если мы копируем эту ссылку в другую переменную к примеру, то сможем потом в дальнейшем в коде изменить объект, хранящийся в памяти по ссылке и все переменные, что указывали на этот объект, будут иметь обновленный объект, но НЕ ссылку.

Рассмотрим следующий пример.

```
class A {

  private String innerValue;

  A() {
     this.innerValue = "Empty";
  }

  A(String innerValue) {
     this.innerValue = innerValue;
  }

  String getInnerValue() {
     return innerValue;
  }

  void setInnerValue(String innerValue) {
     this.innerValue = innerValue;
  }

}

public class Main {

  private static void changeObject(A object) {
     object.setInnerValue("changed value");
  }

  public static void main(String[] args) {
```

```
    A object = new A("simple string");
    System.out.println(object.getInnerValue()); // simple string

    changeObject(object);
    System.out.println(object.getInnerValue()); // changed value

  }
}
```

В этом примере создается класс A, затем создается его объект, после чего мы выводим на консоль текущее хранимое значение в объекте A, затем передаем объект A в метод changeObject, который в свою очередь устанавливает новое значение для innerValue. И дальше мы выводим измененное значение на консоль. Значение в объекте поменяется, хоть мы его и передаем в отдельный метод. В принципе тут все логично. Просто нужно запомнить, что все-все объекты передаются по копии ссылки. Переменная object при создании объекта A сохраняет в себя ссылку на этот объект, дальше, когда мы передаем этот объект в метод changeObject в параметр метода копируется ссылка на объект A. Теперь у нас есть две переменных, которые имеют ссылку на один и тот же объект в памяти. Это локальная переменная object метода main и аргумент метода changeObject. Если мы изменяем объект по ссылке в методе changeObject, то он меняется и для ссылки в переменной object метода main. НО заметьте, что саму ссылку мы для локальной переменной object метода main из метода changeObject изменить не сможем. Это значит, что если мы присвоим аргументу object метода changeObject NULL, то это никак не повлияет на переменную object в методе main.

Массивы.

В Java все примитивные типы передаются по значению, а не по ссылке, то есть.

```
int a = 10;
int b = a;

b = 44;

System.out.println(a); // 10
```

Получается, что мы, изменив переменную b, никак не влияем на переменную a. А теперь следующий пример.

```
public class Main {

  private static void changeArray(int[] input) {
    input[0] = 2;
  }
```

```
  public static void main(String[] args) {

    int[] input = { 1 };
    System.out.println(input[0]); // 1

    changeArray(input);
    System.out.println(input[0]); // 2

  }
}
```

Массив передается по ссылке и значение 0-го элемента меняется. Для меня это казалось нелогичным, когда я изучал Java, потому что примитивы передаются по значениям, а массивы по ссылкам. Сейчас это кажется очевидным решением, потому что массивы могут хранить не только примитивы, но и объекты, да и в принципе, чтобы можно было его менять на лету, но с этим нужно быть осторожным, чтобы потом не засесть на несколько часов с поиском проблемы от которой данные в массиве меняются :).

А теперь разберемся со строками. Начнем с примера, который будет разъяснен позже.

```
public class Main {

  public static void main(String[] args) {

    String s1 = "str1";
    String s2 = s1;

    System.out.println(s1 == s2); // true
    System.out.println(s1.equals(s2)); // true

    String s3 = new String(s2);

    System.out.println(s2 == s3); // false
    System.out.println(s2.equals(s3)); // true

    s1 += "Ex";
    System.out.println(s1 == s2); // false

    s1 = s2;
    System.out.println(s1 == s2); // true

  }
}
```

Вначале мы создаем две строки, первую со значением str1, а вторую из строки s1.

Далее мы их сравниваем s1 == s2 (для тех кто еще не знает, тут сравниваются ссылки на ячейку памяти, а не значения строк), в результате true. Да, эти две строки указывают на одну ячейку памяти. Значит это ссылка, то есть мы предполагаем, если мы изменим строку s2 то и должна поменяться строка s1. Но нет. Строки в Java иммутабельные, и если мы как-то изменить одну строку, то для нее создается новая ячейка в памяти и туда сохраняется измененная строка, соответственно и ссылка на нее тоже меняется и s1 == s2 уже false. Но если вернуть прежнее значение строке s2, то обе строки снова будут указывать на одно значение в памяти. Это вроде правильно называется как-то “Пул значений”.

Типы-значения

Как таковых типов-значений в Java нет, поэтому пример описания данной конструкции приведу на C#.

Ранее мы рассматривали следующие элементарные типы данных: int, byte, double, string, object и др. Также есть сложные типы: перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые значимыми типами, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.

Типы значений:

Целочисленные типы (byte, sbyte, char, short, ushort, int, uint, long, ulong)

Типы с плавающей запятой (float, double)

Тип decimal

Тип bool

Перечисления enum

Структуры (struct)

Ссылочные типы:

Тип object

Тип string

Классы (class)

Интерфейсы (interface)

Делегаты (delegate)

В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу-вверх: каждый новый добавляемый элемент помещаются поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек – это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека, устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Например:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           Calculate(5);
6           Console.ReadKey();
7       }
8
9       static void Calculate(int t)
10      {
11          int x = 6;
12          int y = 7;
13          int z = y + t;
14      }
15  }
```

При запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:

При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода.

Когда метод отработает, все эти переменные уничтожаются, и память в стеке очищается.

Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

Ссылочные типы хранятся в куче или хипе, которую можно представить, как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, то ссылка из стека удаляется, и память очищается. После этого в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, и удаляет этот объект и очищает память.

Так, в частности, если мы изменим метод Calculate следующим образом:

```
1    static void Calculate(int t)
2    {
3        object x = 6;
4        int y = 7;
5        int z = y + t;
6    }
```

То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.

Составные типы

Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

```
1    class Program
2    {
3        private static void Main(string[] args)
4        {
5            State state1 = new State(); // State - структура, ее данные размещены в стеке
6            Country country1 = new Country(); // Country - класс, в стек помещается ссылка на адрес в хипе
```

```
7                                   // а в хипе располагаются все данные объекта country1
8           }
9       }
10      struct State
11      {
12          public int x;
13          public int y;
14          public Country country;
15      }
16      class Country
17      {
18          public int x;
19          public int y;
20      }
```

Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект..

Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе.

Однако в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?

```
1   private static void Main(string[] args)
2   {
3       State state1 = new State();
4       state1.country = new Country();
5       Country country1 = new Country();
6   }
```

Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:

Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

```
1       private static void Main(string[] args)
2       {
3           State state1 = new State(); // Структура State
4           State state2 = new State();
5           state2.x = 1;
6           state2.y = 2;
7           state1 = state2;
8           state2.x = 5; // state1.x=1 по-прежнему
9           Console.WriteLine(state1.x); // 1
10          Console.WriteLine(state2.x); // 5
11
12          Country country1 = new Country(); // Класс Country
13          Country country2 = new Country();
14          country2.x = 1;
15          country2.y = 4;
16          country1 = country2;
17          country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и country1 и country2
18                      // указывают на один объект в хипе
19          Console.WriteLine(country1.x); // 7
20          Console.WriteLine(country2.x); // 7
21
22          Console.Read();
23      }
```

Так как state1 - структура, то при присвоении state1 = state2 она получает копию структуры state2. А объект класса country1 при присвоении country1 = country2; получает ссылку на тот же объект, на который указывает country2. Поэтому с изменением country2, так же будет меняться и country1.

Ссылочные типы внутри типов значений

Теперь рассмотрим более изощренный пример, когда внутри структуры может быть переменная ссылочного типа, например, какого-нибудь класса:

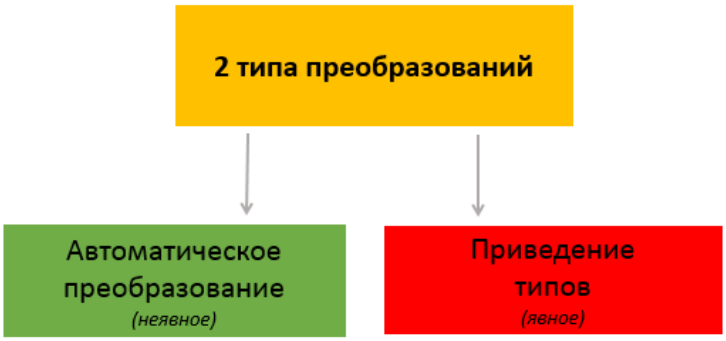
```
class Program
{
    private static void Main(string[] args)
    {
        State state1 = new State();
        State state2 = new State();

        state2.country = new Country();
        state2.country.x = 5;
        state1 = state2;
        state2.country.x = 8; // теперь и state1.country.x=8, так как state1.country и state2.country
                    // указывают на один объект в хипе
        Console.WriteLine(state1.country.x); // 8
        Console.WriteLine(state2.country.x); // 8

        Console.Read();
    }
}
struct State
{
    public int x;
    public int y;
    public Country country;
}
class Country
{
    public int x;
    public int y;
}
```

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в хипе. И при присвоении двух структур state1 =

state2; структура state1 также получит ссылку на объект country в хипе. Поэтому изменение state2.country повлечет за собой также изменение state1.country.

Приведение типов данных

В Java существует 2 типа преобразований:



Автоматическое преобразование

Ну, что ж, давайте попробуем разобраться что такое "автоматическое преобразование". Переменная - это некоторый «контейнер», в котором может храниться значение для дальнейшего использования в программе. Каждый тип переменной имеет свой диапазон допустимых значений и объем занимаемой памяти. Вот она табличка, где это все расписано:

	ТИП ДАННЫХ	ДИАПАЗОН ДОПУСТИМЫХ ЗНАЧЕНИЙ	ОБЪЁМ ЗАНИМАЕМОЙ ПАМЯТИ
целочисленные	byte	от -128 до 127	1 байт
	short	от -32768 до 32767	2 байта
	int	от -2147483648 до 2147483647	4 байта
	long	от -9223372036854775808 до 9223372036854775807	8 байт
с плавающей точкой	float	от -3.4E+38 до 3.4E+38	4 байта
	double	от -1.7E+308 до 1.7E+308	8 байт
символы	char	от 0 до 65536	2 байта
логические	boolean	true или false	Для хранения значения этого типа достаточно 1 бита, но в реальности память такими порциями не выделяется, поэтому переменные этого типа могут быть по-разному упакованы виртуальной машиной

Например:

1. byte и short. byte имеет меньший диапазон допустимых значений, чем short. То есть byte это как бы коробочка поменьше, а short - это коробочка побольше. И значит, мы можем byte вложить в short.

2. byte и int. byte имеет меньший диапазон допустимых значений, чем int. То есть byte это как бы коробочка поменьше, а int - это коробочка побольше. И значит, мы можем byte вложить в int.

3. int и long. int имеет меньший диапазон допустимых значений, чем long. То есть int это как бы коробочка поменьше, а long - это коробочка побольше. И значит, мы можем int вложить в long.

Это и есть пример автоматического преобразования.

Расширяющие автоматические преобразования представлены следующими цепочками:

byte -> short -> int -> long

int -> double

short -> float -> double

char -> int

Давайте рассмотрим, как это работает на практике.

Пример №1

Код №1 - если Вы запустите это код на своем компьютере, в консоли будет выведено число 15

```
1 class Test {
2
3         public static void main(String[] args) {
4                 byte a = 15;
5                 byte b = a;
6                 System.out.println(b);
7         }
8 }
```

Код №2 - если Вы запустите это код на своем компьютере, в консоли будет выведено число 15

```
1 class Test {
2
3        public static void main(String[] args) {
4                byte a = 15;
5                int b = a;
6                System.out.println(b);
7        }
8 }
```

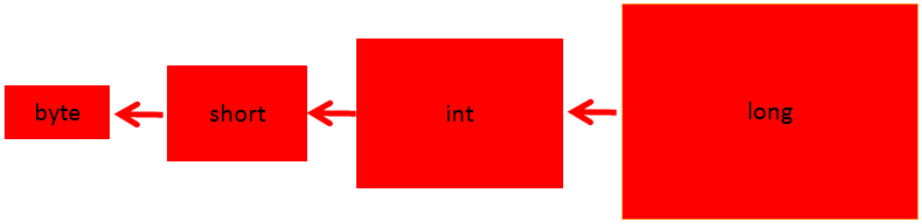
В коде №2 присутствует автоматическое преобразование типов, а в коде №1 – нет.

Хотя число, в принципе, одно и то же, но теперь оно находится в большем контейнере, который занимает больше места на диске. При этом, JVM выполняет автоматические преобразования за Вас. Она знает, что int больше чем byte.

Приведение типов

Другое дело если вы пытаетесь переложить что-то из большего контейнера в более маленький.

Вы можете знать, что в большем контейнере лежит то, что поместиться и в маленьком – но об этом не знает JVM, и пытается предохранить вас от ошибок.

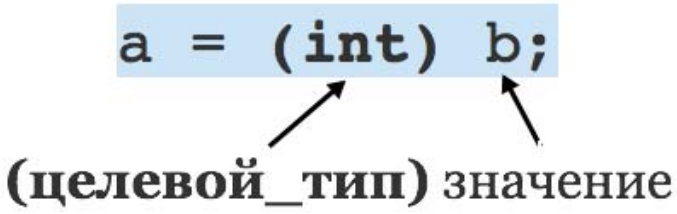


Поэтому, вы должны «прямо сказать», что ситуация под контролем:

```
1 class Test {
```

```
2       public static void main(String[] args) {
3               int a=0;
4               long b=15;
5               a = (int) b;
6       }
7 }
```

Тут мы дописали (int) перед b. Если бы переменная a была, к примеру, типа byte, в скобках бы стояло (byte). Общая формула выглядит так:



Она говорит "сделай из (большего) значения b переменную нужного мне (целевого) типа int".

Если что-то пошло не так.

До этого мы рассматривали ситуации, предполагая, что мы точно знаем, что делаем. Но что если попытаться поместить в контейнер то, что туда не помещается?

Оказывается, в контейнере останется лишь то, что туда «влезло». К примеру, у чисел с плавающей точкой будет «отсекаться» дробная часть:

```
1 //пример 1
2
3 class Test {
4       public static void main(String[] args) {
5               double a=11.2345;
6               int b=(int)a;
7               System.out.println(b); // в консоли получится число 11
8       }
9 }
```

Надо помнить, что дробная часть не округляется, а отбрасывается.

Математика	Программирование
2, 1 → 2	2,1 → 2
2,7 → 3	2,7 → 2

А что будет, если мы попытаемся поместить число, которое выходит за допустимые границы? Например, если в byte (диапазон byte от -128 до 127) положить число 128? Думаете, мы получим 1? Нет. Мы получим -128:

```
1 class Test {
2         public static void main(String[] args) {
3                 double a=128;
4                 byte b=(byte)a;
5                 System.out.println(b); //в консоли увидим -128
6         }
7 }
```

Значение переменной при таком преобразовании можно рассчитать, но цель программиста – не допускать ситуации, когда значение выходит за допустимые границы, поскольку это может привести к неправильной работе программы.

Автоматические преобразования с потерей точности

Некоторые преобразования могут производиться автоматически между типами данных одинаковой разрядности или даже от типа данных с большей разрядностью к типа с меньшей разрядностью. Это следующие цепочки преобразований: int -> float, long -> float и long -> double произволятся без ошибок, но при преобразовании мы можем столкнуться с потерей информации.

Например:

```
1    int a = 2147483647;
2    float b = a;          // от типа int к типу float
3    System.out.println(b);  // 2.14748365E9
```

Явные преобразования

Во всех остальных преобразованиях примитивных типов явным образом применяется опрерация преобразования типов. Обычно это сужающие

преобразования (narrowing) от типа с большей разрядностью к типу с меньшей разрядностью:

```
1    long a = 4;
2    int b = (int) a;
```

Потеря данных при преобразовании

Применении явных преобразований мы можем столкнуться с потерей данных. Например, следующем коде у нас не возникнет никаких проблем:

```
1    int a = 5;
2    byte b = (byte) a;
3    System.out.println(b);    // 5
```

Число 5 вполне укладывается в диапазон значений типа byte, поэтому после преобразования переменная b будет равна 5. Но что будет в следующем случае:

```
1    int a = 258;
2    byte b = (byte) a;
3    System.out.println(b);    // 2
```

Результатом будет число 2. В данном случае число 258 вне диапазона для типа byte (от -128 до 127), поэтому произойдет усечение значения. Почему результатом будет именно число 2?

Число a, которое равно 258, в двоичном системе будет равно 00000000 00000000 00000001 00000010. Значения типа byte занимают в памяти только 8 бит. Поэтому двоичное представление числа int усекается до 8 правых разрядов, то есть 00000010, что в десятичной системе дает число 2.

Преобразования при операциях

Нередки ситуации, когда приходится применять различные операции, например, сложение и произведение, над значениями разных типов. Здесь также действуют некоторые правила:

– если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double

– если предыдущее условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float

– если предыдущие условия не соблюдены, один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long

– иначе все операнды операции преобразуются к типу int

Примеры преобразований:

```
1    int a = 3;
2    double b = 4.6;
3    double c = a+b;
```

Так как в операции участвует значение типа double, то и другое значение приводится к типу double и сумма двух значений a+b будет представлять тип double.

Другой пример:

```
1    byte a = 3;
2    short b = 4;
3    byte c = (byte)(a+b);
```

Две переменных типа byte и short (не double, float или long), поэтому при сложении они преобразуются к типу int, и их суммаа+b представляет значение типа int. Поэтому если затем мы присваиваем эту сумму переменной типа byte, то нам опять надо сделать преобразование типов к byte.

Если в операциях участвуют данные типа char, то они преобразуются в int:

```
1    int d = 'a' + 5;
2    System.out.println(d);  // 102
```

2.14 Алгоритмы ветвления

Одним из фундаментальных элементов многих языков программирования являются условные конструкции. Данные конструкции позволяют направить работу программы по одному из путей в зависимости от определенных условий.

В языке Java используются следующие условные конструкции: if..else и switch..case

Конструкция if/else

Выражение if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
1    int num1 = 6;
2    int num2 = 4;
3    if(num1>num2){
4        System.out.println("Первое число больше второго");
```

```
5    }
```

После ключевого слова if ставится условие. И если это условие выполняется, то срабатывает код, который помещен в далее в блоке if после фигурных скобок. В качестве условий выступает операция сравнения двух чисел.

Так как, в данном случае первое число больше второго, то выражение num1 > num2 истинно и возвращает значение true. Следовательно, управление переходит в блок кода после фигурных скобок и начинает выполнять содержащиеся там инструкции, а конкретно метод System.out.println("Первое число больше второго");. Если бы первое число оказалось бы меньше второго или равно ему, то инструкции в блоке if не выполнялись бы.

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

```
1    int num1 = 6;
2    int num2 = 4;
3    if(num1>num2){
4        System.out.println("Первое число больше второго");
5    }
6    else{
7        System.out.println("Первое число меньше второго");
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. С помощью выражения else if, мы можем обрабатывать дополнительные условия:

```
1    int num1 = 6;
2    int num2 = 8;
3    if(num1>num2){
4        System.out.println("Первое число больше второго");
5    }
6    else if(num1<num2){
7        System.out.println("Первое число меньше второго");
8    }
9    else{
```

```
10        System.out.println("Числа равны");
11    }
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
1    int num1 = 8;
2    int num2 = 6;
3    if(num1 > num2 && num1>7){
4        System.out.println("Первое число больше второго и больше 7");
5    }
```

Здесь блок if будет выполняться, если num1 > num2 равно true и одновременно num1>7 равно true.

Конструкция switch

Конструкция switch/case аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

```
1     int num = 8;
2     switch(num){
3
4         case 1:
5             System.out.println("число равно 1");
6             break;
7         case 8:
8             System.out.println("число равно 8");
9             num++;
10            break;
11        case 9:
12            System.out.println("число равно 9");
13            break;
14        default:
15            System.out.println("число не равно 1, 8, 9");
16    }
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями,

помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце блока case ставится оператор break, чтобы избежать выполнения других блоков. Например, если бы убрали бы оператор break в следующем случае:

```
1     case 8:
2        System.out.println("число равно 8");
3        num++;
4     case 9:
5        System.out.println("число равно 9");
6        break;
```

то так как у нас переменная num равно 8, то выполнился бы блок case 8, но так как в этом блоке переменная num увеличивается на единицу, оператор break отсутствует, то начал бы выполняться блок case 9.

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default, как в примере выше. Хотя блок default необязателен.

Также можго определить одно действие сразу для нескольких блоков case подряд:

```
1     int num = 3;
2     int output = 0;
3     switch(num){
4
5        case 1:
6           output = 3;
7           break;
8        case 2:
9        case 3:
10       case 4:
11          output = 6;
12          break;
13       case 5:
14          output = 12;
```

```
15          break;
16        default:
17          output = 24;
18    }
19    System.out.println(output);
```

2.15 Логические и тернарные операции

Побитовые или поразрядные операции выполняются над отдельными разрядами или битами чисел. В данных операциях в качестве операндов могу выступать только целые числа.

Каждое число имеет определенное двоичное представление. Например, число 4 в двоичной системе 100, а число 5 - 101 и так далее.

К примеру, возьмем следующие переменны:

```
1    byte b = 7;    // 0000 0111
2    short s = 7;   // 0000 0000 0000 0111
```

Тип byte занимает 1 байт или 8 битов, соответственно представлен 8 разрядами. Поэтому значение переменной b в двоичном коде будет равно 00000111. Тип short занимает в памяти 2 байта или 16 битов, поэтому число данного типа будет представлено 16 разрядами. И в данном случае переменная s в двоичной системе будет иметь значение 0000 0000 0000 0111.

Для записи чисел со знаком в Java применяется дополнительный код (two’s complement), при котором старший разряд является знаковым. Если его значение равно 0, то число положительное, и его двоичное представление не отличается от представления беззнакового числа. Например, 0000 0001 в десятичной системе 1.

Если старший разряд равен 1, то мы имеем дело с отрицательным числом. Например, 1111 1111 в десятичной системе представляет -1. Соответственно, 1111 0011 представляет -13.

Логические операции

Логические операции над числами представляют поразрядные операции. В данном случае числа рассматриваются в двоичном представлении, например, 2 в двоичной системе равно 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

& (логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

```
1    int a1 = 2; //010
2     int b1 = 5;//101
3     System.out.println(a1&b1); // результат 0
4
5    int a2 = 4; //100
6    int b2 = 5; //101
7    System.out.println(a2 & b2); // результат 4
```

В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядное умножение чисел (0*1, 1*0, 0*1) дает результат 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому здесь результатом операции (1*1, 0*0, 0 *1) = 100 будет число 4 в десятичном формате.

| (логическое сложение)

Данная операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица (операция "логическое ИЛИ"). Например:

```
1    int a1 = 2; //010
2    int b1 = 5;//101
3    System.out.println(a1|b1); // результат 7 - 111
4    int a2 = 4; //100
5    int b2 = 5;//101
6    System.out.println(a2 | b2); // результат 5 - 101
```

^ (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

```
1    int number = 45; // 1001 Значение, которое надо зашифровать - в двоичной форме 101101
2    int key = 102; //Ключ шифрования - в двоичной системе 1100110
3    int encrypt = number ^ key; //Результатом будет число 1001011 или 75
4    System.out.println("Зашифрованное число: " +encrypt);
```

```
5

6    int decrypt = encrypt ^ key; // Результатом будет исходное число 45

7    System.out.println("Расшифрованное число: " + decrypt);
```

Здесь также производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Например, результатом выражения 9^5 будет число 12. А чтобы расшифровать число, мы применяем обратную операцию к результату.

~ (логическое отрицание)

Поразрядная операция, которая инвертирует все разряды числа: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

```
1    byte a = 12;            // 0000 1100

2    System.out.println(~a);    //  1111 0011   или -13
```

Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

a<<b - сдвигает число a влево на b разрядов. Например, выражение 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, в результате получается число 1000 или число 8 в десятичном представлении.

a>>b - смещает число a вправо на b разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичной системе 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

a>>>b - в отличие от предыдущих типов сдвигов данная операция представляет беззнаковый сдвиг - сдвигает число a вправо на b разрядов. Например, выражение -8>>>2 будет равно 1073741822.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два, так как операция сдвига на аппаратном уровне менее дорогостоящая операция в отличие от операции деления или умножения.

Также в Java есть логические операции, которые также представляют условие и возвращают true или false и обычно объединяют несколько операций сравнения. К логическим операциям относят следующие:

|

c=a|b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

&

c=a&b; (c равно true, если и a, и b равны true, иначе c будет равно false)

!

c=!b; (c равно true, если b равно false, иначе c будет равно false)

^

c=a^b; (c равно true, если либо a, либо b (но не одновременно) равны true, иначе c будет равно false)

||

c=a||b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

&&

c=a&&b; (c равно true, если и a, и b равны true, иначе c будет равно false)

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны.

Выражение c=a|b; будет вычислять сначала оба значения - a и b и на их основе выводить результат.

В выражении же c=a||b; вначале будет вычисляться значение a, и если оно равно true, то вычисление значения b уже смысла не имеет, так как у нас в любом случае уже c будет равно true. Значение b будет вычисляться только в том случае, если a равно false

То же самое касается пары операций &/&&. В выражении c=a&b; будут вычисляться оба значения - a и b.

В выражении же c=a&&b; сначала будет вычисляться значение a, и если оно равно false, то вычисление значения b уже не имеет смысла, так как значение c в любом случае равно false. Значение b будет вычисляться только в том случае, если a равно true

Таким образом, операции || и && более удобны в вычислениях, позволяя сократить время на вычисление значения выражения и тем самым повышая производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

Примеры:

```
1   boolean a1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
2   boolean a2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
3   boolean a3 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
4   boolean a4 = (50 > 6) && (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается true
```

```
5    boolean a5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
6    boolean a6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false
```

Тернарная операция

Тернарную операция имеет следующий синтаксис: [первый операнд - условие] ? [второй операнд] : [третий операнд]. Таким образом, в этой операции участвуют сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
1    int x=3;
2    int y=2;
3    int z = x<y? (x+y) : (x-y);
4    System.out.println(z);
```

Здесь результатом тернарной операции является переменная z. Сначала проверяется условие x<y. И если оно соблюдается, то z будет равно второму операнду - (x+y), иначе z будет равно третьему операнду.

2.16 Циклические алгоритмы

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз. В языке Java есть следующие виды циклов:

for

while

do...while

Цикл for

Цикл for имеет следующее формальное определение:

```
1    for ([инициализация счетчика]; [условие]; [изменение счетчика])
2    {
3        // действия
4    }
```

Рассмотрим стандартный цикл for:

```
1    for (int i = 1; i < 9; i++){
```

```
2       System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
3   }
```

Первая часть объявления цикла - int i = 1 создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и любой другой числовой тип, например, float. Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.

В итоге блок цикла сработает 8 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
1   int i = 1;
2   for (; ;){
3       System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
4   }
```

Определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). Теперь нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Либо можно опустить ряд блоков:

```
1   int i = 1;
2   for (; i<9;){
3       System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
4       i++;
5   }
```

Этот пример эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Цикл for может определять сразу несколько переменных и управлять ими:

```
1   int n = 10;
```

```
2     for(int i=0, j = n - 1; i < j; i++, j--){
3
4         System.out.println(i * j);
5     }
```

Цикл в **стиле for each** предназначен для строго последовательного выполнения повторяющихся действий над коллекцией объектов вроде массива.

Преимущество такого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово, а уже существующий код не нарушается. Цикл for в стиле for each называется также усовершенствованным циклом for. Общая форма разновидности цикла for в стиле for each имеет следующий вид:

for (тип итерационная_переменная : коллекция) блок_операторов

где тип обозначает конкретный тип данных; итерационна я_ переменна я - имя итерационной переменной, которая последовательно принимает значения из коллекции: от первого и до последнего: а коллекция - перебираемую в цикле коллекцию.

В цикле for можно перебирать разные типы коллекций, но здесь для этой цели будут использоваться только массивы. (Другие типы коллекций, которые можно перебирать в цикле for, определены в каркасе коллекций Collection Framework) На каждом шаге цикла из коллекции извлекается очередной элемент, который сохраняется в указанной итерационной_ переменной. Цикл выполняется до тех пор, пока из коллекции не будут извлечены все элементы.

Поскольку итерационная переменная получает значения из коллекции, тип должен совпадать (или быть совместимым) с типом элементов, хранящихся в коллекции. Таким образом, при переборе массива тип должен быть совместим с типом элемента массива.

Формальное ее объявление:

```
1     for (тип_данных название_переменной : контейнер){
2         // действия
3     }
```

Например:

```
1    int[] array = new int[] { 1, 2, 3, 4, 5 };
2    for (int i : array){
3
4        System.out.println(i);
5    }
```

В качестве контейнера в данном случае выступает массив данных типа int. Затем объявляется переменная с типом int

То же самое можно было бы сделать и с помощью обычной версии for:

```
1    int[] array = new int[] { 1, 2, 3, 4, 5 };
2    for (int i = 0; i < array.length; i++){
3        System.out.println(array[i]);
4    }
```

В то же время эта версия цикла for более гибкая по сравнению for (int i : array). В частности, в этой версии мы можем изменять элементы:

```
1    int[] array = new int[] { 1, 2, 3, 4, 5 };
2    for (int i=0; i<array.length;i++){
3        array[i] = array[i] * 2;
4        System.out.println(array[i]);
5    }
```

Цикл do

Цикл do сначала выполняет код цикла, а потом проверяет условие в инструкции while. И пока это условие истинно, цикл повторяется. Например:

```
1    int j = 7;
2    do{
3        System.out.println(j);
4        j--;
5    }
6    while (j > 0);
```

В данном случае код цикла сработает 7 раз, пока j не окажется равным нулю. Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно. Так, мы можем написать:

```
1    int j = -1;
2    do{
3       System.out.println(j);
4       j--;
5    }
6    while (j > 0);
```

Хотя переменная j изначально меньше 0, цикл все равно один раз выполнится.

Цикл while

Цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
1    int j = 6;
2    while (j > 0){
3
4       System.out.println(j);
5       j--;
6    }
```

Операторы continue и break

Оператор break позволяет выйти из цикла в любой его момент, даже если цикл не закончил свою работу:

Например:

```
1    for (int i = 0; i < 10; i++){
2       if (i == 5)
3          break;
4       System.out.println(i);
5    }
```

Когда счетчик станет равным 5, сработает оператор break, и цикл завершится.

Теперь сделаем так, чтобы если число равно 5, цикл не завершался, а просто переходил к следующей итерации. Для этого используем оператор continue:

```
1    for (int i = 0; i < 10; i++){
```

```
2        if (i == 5)
3            continue;
4        System.out.println(i);
5    }
```

В этом случае, когда выполнение цикла дойдет до числа 5, которое не удовлетворяет условию проверки, то программа просто пропустит это число и перейдет к следующему.

2.17 Работа со строками

Строка представляет собой последовательность символов. Для работы со строками в Java определен класс String, который предоставляет ряд методов для манипуляции строками. Физически объект String представляет собой ссылку на область в памяти, в которой размещены символы.

Для создания новой строки мы можем использовать один из конструкторов класса String, либо напрямую присвоить строку в двойных кавычках:

```
1    public static void main(String[] args) {
2
3        String str1 = "Java";
4        String str2 = new String(); // пустая строка
5        String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
6        String str4 = new String(new char[]{'w', 'e', 'l', 'c', 'o', 'm', 'e'}, 3, 4);//3 -начальный индекс, 4 -кол-во символов
7
8        System.out.println(str1); // Java
9        System.out.println(str2); //
10       System.out.println(str3); // hello
11       System.out.println(str4); // come
12   }
```

При работе со строками важно понимать, что объект String является неизменяемым (immutable). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

Поскольку строка рассматривается как набор символов, то мы можем применить метод length() для нахождения длины строки или длины набора символов:

```
String str1 = "Java";
System.out.println(str1.length()); // 4
```

А с помощью метода toCharArray() можно обратно преобразовать строку в массив символов:

```
String str1 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
char[] helloArray = str1.toCharArray();
```

Строка может быть пустой. Для этого ей можно присвоить пустые кавычки или удалить из стоки все символы:

```
String s = "";   // строка не указывает на объект
if(s.length() == 0) System.out.println("String is empty");
```

В этом случае длина строки, возвращаемая методом length(), равна 0.

Класс String имеет специальный метод, который позволяет проверить строку на пустоту - isEmpty(). Если строка пуста, он возвращает true:

```
String s = "";   // строка не указывает на объект
if(s.length() == 0) System.out.println("String is empty");
```

Переменная String может не указывать на какой-либо объект и иметь значение null:

```
String s = null;   // строка не указывает на объект
if(s == null) System.out.println("String is null");
```

Значение null не эквивалентно пустой строке. Например, в следующем случае мы столкнемся с ошибкой выполнения:

```
String s = null;   // строка не указывает на объект
if(s.length()==0) System.out.println("String is empty");    // ! Ошибка
```

Так как переменная не указывает ни на какой объект String, то соответственно мы не можем обращаться к методам объекта String. Чтобы избежать подобных ошибок, можно предварительно проверять строку на null:

```
String s = null;   // строка не указывает на объект
if(s!=null && s.length()==0) System.out.println("String is empty");
```

Основные методы класса String

Основные операции со строками раскрывается через методы класса String, среди которых можно выделить следующие:

concat(): объединяет строки

valueOf(): преобразует объект в строковый вид

join(): соединяет строки с учетом разделителя

compare(): сравнивает две строки

charAt(): возвращает символ строки по индексу

getChars(): возвращает группу символов

equals(): сравнивает строки с учетом регистра

equalsIgnoreCase(): сравнивает строки без учета регистра

regionMatches(): сравнивает подстроки в строках

indexOf(): находит индекс первого вхождения подстроки в строку

lastIndexOf(): находит индекс последнего вхождения подстроки в строку

startsWith(): определяет, начинается ли строка с подстроки

endsWith(): определяет, заканчивается ли строка на определенную подстроку

replace(): заменяет в строке одну подстроку на другую

trim(): удаляет начальные и конечные пробелы

substring(): возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса

toLowerCase(): переводит все символы строки в нижний регистр

toUpperCase(): переводит все символы строки в верхний регистр

Разберем работу этих методов.

Соединение строк

Для соединения строк можно использовать операцию сложения ("+"):

```
1    String str1 = "Java";
2    String str2 = "Hello";
3    String str3 = str1 + " " + str2;
4
5    System.out.println(str3); // Hello Java
```

При этом если в операции сложения строк используется нестроковый объект, например, число, то этот объект преобразуется к строке:

```
1    String str3 = "Год " + 2015;
```

Фактически же при сложении строк с нестроковыми объектами будет вызываться метод valueOf() класса String. Данный метод имеет множество перегрузок и преобразует практически все типы данных к строке. Для преобразования объектов различных классов метод valueOfвызывает метод toString() этих классов.

Другой способ объединения строк представляет метод concat():

```
1    String str1 = "Java";
2    String str2 = "Hello";
3    str2 = str2.concat(str1); // HelloJava
```

Метод concat() принимает строку, с которой надо объединить вызывающую строку, и возвращает соединенную строку.

Еще один метод объединения - метод join() позволяет объединить строки с учетом разделителя. Например, выше две строки сливались в одно слово "HelloJava", но в идеале мы бы хотели, чтобы две подстроки были разделены пробелом. И для этого используем метод join():

```
1    String str1 = "Java";
2    String str2 = "Hello";
3    String str3 = String.join(" ", str2, str1); // Hello Java
```

Метод join является статическим. Первым параметром идет разделитель, которым будут разделяться подстроки в общей строке, а все последующие параметры передают через запятую произвольный набор объединяемых подстрок - в данном случае две строки, хотя их может быть и больше

Извлечение символов и подстрок

Для извлечения символов по индексу в классе String определен метод char charAt(int index). Он принимает индекс, по которому надо получить символов, и возвращает извлеченный символ:

```
1    String str = "Java";
2    char c = str.charAt(2);
3    System.out.println(c); // v
```

Как и в массивах индексация начинается с нуля.

Если надо извлечь сразу группу символов или подстроку, то можно использовать метод getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin). Он принимает следующие параметры:

srcBegin: индекс в строке, с которого начинается извлечение символов

srcEnd: индекс в строке, до которого идет извлечение символов

dst: массив символов, в который будут извлекаться символы

dstBegin: индекс в массиве dst, с которого надо добавлять извлеченные из строки символы

```
1    String str = "Hello world!";
2    int start = 6;
3    int end = 11;
4    char[] dst=new char[end - start];
5    str.getChars(start, end, dst, 0);
6    System.out.println(dst); // world
```

Сравнение строк

Для сравнения строк используются методы equals() (с учетом регистра) и equalsIgnoreCase() (без учета регистра). Оба метода в качестве параметра принимают строку, с которой надо сравнить:

```
1    String str1 = "Hello";
2    String str2 = "hello";
3
4    System.out.println(str1.equals(str2)); // false
5    System.out.println(str1.equalsIgnoreCase(str2)); // true
```

В отличие от сравнения числовых и других данных примитивных типов для строк не применяется знак равенства ==. Вместо него надо использовать метод equals().

Еще один специальный метод regionMatches() сравнивает отдельные подстроки в рамках двух строк. Он имеет следующие формы:

```
1    boolean regionMatches(int toffset, String other, int oofset, int len)
2    boolean regionMatches(boolean ignoreCase, int toffset, String other, int oofset, int len)
```

Метод принимает следующие параметры:

ignoreCase: надо ли игнорировать регистр символов при сравнении. Если значение true, регистр игнорируется

toffset: начальный индекс в вызывающей строке, с которого начнется сравнение

other: строка, с которой сравнивается вызывающая

oofset: начальный индекс в сравниваемой строке, с которого начнется сравнение

len: количество сравниваемых символов в обеих строках

Используем метод:

```
1    String str1 = "Hello world";
2    String str2 = "I work";
3    boolean result = str1.regionMatches(6, str2, 2, 3);
4    System.out.println(result); // true
```

В данном случае метод сравнивает 3 символа с 6-го индекса первой строки ("wor") и 3 символа со 2-го индекса второй строки ("wor"). Так как эти подстроки одинаковы, то возвращается true.

И еще одна пара методов int compareTo(String str) и int compareToIgnoreCase(String str) также позволяют сравнить две строки, но при этом они также позволяют узнать больше ли одна строка, чем другая или нет. Если возвращаемое значение больше 0, то первая строка больше второй, если меньше нуля, то, наоборот, вторая больше первой. Если строки равны, то возвращается 0.

Для определения больше или меньше одна строка, чем другая, используется лексикографический порядок. То есть, например, строка "A" меньше, чем строка "B", так как символ 'A' в алфавите стоит перед символом 'B'. Если первые символы строк равны, то в расчет берутся следующие символы. Например:

```
1    String str1 = "hello";
2    String str2 = "world";
3    String str3 = "hell";
4
5    System.out.println(str1.compareTo(str2)); // -15 - str1 меньше чем strt2
6    System.out.println(str1.compareTo(str3)); // 1 - str1 больше чем str3
```

Поиск в строке

Метод indexOf() находит индекс первого вхождения подстроки в строку, а метод lastIndexOf() - индекс последнего вхождения. Если подстрока не будет найдена, то оба метода возвращают -1:

```
1    String str = "Hello world";
2    int index1 = str.indexOf('l'); // 2
3    int index2 = str.indexOf("wo"); //6
4    int index3 = str.lastIndexOf('l'); //9
```

Метод startsWith() позволяют определить начинается ли строка с определенной подстроки, а метод endsWith() позволяет определить заканчивается строка на определенную подстроку:

```
1    String str = "myfile.exe";
2    boolean start = str.startsWith("my"); //true
3    boolean end = str.endsWith("exe"); //true
```

Замена в строке

Метод replace() позволяет заменить в строке одну последовательность символов на другую:

```
1    String str = "Hello world";
2    String replStr1 = str.replace('l', 'd'); // Heddo wordd
3    String replStr2 = str.replace("Hello", "Bye"); // Bye world
```

Обрезка строки

Метод trim() позволяет удалить начальные и конечные пробелы:

```
1    String str = "  hello world  ";
2    str = str.trim(); // hello world
```

Метод substring() возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса:

```
1    String str = "Hello world";
2    String substr1 = str.substring(6); // world
3    String substr2 = str.substring(3,5); //lo
```

Изменение регистра

Метод toLowerCase() переводит все символы строки в нижний регистр, а метод toUpperCase() - в верхний:

```
1    String str = "Hello World";
2    System.out.println(str.toLowerCase()); // hello world
3    System.out.println(str.toUpperCase()); // HELLO WORLD
```

Split

Метод split() позволяет разбить строку на подстроки по определенному разделителю. Разделитель - какой-нибудь символ или набор символов передается в качестве параметра в метод. Например, разобьем текст на отдельные слова:

```
String text = "FIFA will never regret it";
String[] words = text.split(" ");
for(String word : words){
    System.out.println(word);
}
```

В данном случае строка будет разделяться по пробелу. Консольный вывод:

```
FIFA
will
never
regret
it
```

2.18 Массивы. Одномерные массивы и многомерные массивы.

Массив представляет набор однотипных значений. Объявление массива похоже на объявление обычной переменной, которая хранит одиночное значение, причем есть два способа объявления массива:

```
тип_данных название_массива[];
// либо
тип_данных[] название_массива;
```

Например, определим массив чисел:

```
int nums[];
int[] nums2;
```

После объявления массива мы можем инициализовать его:

```
int nums[];
```

```
2    nums = new int[4];  // массив из 4 чисел
```

Создание массива производится с помощью следующей конструкции: new тип_данных[количество_элементов], где new - ключевое слово, выделяющее память для указанного в скобках количества элементов. Например, nums = new int[4]; - в этом выражении создается массив из четырех элементов int, и каждый элемент будет иметь значение по умолчанию - число 0.

Также можно сразу при объявлении массива инициализировать его:

```
1    int nums[] = new int[4];    // массив из 4 чисел
2    int[] nums2 = new int[5];   // массив из 5 чисел
```

При подобной инициализации все элементы массива имеют значение по умолчанию. Для числовых типов (в том числе для типа char) это число 0, для типа boolean это значение false, а для остальных объектов это значение null. Например, для типа int значением по умолчанию является число 0, поэтому выше определенный массив nums будет состоять из четырех нулей.

Однако также можно задать конкретные значения для элементов массива при его создании:

```
1    // эти два способа равноценны
2    int[] nums = new int[] { 1, 2, 3, 5 };
3
4    int[] nums2 = { 1, 2, 3, 5 };
```

Стоит отметить, что в этом случае в квадратных скобках не указывается размер массива, так как он вычисляется по количеству элементов в фигурных скобках.

После создания массива мы можем обратиться к любому его элементу по индексу, который передается в квадратных скобках после названия переменной массива:

```
1    int[] nums = new int[4];
2    // устанавливаем значения элементов массива
3    nums[0] = 1;
4    nums[1] = 2;
5    nums[2] = 4;
6    nums[3] = 100;
7
8    // получаем значение третьего элемента массива
```

```
9    System.out.println(nums[2]);   // 4
```

Индексация элементов массива начинается с 0, поэтому в данном случае, чтобы обратиться к четвертому элементу в массиве, нам надо использовать выражение nums[3].

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим ошибку.

Длина массива

Важнейшее свойство, которым обладают массивы, является свойство length, возвращающее длину массива, то есть количество его элементов:

```
1    int[] nums = {1, 2, 3, 4, 5};

2    int length = nums.length;   // 5
```

Нередко бывает неизвестным последний индекс, и чтобы получить последний элемент массива, мы можем использовать это свойство:

```
1    int last = nums[nums.length-1];
```

Многомерные массивы

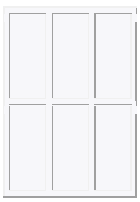
Ранее мы рассматривали одномерные массивы, которые можно представить, как цепочку или строку однотипных значений. Но кроме одномерных массивов также бывают и многомерными. Наиболее известный многомерный массив - таблица, представляющая двухмерный массив:

```
1    int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };

2

3    int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Визуально оба массива можно представить следующим образом:

Одномерный массив nums1

Двухмерный массив nums2



Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Его также можно было создать следующим образом: int[][] nums2 =

new int[2][3];. Количество квадратных скобок указывает на размерность массива. А числа в скобках - на количество строк и столбцов. И также, используя индексы, мы можем использовать элементы массива в программе:

```
// установим элемент первого столбца второй строки
nums2[1][0]=44;
System.out.println(nums2[1][0]);
```

Объявление трехмерного массива могло бы выглядеть так:

```
int[][][] nums3 = new int[2][3][4];
```

Перебор многомерных массивов в цикле

```
int[][] nums = new int[][]
{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
for (int i = 0; i < nums.length; i++){
    for(int j=0; j < nums[i].length; j++){

        System.out.printf("%d ", nums[i][j]);
    }
    System.out.println();
}
```

2.19 Ступенчатые массивы.

Многомерные массивы могут быть также представлены как "зубчатые массивы". В вышеприведенном примере двухмерный массив имел 3 строчки и три столбца, поэтому у нас получалась ровная таблица. Но мы можем каждому элементу в двухмерном массиве присвоить отдельный массив с различным количеством элементов:

```
1	int[][] nums = new int[3][];
2	nums[0] = new int[2];
3	nums[1] = new int[3];
4	nums[2] = new int[5];
```

Пример задачи на ступенчатый массив

С клавиатуры вводятся два числа A и B. С учетом этих чисел сформировать ступенчатый массив, состоящий из 10 массивов, со следующими параметрами: длина массивов с четными номерами равна B, а с нечетными A.

Решение

```
java.util.Scanner in = new java.util.Scanner(System.in);
int A = in.nextInt();
int B = in.nextInt();
int arr[][] = new int[10][];

for(int i=0;i < arr.length; ){
arr[i++] = new int[A];
arr[i++] = new int[B];
}
```

2.20 Инициализация. Поиск и сортировка в массивах классическими методами

Не всегда нужно иметь значения по умолчанию. вы можете инициализировать массив собственными значениями, когда он объявляется, и определить количество элементов. Вслед за объявлением переменной массива добавьте знак равенства, за которым следует список значений элементов, помещенный в фигурные скобки. В этом случае ключевое слово new не используется:

```
int[] cats = {2, 5, 7, 8, 3, 0};  // массив из 6 элементов
```

Можно смешать два способа. Например, если требуется задать явно значения только для некоторых элементов массива, а остальные должные иметь значения по умолчанию.

```
int[] cats = new int[6]; // массив из шести элементов с начальным значением 0 для каждого элемента
cats[3] = 5; // четвертому элементу присвоено значение 5
cats[5] = 7; // шестому элементу присвоено значение 7
```

Массивы часто используют в циклах. Допустим, 5 котов отчитались перед вами о количестве пойманных мышек. Как узнать среднее арифметическое значение:

```
int[] mice = {4, 8, 10, 12, 16};
int result = 0;

for(int i = 0; i < 5; i++){
        result = result + mice[i];
}
result = result / 5;
mInfoTextView.append("Среднее арифметическое: " + result);
```

Массив содержит специальное поле length, которое можно прочитать (но не изменить). Оно позволяет получить количество элементов в массиве. Данное свойство удобно тем, что вы не ошибётесь с размером массива. Последний элемент массива всегда mice[mice.length - 1]. Предыдущий пример можно переписать так:

```
int[] mice = { 4, 8, 10, 12, 16 };
int result = 0;

for (int i = 0; i < mice.length; i++) {
        result = result + mice[i];
}
result = result / mice.length; // общий результат делим на число элементов в массиве
mInfoTextView.append("Среднее арифметическое: " + result);
```

Теперь длина массива вычисляется автоматически, и если вы создадите новый массив из шести котов, то в цикле ничего менять не придётся.

Если вам нужно изменять длину, то вместо массива следует использовать списочный массив ArrayList. Сами массивы неизменяемы.

Будьте осторожны с копированием массивов. Массив - это не числа, а специальный объект, который по особому хранится в памяти. Чтобы не загромождать вас умными словами, лучше покажу на примере.

Допустим, у нас есть одна переменная, затем мы создали вторую переменную и присвоили ей значение первой переменной. А затем проверим их.

```
int a = 5;
int b = a;
mInfoTextView.setText("a = " + a + "\nb = " + b);
```

Получим ожидаемый результат.

```
a = 5
b = 5
```

Попробуем сделать подобное с массивом.

```
int[] anyNumbers = {2, 8, 11};
int[] luckyNumbers = anyNumbers;
luckyNumbers[2] = 25;
mInfoTextView.setText("anyNumbers: " + Arrays.toString(anyNumbers)
    + "\nluckyNumbers: " + Arrays.toString(luckyNumbers));
```

Получим результат.

```
anyNumbers: [2, 8, 25];
luckyNumbers: [2, 8, 25];
```

Мы скопировали первый массив в другую переменную и в ней поменяли третий элемент. А когда стали проверять значения у обоих массивов, то оказалось, что у первого массива тоже поменялось значение. Но мы же его не трогали! Магия. На самом деле нет, просто массив остался прежним и вторая переменная обращается к нему же, а не создаёт вторую копию. Помните об этом.

Если же вам реально нужна копия массива, то используйте метод Arrays.copyOf()

Если ваша программа выйдет за пределы индекса массива, то программа остановится с ошибкой времени исполнения ArrayOutOfBoundsException. Это очень частая ошибка у программистов, проверяйте свой код.

Сортировка в массивах

Сортировка массива используется в реальных проектах чаще, чем Вы это можете представить. Например, файловому менеджеру необходимо отсортировать отображаемые файлы по имени, чтобы пользователь мог легко перемещаться по ним. Или, еще один пример, в видеоигре может понадобиться сортировка 3D-объектов, отображаемых в мире, на основе их расстояния от глаза игрока в виртуальном мире, чтобы определить, что видимо, а что нет. Объекты, которые оказываются ближе всего к игроку видны, а те, которые находятся дальше, могут скрываться.

С полезностью сортировки разобрались. Теперь можно и рассмотреть алгоритмы. И первое, что мы рассмотрим это будет **алгоритм сортировки выбором**. Это один из самых простых алгоритмов сортировки массива. Его простота реализации сказывается на скорости работы такого алгоритма.

Шаги алгоритма:

Находим номер минимального значения в текущем списке;

Производим обмен этого значения со значением первой не отсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции);

Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

```
int[] arr = {4, 4, 9, 2, 3};
  /*
   * По очереди будем просматривать все подмножества элементов массива (0 -
   * последний, 1-последний, 2-последний,...)
   */
  for(int i = 0;i<arr.length;i++) {
     /*
      * Предполагаем, что первый элемент (в каждом подмножестве элементов)
      * является минимальным
      */
     int min = arr[i];
     int min_i = i;
     /*
      * В оставшейся части подмножества ищем элемент, который меньше
```

```
     * предположенного минимума
     */
    for (int j = i + 1; j < arr.length;j++) {
       // Если находим, запоминаем его индекс
       if (arr[j] < min) {
          min = arr[j];
          min_i = j;
       }
    }
    /*
     * Если нашелся элемент, меньший, чем на текущей позиции, меняем их
     * местами
     */
    if (i != min_i) {
       int tmp = arr[i];
       arr[i] = arr[min_i];
       arr[min_i] = tmp;
    }
  }
```

Скорость работы данного алгоритма сильно зависит от исходного массива. Частично упорядочен массив будет отсортирован очень быстро, несмотря на простоту алгоритма.

Следующий алгоритм, который мы рассмотрим будет **алгоритм сортировки пузырьком**. Очень часто на собеседованиях просят написать именно этот алгоритм.

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма).

Вот его код на Java:

```
for (int i = 0; i < a.length - 1; i++)
        for (int j = 0; j < a.length - i -1; j++)
          if (a[j] > a[j + 1]) {
             int temp = a[j];
             a[j] = a[j + 1];
             a[j + 1] = temp;
          }
```

Далее посмотрим на **алгоритм сортировки вставками**, который удобен для сортировки коротких последовательностей. Чтобы представить, как работает алгоритм необходимо вспомнить, игру в карты. А именно: когда мы сортируем карты в руках от большей к меньшей или наоборот. Держа в левой руке уже упорядоченные карты и взяв правой рукой очередную карту, мы вставляем ее в нужное место, сравнивая с имеющимися и идя справа налево.

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.

```
int key;
    for (int i = 1; i < array.length; i++) {
       key = array[i];
       int j = i - 1;
       while (j >= 0 && array[j] < key) {
          array[j + 1] = array[j];
          j = j - 1;
       }
       array[j + 1] = key;
    }
```

Время сортировки вставками зависит от массива. Чем больше сортируемый массив, тем больше может понадобиться времени для его сортировки. Причем здесь, как Вы могли заметить, важен не только размер, но

и входной массив. Если Вы читали предыдущий материал о сложности алгоритма, то можете посчитать, что его сложность составляет O(n)=n^2 что не очень хорошо для алгоритма. Этот алгоритм используется как часть **алгоритма сортировки Шелла**.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d. Выбор d – как последовательность чисел Фибоначчи. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d=1 (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Догадаюсь, что немного непонятно. Попробую привести пример. Пусть дан список A = (32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68) и выполняется его сортировка методом Шелла, а в качестве значений d выбраны 5, 3, 1.

На первом шаге сортируются подмассивы A, составленные из всех элементов A, различающихся на 5 позиций, то есть подмассивы A_{5,1} = (32, 66, 40), A_{5, 2} = (95, 35, 43), A_{5, 3} = (16, 19, 93), A_{5, 4} = (82, 75, 68), A_{5, 5} = (24, 54).

В полученном массиве на втором шаге вновь сортируются подмассивы из отстоящих на 3 позиции элементов. Процесс завершается обычной сортировкой вставками получившегося списка.

Вот пример кода на Java:

```
int h = 1;
    int n = array.length;
    while (h < n / 3)
      h = 3 * h + 1;

    while (h >= 1) {
      for (int i = h; i < array.length;i++) {
        for (int j = i; j >= h &&array[j] > array[j - h]; j -= h) {
          int temp = array[j];
          array[j] = array[j - h];
```

```
            array[j - h] = temp;
          }
        }
        h = h / 3;
      }
```

Все приведенные выше алгоритмы практически не применяются на практике, так как являются очень медленными и часто зависят от входного массива. Если Вам нужно отсортировать массив или список в реальных проектах — лучше использовать готовые решения, которые сэкономят Вам время и часто будут быстрее самописных.

Линейный способ поиска элементов в массиве

Линейный или последовательный поиск является простейшим способом найти элемент в массиве чисел. Поиск происходит поочередным сравнением элементом всего массива.

Ниже представлена Java программа поиска элементов в массиве простым перебором:

```
1  package ua.com.prologistic;
2
3  import java.util.Scanner;
4
5  public class LinearSearchExample {
6    public static void main(String args[]) {
7      int counter, num, item, array[];
8
9      //объект для считывания чисел, введенных пользователем в консоль
10     Scanner input = new Scanner(System.in);
11     System.out.println("Введите размер массива: ");
12     num = input.nextInt();
13
14     // создаем пустой массив, определенного выше размера
15     array = new int[num];
```

```
16
17      // просим пользователя заполнить массив, вводя элементы в консоль
18      System.out.println("Введите " + num + " чисел");
19
20      // цикл по размеру массива - вводим числа в консоль
21      for (counter = 0; counter < num; counter++) {
22        array[counter] = input.nextInt();
23      }
24      System.out.println("Введите число, которое надо найти: ");
25      item = input.nextInt();
26
27      for (counter = 0; counter < num; counter++) {
28        if (array[counter] == item) {
29          System.out.println(item + " является " + (counter+1) + " по счету в массиве");
30          // Число найдено, следовательно прекращаем поиск
31          // вызовом оператора break
32          break;
33        }
34      }
35      if (counter == num) {
36        System.out.println("Число " + item + " не найдено в массиве");
37      }
38    }
39 }
```

Обратите внимание, в приведенной выше программе мы использовали оператор break.

Результат выполнения Java программы поиска элемента в массиве:

```
1  Введите количество элементов массива:
2  5
3  Введите 5 чисел
```

```
4  13
5  5
6  23
7  75
8  2
9  Введите число, которое надо найти:
10 23
11 23 является 3 по счету в массиве
```

Пример неудачного поиска элемента:

```
1 Введите количество элементов массива:
2 2
3 Введите 2 чисел
4 52
5 11
6 Введите число, которое надо найти:
7 12
8 Число 12 не найдено в массиве
```

Бинарный поиск

Бинарный или двоичный поиск является одним из классических алгоритмов поиска элементов в отсортированном списке или массиве чисел. Поиск происходит путем деления элементов массива на половины.

Ниже представлена программа, реализующая алгоритм двоичного поиска. Размер и элементы массива будут вводиться пользователем.

```
1  package ua.com.prologistic;
2
3  import java.util.Arrays;
4  import java.util.Scanner;
5
6  public class ExampleBinarySearch {
7
8      public static void main(String args[]) {
```

```
9       int counter, num, item, array[], first, last;
10
11      //Создаем объект Scanner для считывания чисел, введенных пользователем
12      Scanner input = new Scanner(System.in);
13      System.out.println("Введите количество элементов массива: ");
14      num = input.nextInt();
15
16      // Создаем массив введенного пользователем размера
17      array = new int[num];
18
19      System.out.println("Введите " + num + " чисел");
20
21      //Заполняем массив, вводя элементы в консоль
22      for (counter = 0; counter < num; counter++)
23          array[counter] = input.nextInt();
24
25      // сортируем элементы массива, так как для бинарного поиска
26      // элементы массива должны быть отсортированными
27      Arrays.sort(array);
28
29      System.out.println("Введите элемент для бинарного поиска: ");
30      item = input.nextInt();
31      first = 0;
32      last = num - 1;
33
34      // метод принимает начальный и последний индекс, а также число для поиска
35      binarySearch(array, first, last, item);
36  }
37
38  // бинарный поиск
39  public static void binarySearch(int[] array, int first, int last, int item) {
40      int position;
```

```
41        int comparisonCount = 1;    // для подсчета количества сравнений
42
43        // для начала найдем индекс среднего элемента массива
44        position = (first + last) / 2;
45
46        while ((array[position] != item) && (first <= last)) {
47           comparisonCount++;
48           if (array[position] > item) {  // если число заданного для поиска
49              last = position - 1; // уменьшаем позицию на 1.
50           } else {
51              first = position + 1;    // иначе увеличиваем на 1
52           }
53           position = (first + last) / 2;
54        }
55        if (first <= last) {
56           System.out.println(item + " является " + ++position + " элементом в массиве");
57           System.out.println("Метод бинарного поиска нашел число после " + comparisonCount +
58                 " сравнений");
59        } else {
60           System.out.println("Элемент не найден в массиве. Метод бинарного поиска закончил работу
   после "
61                 + comparisonCount + " сравнений");
62        }
63     }
64
65  }
```

Результат выполнения программы бинарного (двоичного) поиска на Java:

```
1  Введите количество элементов массива:
2  5
3  Введите 5 чисел
4  43
5  16
```

```
6  70
7  61
8  9
9  Введите элемент для бинарного поиска:
10 16
11 16 является 4 элементом в массиве
12 Метод бинарного поиска нашел число после 3 сравнений
```

Результат неудачного выполнения бинарного поиска:

```
1  Введите количество элементов массива:
2  3
3  Введите 3 чисел
4  65
5  9
6  11
7  Введите элемент для бинарного поиска:
8  96
9  96 не найден в массиве
10 Элемент не найден в массиве. Метод бинарного поиска закончил работу после 3 сравнений
```

Вот такая простая программа для демонстрации работы двоичного поиска на Java.

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 4. Структурные элементы класса, методы взаимодействия объектов и организация наследования

Компонентные характеристики в определении класса. Доступ к переменным и методам. Конструкторы. Создание объектов класса и время жизни объекта. Перегрузка и переопределение методов. Наследование класса (наследование членов данных, их сокрытие, унаследованные методы). Создание многоуровневой иерархии. Переопределение методов и их применение. Динамическая диспетчеризация методов. Понятие и использование абстрактных классов

Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова class:

```
class Person{

}
```

В данном случае класс называется Person. После названия класса идут фигурные скобки, между которыми помещается тело класса - то есть его поля и методы.

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы. Например, класс Person, который представляет человека, мог бы иметь следующее определение:

```
class Person{

    String name;        // имя
    int age;            // возраст
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

В классе Person определены два поля: name представляет имя человека, а age - его возраст. И также определен метод displayInfo, который ничего не возвращает и просто выводит эти данные на консоль.

Теперь используем данный класс. Для этого определим следующую программу:

```
public class Program{
```

```
    public static void main(String[] args) {

        Person tom;
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Как правило, классы определяются в разных файлах. В данном случае для простоты мы определяем два класса в одном файле. Стоит отметить, что в этом случае только один класс может иметь модификатор public (в данном случае это класс Program), а сам файл кода должен называться по имени этого класса, то есть в данном случае файл должен называться Program.java.

Класс представляет новый тип, поэтому мы можем определять переменные, которые представляют данный тип. Так, здесь в методе main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение null. По большому счету мы ее пока не можем использовать, поэтому вначале необходимо создать объект класса Person.

Конструкторы

Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.

Выше определенный класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию, который мы можем использовать для создания объекта Person. В частности, создадим один объект:

```
public class Program{

    public static void main(String[] args) {
```

```
        Person tom = new Person(); // создание объекта
        tom.displayInfo();

        // изменяем имя и возраст
        tom.name = "Tom";
        tom.age = 34;
        tom.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom".

В итоге мы увидим на консоли:

```
Name: null          Age: 0
Name: Tom           Age: 34
```

Если необходимо, чтобы при создании объекта производилась какая-то логика, например, чтобы поля класса получали какие-то определенные значения, то можно определить в классе свои конструкторы. Например:

```
public class Program{

    public static void main(String[] args) {
```

```
        Person bob = new Person();          // вызов первого конструктора без параметров
        bob.displayInfo();

        Person tom = new Person("Tom"); // вызов второго конструктора с одним параметром
        tom.displayInfo();

        Person sam = new Person("Sam", 25); // вызов третьего конструктора с двумя параметрами
        sam.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    Person()
    {
        name = "Undefined";
        age = 18;
    }
    Person(String n)
    {
        name = n;
        age = 18;
    }
    Person(String n, int a)
    {
        name = n;
        age = a;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Теперь в классе определено три коструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса.

Консольный вывод программы:

```
Name: Undefined         Age: 18
Name: Tom               Age: 18
Name: Sam               Age: 25
```

Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы. Например:

```
public class Program{

    public static void main(String[] args) {

        Person undef = new Person();
        undef.displayInfo();

        Person tom = new Person("Tom");
        tom.displayInfo();

        Person sam = new Person("Sam", 25);
        sam.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст
    Person()
    {
        this("Undefined", 18);
    }
    Person(String name)
    {
        this(name, 18);
    }
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

В третьем конструкторе параметры называются так же, как и поля класса. И чтобы разграничить поля и параметры, применяется ключевое слово this:

```
this.name = name;
```

Так, в данном случае указываем, что значение параметра name присваивается полю name.

Кроме того, у нас три конструктора, которые выполняют идентичные действия: устанавливают поля name и age. Чтобы избежать повторов, с помощью this можно вызвать один из конструкторов класса и передать для его параметров необходимые значения:

```
Person(String name)
{
    this(name, 18);
}
```

В итоге результат программы будет тот же, что и в предыдущем примере.

Инициализаторы

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Инициализатор выполняется до любого конструктора. То есть в инициализатор мы можем поместить код, общий для всех конструкторов:

```
public class Program{

    public static void main(String[] args) {

        Person undef = new Person();
        undef.displayInfo();

        Person tom = new Person("Tom");
        tom.displayInfo();
    }
}
class Person{

    String name;    // имя
    int age;        // возраст

    /*начало блока инициализатора*/
    {
        name = "Undefined";
        age = 18;
    }
    /*конец блока инициализатора*/
    Person(){

    }
```

```
    Person(String name){

        this.name = name;
    }
    Person(String name, int age){

        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Наследование

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, имеется следующий класс Person, описывающий отдельного человека:

```
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
```

И, возможно, впоследствии мы захотим добавить еще один класс, который описывает сотрудника предприятия - класс Employee. Так как этот класс реализует тот же функционал, что и класс Person, поскольку сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (наследником, подклассом) от класса Person, который, в свою очередь, называется базовым классом, родителем или суперклассом:

```
class Employee extends Person{
    public Employee(String name){
        super(name);    // если базовый класс определяет конструктор
```

```
        //  то производный класс должен его вызвать
    }
}
```

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово extends, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова super. Например, класс Person имеет конструктор, который принимает один параметр. Поэтому в классе Employee в конструкторе нужно вызвать конструктор класса Person. То есть вызов super(name) будет представлять вызов конструктора класса Person.

При вызове конструктора после слова super в скобках идет перечисление передаваемых аргументов. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса. Таким образом, установка имени сотрудника делегируется конструктору базового класса.

Причем даже если производный класс никакой другой работы не производит в конструкторе, как в примере выше, все равно необходимо вызвать конструктор базового класса.

Использование классов:

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
        tom.display();
        Employee sam = new Employee("Sam");
        sam.display();
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){
```

```
        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
    public  Employee(String name){
        super(name);    // если базовый класс определяет конструктор
                        //  то производный класс должен его вызвать
    }
}
```

Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором private. При этом производный класс также может добавлять свои поля и методы:

```
public class Program{

    public static void main(String[] args) {

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
        sam.work();     // Sam works in Microsoft
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
```

```
    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    public void work(){
        System.out.printf("%s works in %s \n", getName(), company);
    }
}
```

В данном случае класс Employee добавляет поле company, которое хранит место работы сотрудника, а также метод work.

Переопределение методов

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса. Например, переопределим в классе Employee метод display:

```
public class Program{

    public static void main(String[] args) {

        Employee sam = new Employee("Sam", "Microsoft");
        sam.display();  // Sam
                        // Works in Microsoft
    }
}
class Person {

    String name;
    public String getName(){ return name; }

    public Person(String name){

        this.name=name;
    }

    public void display(){

        System.out.println("Name: " + name);
    }
}
class Employee extends Person{
```

```
    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){

        System.out.printf("Name: %s \n", getName());
        System.out.printf("Works in %s \n", company);
    }
}
```

Перед переопределяемым методом указывается аннотация @Override. Данная аннотация в принципе необязательна.

При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом класса. Например, если в базовом классе метод имеет модификатор public, то и в производном классе метод должен иметь модификатор public.

Однако в данном случае мы видим, что часть метода display в Employee повторяет действия из метода display базового класса. Поэтому мы можем сократить класс Employee:

```
class Employee extends Person{

    String company;

    public Employee(String name, String company) {

        super(name);
        this.company=company;
    }
    @Override
    public void display(){

        super.display();
        System.out.printf("Works in %s \n", company);
    }
}
```

С помощью ключевого слова super мы также можем обратиться к реализации методов базового класса.

Запрет наследования

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова final. Например:

```
public final class Person {
}
```

Если бы класс Person был бы определен таким образом, то следующий код был бы ошибочным и не сработал, так как мы тем самым запретили наследование:

```
class Employee extends Person{ {
}
```

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод display(), запретим его переопределение:

```
public class Person {

    //.....................

    public final void display(){

        System.out.println("Имя: " + name);
    }
}
```

В этом случае класс Employee не сможет переопределить метод display.

Динамическая диспетчеризация методов

Наследование и возможность переопределения методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса:

```
Person sam = new Employee("Sam", "Oracle");
```

Так как Employee наследуется от Person, то объект Employee является в то же время и объектом Person. Грубо говоря, любой работник предприятия одновременно является человеком.

Однако несмотря на то, что переменная представляет объект Person, виртуальная машина видит, что в реальности она указывает на объект Employee. Поэтому при вызове методов у этого объекта будет вызываться та версия метода, которая определена в классе Employee, а не в Person. Например:

```
public class Program{
```

```
        public static void main(String[] args) {

            Person tom = new Person("Tom");
            tom.display();
            Person sam = new Employee("Sam", "Oracle");
            sam.display();
        }
    }
    class Person {

        String name;

        public String getName() { return name; }

        public Person(String name){

            this.name=name;
        }

        public void display(){

            System.out.printf("Person %s \n", name);
        }
    }

    class Employee extends Person{

        String company;

        public Employee(String name, String company) {

            super(name);
            this.company = company;
        }
        @Override
        public void display(){

            System.out.printf("Employee %s works in %s \n", super.getName(),
company);
        }
    }
```

Консольный вывод данной программы:

```
Person Tom
```

Employee Sam works in Oracle

При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется dynamic method lookup или динамический поиск метода или динамическая диспетчеризация методов.

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 5. Типы исключительных ситуаций и процесс их обработки

Необходимость обработки исключительных ситуаций. Основные принципы обработки исключений. Типы исключений. Обработка исключительных ситуаций. Объекты исключительных ситуаций. Стандартные исключительные ситуации. Определение и порождение собственных исключительных ситуаций

Нередко в процессе выполнения программы могут возникать ошибки, при том необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Так, например, может неожиданно оборваться сетевое подключение при передаче файла. Подобные ситуации называются исключениями.

В языке Java предусмотрены специальные средства для обработки подобных ситуаций. Одним из таких средств является конструкция try...catch...finally. При возникновении исключения в блоке try управление переходит в блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch. Например:

```
int[] numbers = new int[3];
numbers[4]=45;
System.out.println(numbers[4]);
```

Так как у нас массив numbers может содержать только 3 элемента, то при выполнении инструкции numbers[4]=45 консоль отобразит исключение, и выполнение программы будет завершено. Теперь попробуем обработать это исключение:

```
try{
   int[] numbers = new int[3];
```

```
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){

    ex.printStackTrace();
}
System.out.println("Программа завершена");
```

При использовании блока try...catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. Поэтому когда выполнение программы дойдет до строки numbers[4]=45;, программа остановится и перейдет к блоку catch

Выражение catch имеет следующий синтаксис: catch (тип_исключения имя_переменной). В данном случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.

Но так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Обработка же исключения в данном случае сводится к выводу на консоль стека трассировки ошибки с помощью метода printStackTrace(), определенного в классе Exception.

После завершения выполнения блока catch программа продолжает свою работу, выполняя все остальные инструкции после блока catch.

Конструкция try..catch также может иметь блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Блок finally выполняется в любом случае, возникло ли исключение в блоке try или нет:

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){

    ex.printStackTrace();
}
finally{
    System.out.println("Блок finally");
}
System.out.println("Программа завершена");
```

Обработка нескольких исключений

В Java имеется множество различных типов исключений, и мы можем разграничить их обработку, включив дополнительные блоки catch:

```
int[] numbers = new int[3];
try{
   numbers[6]=45;
   numbers[6]=Integer.parseInt("gfd");
}
catch(ArrayIndexOutOfBoundsException ex){

   System.out.println("Выход за пределы массива");
}
catch(NumberFormatException ex){

   System.out.println("Ошибка преобразования из строки в число");
}
```

Если у нас возникает исключение определенного типа, то оно переходит к соответствующему блоку catch.

Оператор throw

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор throw. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод числа, и мы хотим, чтобы, если число больше 30, то возникало исключение:

```
package firstapp;
 import java.util.Scanner;
public class FirstApp {

   public static void main(String[] args) {

      try{
         Scanner in = new Scanner(System.in);
         int x = in.nextInt();
         if(x>=30){
            throw new Exception("Число x должно быть меньше 30");
        }
      }
      catch(Exception ex){

         System.out.println(ex.getMessage());
      }
      System.out.println("Программа завершена");
   }
```

}

Здесь для создания объекта исключения используется конструктор класса Exception, в который передается сообщение об исключении. И если число x окажется больше 29, то будет выброшено исключение и управление перейдет к блоку catch.

В блоке catch мы можем получить сообщение об исключении с помощью метода getMessage().

Базовым классом для всех исключений является класс Throwable. От него уже наследуются два класса: Error и Exception. Все остальные классы являются производными от этих двух классов.

Класс Error описывает внутренние ошибки в исполняющей среде Java. Программист имеет очень ограниченные возможности для обработки подобных ошибок.

Собственно, исключения наследуются от класса Exception. Среди этих исключений следует выделить класс RuntimeException. RuntimeException является базовым классом для так называемой группы непроверяемых исключений (unchecked exceptions) - компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором throws в объявлении метода. Такие исключения являются следствием ошибок разработчика, например, неверное преобразование типов или выход за пределы массива.

Некоторые из классов непроверяемых исключений:

ArithmeticException: исключение, возникающее при делении на ноль

IndexOutOfBoundException: индекс вне границ массива

IllegalArgumentException: использование неверного аргумента при вызове метода

NullPointerException: использование пустой ссылки

NumberFormatException: ошибка преобразования строки в число

Все остальные классы, образованные от класса Exception, называются проверяемыми исключениями (checked exceptions).

Некоторые из классов проверяемых исключений:

CloneNotSupportedException: класс, для объекта которого вызывается клонирование, не реализует интерфейс Cloneable

InterruptedException: поток прерван другим потоком

ClassNotFoundException: невозможно найти класс

Подобные исключения обрабатываются с помощью конструкции try..catch. Либо можно передать обработку методу, который будет вызывать данный метод, указав исключения после оператора throws:

public Person clone() throws CloneNotSupportedException{

```
    Person p = (Person) super.clone();
    return p;
}
```

В итоге получается следующая иерархия исключений:

Иерархия классов исключений в Java

Поскольку все классы исключений наследуются от класса Exception, то все они наследуют ряд его методов, которые позволяют получить информацию о характере исключения. Среди этих методов отметим наиболее важные:

Метод getMessage() возвращает сообщение об исключении

Метод getStackTrace() возвращает массив, содержащий трассировку стека исключения

Метод printStackTrace() отображает трассировку стека

Например:

```
try{
    int x = 6/0;
}
catch(Exception ex){

    ex.printStackTrace();
}
```

Оператор throws

Иногда метод, в котором может генерироваться исключение, сам не обрабатывает это исключение. В этом случае в объявлении метода используется оператор throws, который надо обработать при вызове этого метода. Например, у нас имеется метод вычисления факториала, и нам надо обработать ситуацию, если в метод передается число меньше 1:

```
public static int getFactorial(int num) throws Exception{

    if(num<1) throw new Exception("The number is less than 1");
    int result=1;
    for(int i=1; i<=num;i++){

        result*=i;
    }
    return result;
}
```

С помощью оператора throw по условию выбрасывается исключение. В то же время метод сам это исключение не обрабатывает с помощью try..catch, поэтому в определении метода используется выражение throws Exception.

Теперь при вызове этого метода нам обязательно надо обработать выбрасываемое исключение:

```
public static void main(String[] args){

    try{
        int result = getFactorial(-6);

        System.out.println(result);
    }
    catch(Exception ex){

        System.out.println(ex.getMessage());
    }
}
```

Без обработки исключение у нас возникнет ошибка компиляции, и мы не сможем скомпилировать программу.

В качестве альтернативы мы могли бы и не использовать оператор throws, а обработать исключение прямо в методе:

```
public static int getFactorial(int num){

    int result=1;
    try{
        if(num<1) throw new Exception("The number is less than 1");

        for(int i=1; i<=num;i++){

            result*=i;
        }
    }
    catch(Exception ex){

        System.out.println(ex.getMessage());
        result=num;
    }
    return result;
}
```

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 6. Потоки ввода/вывода и работа с файлами

Чтение консольного ввода. Чтение символов. Чтение строк. Запись консольного вывода. Чтение и запись файлов. Классы и интерфейсы ввода/вывода. Каталоги. Поточные классы. Байтовые потоки. Буферизированные байтовые потоки. Символьные потоки. Использование поточного ввода/вывода. Сериализация. Классы и интерфейсы потоков ввода/вывода. Преимущества потоков.

Для получения данных, введенных пользователем, а также для вывода сообщений нам необходим ряд классов, через которые мы сможем взаимодействовать с консолью. Частично их использование уже рассматривалось в предыдущих темах. Для взаимодействия с консолью нам необходим класс System. Этот класс располагается в пакете java.lang, который автоматически подключается в программу, поэтому нам не надо дополнительно импортировать данный пакет и класс.

Вывод на консоль

Для создания потока вывода в класс System определен объект out. В этом объекте определен метод println, который позволяет вывести на консоль некоторое значение с последующим переводом консоли на следующую строку:

```
System.out.println("Hello world");
```

В метод println передается любое значение, как правило, строка, которое надо вывести на консоль. При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод System.out.print(), который аналогичен println за тем исключением, что не осуществляет перевода на следующую строку.

```
System.out.print("Hello world");
```

Но с помощью метода System.out.print также можно осуществить перевод каретки на следующую строку. Для этого надо использовать escape-последовательность \n:

```
System.out.print("Hello world \n");
```

Если у нас есть два числа, и мы хотим вывести их значения на экран, то мы можем, например, написать так:

```
int x=5;
int y=6;
System.out.println("x="+x +"; y="+y);
```

Но в Java есть также функция для форматированного вывода, унаследованная от языка C: System.out.printf(). С ее помощью мы можем переписать предыдущий пример следующим образом:

```
int x=5;
int y=6;
System.out.printf("x=%d; y=%d \n", x, y);
```

В данном случае символы %d обозначают спецификатор, вместо которого подставляет один из аргументов. Спецификаторов и соответствующих им аргументов может быть множество. В данном случае у нас только два аргумента, поэтому вместо первого %d подставляет значение переменной x, а вместо второго - значение переменной y. Сама буква d означает, что данный спецификатор будет использоваться для вывода целочисленных значений типа int.

Кроме спецификатора %d мы можем использовать еще ряд спецификаторов для других типов данных:

%x: для вывода шестнадцатеричных чисел

%f: для вывода чисел с плавающей точкой

%e: для вывода чисел в экспоненциальной форме, например, 1.3e+01

%c: для вывода одиночного символа

%s: для вывода строковых значений

Например:

```
String name = "Иван";
int age = 30;
float height = 1.7f;
System.out.printf("Имя: %s   Возраст: %d лет   Рост: %.2f метров \n", name, age, height);
```

При выводе чисел с плавающей точкой мы можем указать количество знаков после запятой, для этого используем спецификатор на %.2f, где .2 указывает, что после запятой будет два знака. В итоге мы получим следующий вывод:

```
Имя: Иван   Возраст: 30 лет   Рост: 1,70 метров
```

Консольный ввод

Для получения консольного ввода в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс Scanner, который, в свою очередь использует System.in. Например, создадим маленькую программу, которая осуществляет ввод чисел:

```
import java.util.Scanner;

public class FirstApp {
```

```
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        int[] nums = new int[5];
        for(int i=0;i < nums.length; i++){
            nums[i]=in.nextInt();
        }

        for(int i=0;i < nums.length; i++){
            System.out.print(nums[i]);
        }
        System.out.println();
    }
}
```

Так как класс Scanner находится в пакете java.util, то мы вначале его импортируем. Для создания самого объекта Scanner в его конструктор передается объект System.in. После этого мы можем получать вводимые значения. Например, чтобы получить введенное число, используется метод in.nextInt();, который возвращает введенное с клавиатуры целочисленное значение.

В данном случае в цикле вводятся все элементы массива, а с помощью другого цикла все ранее введенные элементы массива выводятся в строчку.

Класс Scanner имеет еще ряд методов, которые позволяют получить введенные пользователем значения:

next(): считывает введенную строку до первого пробела
nextLine(): считывает всю введенную строку
nextInt(): считывает введенное число int
nextDouble(): считывает введенное число double
hasNext(): проверяет, было ли введено слово
hasNextInt(): проверяет, было ли введено число int
hasNextDouble(): проверяет, было ли введено double

Кроме того, класс Scanner имеет еще ряд методов nextByte/nextShort/nextFloat/nextBoolean, которые по аналогии с nextInt считывают данные определенного типа данных.

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 7. Организация потоков, параллельной обработки, синхронизации и распределенной обработки синхронизуемых участков кода

Общее представление о потоках. Создание, остановка и соединение потоков. Планирование потоков. Управление потоками. Синхронизация.

Синхронизированные методы. Синхронизация блоков операторов. Тупики. Коммуникация между потоками

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.

Класс Thread

В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

С помощью статического метода Thread.currentThread() мы можем получить текущий поток выполнения:

```
public static void main(String[] args) {

    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t.getName()); // main
}
```

По умолчанию именем главного потока будет main.

Для управления потоком класс Thread предоставляет еще ряд методов. Наиболее используемые из них:

getName(): возвращает имя потока

setName(String name): устанавливает имя потока

getPriority(): возвращает приоритет потока

setPriority(int proirity): устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.

isAlive(): возвращает true, если поток активен

isInterrupted(): возвращает true, если поток был прерван

join(): ожидает завершение потока

run(): определяет точку входа в поток
sleep(): приостанавливает поток на заданное количество миллисекунд
start(): запускает поток, вызывая его метод run()

Мы можем вывести всю информацию о потоке:

```
public static void main(String[] args) {

    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t); // main
}
```

Консольный вывод:
Thread[main,5,main]

Первое main будет представлять имя потока (что можно получить через t.getName()), второе значение 5 предоставляет приоритет потока (также можно получить через t.getPriority()), и последнее main представляет имя группы потоков, к которому относится текущий - по умолчанию также main (также можно получить через t.getThreadGroup().getName())

Недостатки при использовании потоков

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

Синхронизация потоков

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми. Например, определим следующий код:

```
public class Program {
```

```
    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

class CommonResource{

    int x=0;
}

class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        res.x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s    %d    \n",    Thread.currentThread().getName(),
res.x);
            res.x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

Здесь определен класс CommonResource, который представляет общий ресурс и в котором определено одно целочисленное поле x.

Этот ресурс используется классом потока CountThread. Этот класс просто увеличивает в цикле значение x на единицу. Причем при входе в поток значение x=1:

```
res.x=1;
```

То есть в итоге мы ожидаем, что после выполнения цикла res.x будет равно 4.

В главном классе программы запускается пять потоков. То есть мы ожидаем, что каждый поток будет увеличивать res.x с 1 до 4 и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным:

```
Thread 1 1
Thread 2 1
Thread 3 1
Thread 5 1
Thread 4 1
Thread 5 6
Thread 2 6
Thread 1 6
Thread 3 6
Thread 4 6
Thread 4 11
Thread 2 11
Thread 5 11
Thread 3 11
Thread 1 11
Thread 4 16
Thread 1 16
Thread 3 16
Thread 5 16
Thread 2 16
```

То есть пока один поток не окончил работу с полем res.x, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова synchronized. Этот оператор предваряет блок кода или метод, который подлежит синхронизации. Для его применения изменим класс CountThread:

```
class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        synchronized(res){
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s  %d  \n",  Thread.currentThread().getName(),
res.x);
                res.x++;
                try{
                    Thread.sleep(100);
```

```
                }
                catch(InterruptedException e){}
            }
        }
    }
}
```

При создании синхронизированного блока кода после оператора synchronized идет объект-заглушка: synchronized(res). Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.

Каждый объект в Java имеет ассоциированный с ним монитор. Монитор представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора synchronized, монитор объекта res блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта res освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

В итоге консольный вывод изменится:

```
Thread 1 1
Thread 1 2
Thread 1 3
Thread 1 4
Thread 3 1
Thread 3 2
Thread 3 3
Thread 3 4
Thread 5 1
Thread 5 2
Thread 5 3
Thread 5 4
Thread 4 1
Thread 4 2
Thread 4 3
Thread 4 4
Thread 2 1
Thread 2 2
Thread 2 3
Thread 2 4
```

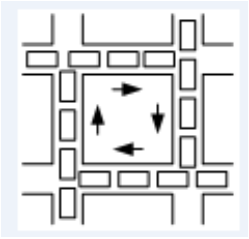
При применении оператора synchronized к методу пока этот метод не завершит выполнение, монопольный доступ имеет только один поток - первый,

который начал его выполнение. Для применения synchronized к методу, изменим классы программы:

```
public class Program {

    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

class CommonResource{

    int x;
    synchronized void increment(){
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), x);
            x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}

class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }

    public void run(){
        res.increment();
    }
}
```

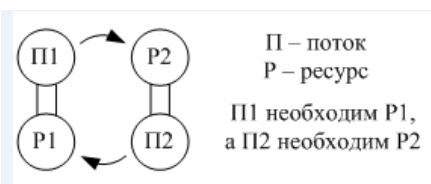
Результат работы в данном случае будет аналогичен примеру выше с блоком synchronized. Здесь опять в дело вступает монитор объекта CommonResource - общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод run() в классе CountThread, а метод increment класса CommonResource. Когда первый поток начинает выполнение метода increment, он захватывает монитор объекта CommonResource. А все потоки также продолжают ожидать его освобождения.

Тупики и методы борьбы с ними. ***Определение.*** Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое только другой процесс данного множества может вызвать. Одновременно в нем может находится как 1, так и несколько процессов. ***Определение 2.*** Процесс или поток находится в состоянии тупика, если он ожидает какого-либо события, которое никогда не произойдет. **Зависание системы (DeadLock)** – ситуация когда один или несколько потоков находятся в состоянии тупика.



Примеры тупиковых ситуаций: 1) транспортная пробка: Потоки – «движение транспорта», ресурсы – «перекрестки, через которые нужно проехать». Пусть организовано одностороннее движение à круговое ожидание. Методы борьбы:

- предотвращение тупика, добавление ресурса (например добавить «мост»);
- обход тупика, то есть проверить условие, можно ли проехать перекресток;
- обнаружение тупика и возобновление потока, для этого нужно убрать один поток (метод наиболее радикальный, используется редко);
- разорвать цепь тупика по timeout.



2) 1 поток взаимодействует с каким-либо устройством. Механизм **timeout**, какое-то другое устройство или поток контролирует время ожидания, при необходимости выводит из тупика. timeout – время тупика не больше установленного значения. В большинстве случаев необходимо делать

функции обработки внутри процесса (программы) в другом потоке. В 1971 г. Хавендер сформулировал следующие четыре необходимых условия для возникновения тупиков. 1. Условие взаимоисключения (монопольности) (Mutual exclusion). Каждый ресурс выделен в точности одному процессу или доступен. Процессы требуют предоставления им монопольного управления ресурсами, которые им выделяются. 2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (которые при этом обычно удерживаются другими процессами). 3. Условие неперераспределяемости (No preemtion). Ресурс, данный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает. 4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся другим процессам цепи. Тупик может возникнуть, если все 4 условия выполняются одновременно. Методы борьбы: 1) предотвращение. 2) обход. 3) обнаружение и восстановление. **Предотвращение тупиков за счет нарушения условий возникновения тупиков (хотя бы одного). Нарушение условия взаимоисключения.** Если в системе отсутствуют выделенные ресурсы, тупиков не будет. Тем не менее, ясно, что обобществление, например, принтера, то есть, разрешение двум процессам писать на один принтер в одно и то же время приведет к хаосу. За счет организации спулинга одновременная печать для нескольких процессов становится возможной. **Нарушение условия ожидания дополнительных ресурсов** Хавендер в 1968 г. предложил следующую стратегию.

- Каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начать выполняться до тех пор, пока все они не будут ему предоставлены.

Если же процесс, удерживает определенные ресурсы и получает отказ в выделении ему дополнительных ресурсов, то он должен освободить свои первоначальные ресурсы и, при необходимости, запросить их снова вместе с дополнительными. **Нарушение принципа неперераспределяемости.** В соответствии со вторым принципом Хавендера можно отбирать ресурсы у удерживающих их процессов до завершения этих процессов. **Нарушение условия кругового ожидания** Циклического ожидания можно избежать несколькими путями. Один из них действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый момент времени. Если нужен второй ресурс - освободи первый. Другой способ - присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке возрастания номеров. Тогда круговое ожидание возникнуть не может. В большинстве ОС предотвращение идет по 4-том пункту.

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 8. Структурные механизмы языка программирования для реализации полиморфизма в программах. Расширение возможностей классов

Использование структурных механизмов (интерфейсов, механизма перегрузки функций, механизма виртуальных функций, механизма перегрузки операций языка) для реализации возможности создания множественных определений для операций и функций. Расширение интерфейсов для создания многоуровневой структурированной иерархии классов

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово interface. Например:

```
interface Printable{

   void print();
}
```

Данный интерфейс называется Printable. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово implements:

```
public class Program{

   public static void main(String[] args) {

      Book b1 = new Book("Java. Complete Referense.", "H. Shildt");
      b1.print();
   }
```

```
}
interface Printable{

    void print();
}
class Book implements Printable{

    String name;
    String author;

    Book(String name, String author){

        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
}
```

В данном случае класс Book реализует интерфейс Printable. При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print. Потом в методе main мы можем создать объект класса Book и вызвать его метод print. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

```
Printable pr = new Printable();
pr.print();
```

Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости. Например, в дополнение к классу Book определим еще один класс, который будет реализовывать интерфейс Printable:

```
class Journal implements Printable {

    private String name;

    String getName(){
```

```
        return name;
    }

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

Класс Book и класс Journal связаны тем, что они реализуют интерфейс Printable. Поэтому мы динамически в программе можем создавать объекты Printable как экземпляры обоих классов:

```
public class Program{

    public static void main(String[] args) {

        Printable printable = new Book("Java. Complete Reference", "H. Shildt");
        printable.print();      //  Java. Complete Reference (H. Shildt)
        printable = new Journal("Foreign Policy");
        printable.print();      // Foreign Policy
    }
}
interface Printable{

    void print();
}
class Book implements Printable{

    String name;
    String author;

    Book(String name, String author){

        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
}
```

```
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Например, так как класс Journal реализует интерфейс Printable, то переменная типа Printable может хранить ссылку на объект типа Journal:

```
p =new Journal("Foreign Affairs");
p.print();
// Интерфейс не имеет метода getName, необходимо явное приведение
String name = ((Journal)p).getName();
System.out.println(name);
```

И если мы хотим обратиться к методам класса Journal, которые определены не в интерфейсе Printable, а в самом классе Journal, то нам надо явным образом выполнить преобразование типов: ((Journal)p).getName();

Методы по умолчанию

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

```
interface Printable {

    default void print(){
```

```
        System.out.println("Undefined printable");
    }
}
```

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом default. Затем в классе Journal нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

```
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }
    Journal(String name){

        this.name = name;
    }
}
```

Статические методы

Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам класса:

```
interface Printable {

    void print();

    static void read(){

        System.out.println("Read printable");
    }
}
```

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

```
public static void main(String[] args) {

    Printable.read();
}
```

Приватные методы

По умолчанию все методы в интерфейсе фактически имеют модификатор public. Однако начиная с Java 9 мы также можем определять в интерфейсе

методы с модификатором private. Они могут быть статическими и нестатическими, но они не могут иметь реализации по умолчанию.

Подобные методы могут использоваться только внутри самого интерфейса, в котором они определены. То есть к примеру, нам надо выполнять в интерфейсе некоторые повторяющиеся действия, и в этом случае такие действия можно выделить в приватные методы:

```
public class Program{

    public static void main(String[] args) {

        Calculatable c = new Calculation();
        System.out.println(c.sum(1, 2));
        System.out.println(c.sum(1, 2, 4));
    }
}
class Calculation implements Calculatable{

}
interface Calculatable{

    default int sum(int a, int b){
        return sumAll(a, b);
    }
    default int sum(int a, int b, int c){
        return sumAll(a, b, c);
    }

    private int sumAll(int... values){
         int result = 0;
         for(int n : values){
            result += n;
         }
         return result;
    }
}
```

Константы в интерфейсах

Кроме методов в интерфейсах могут быть определены статические константы:

```
interface Stateable{

    int OPEN = 1;
```

```
    int CLOSED = 0;

    void printState(int n);
}
```

Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.

Применение констант:

```
public class Program{

    public static void main(String[] args) {

        WaterPipe pipe = new WaterPipe();
        pipe.printState(1);
    }
}
class WaterPipe implements Stateable{

    public void printState(int n){
        if(n==OPEN)
            System.out.println("Water is opened");
        else if(n==CLOSED)
            System.out.println("Water is closed");
        else
            System.out.println("State is invalid");
    }
}
interface Stateable{

    int OPEN = 1;
    int CLOSED = 0;

    void printState(int n);
}
```

Множественная реализация интерфейсов

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова implements:

```
interface Printable {

    // методы интерфейса
}
```

```
interface Searchable {

    // методы интерфейса
}

class Book implements Printable, Searchable{

    // реализация класса
}
```

Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable{

    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Вложенные интерфейсы

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах. Например:

```
class Printer{
    interface Printable {

        void print();
    }
}
```

При применении такого интерфейса нам надо указывать его полное имя вместе с именем класса:

```
public class Journal implements Printer.Printable {

    String name;

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
```

```
}
```

Использование интерфейса будет аналогично предыдущим случаям:

```
Printer.Printable p =new Journal("Foreign Affairs");
p.print();
```

Интерфейсы как параметры и результаты методов

И также как и в случае с классами, интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

```
public class Program{

    public static void main(String[] args) {

        Printable printable = createPrintable("Foreign Affairs",false);
        printable.print();

        read(new Book("Java for impatients", "Cay Horstmann"));
        read(new Journal("Java Dayly News"));
    }

    static void read(Printable p){

        p.print();
    }

    static Printable createPrintable(String name, boolean option){

        if(option)
            return new Book(name, "Undefined");
        else
            return new Journal(name);
    }
}
interface Printable{

    void print();
}
class Book implements Printable{

    String name;
    String author;

    Book(String name, String author){
```

```
        this.name = name;
        this.author = author;
    }

    public void print() {

        System.out.printf("%s (%s) \n", name, author);
    }
}
class Journal implements Printable {

    private String name;

    String getName(){
        return name;
    }

    Journal(String name){

        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

Метод read() в качестве параметра принимает объект интерфейса Printable, поэтому в этот метод мы можем передать как объект Book, так и объект Journal.

Метод createPrintable() возвращает объект Printable, поэтому также мы можем возвратить как объект Book, так и Journal.

Консольный вывод:

```
Foreign Affairs
Java for impatients (Cay Horstmann)
Java Dayly News
```

Перегрузка методов

В программе мы можем использовать методы с одним и тем же именем, но с разными типами и/или количеством параметров. Такой механизм называется перегрузкой методов (method overloading).

Например:

```
public class Program{
```

```
    public static void main(String[] args) {

        System.out.println(sum(2, 3));          // 5
        System.out.println(sum(4.5, 3.2));      // 7.7
        System.out.println(sum(4, 3, 7));       // 14
    }
    static int sum(int x, int y){

        return x + y;
    }
    static double sum(double x, double y){

        return x + y;
    }
    static int sum(int x, int y, int z){

        return x + y + z;
    }
}
```

Здесь определено три варианта или три перегрузки метода sum(), но при его вызове в зависимости от типа и количества передаваемых параметров система выберет именно ту версию, которая наиболее подходит.

Стоит отметить, что на перегрузку методов влияют количество и типы параметров. Однако различие в типе возвращаемого значения для перегрузки не имеют никакого значения. Например, в следующем случае методы различаются по типу возвращаемого значения:

```
public class Program{

    public static void main(String[] args) {

        System.out.println(sum(2, 3));
        System.out.println(sum(4, 3));
    }
    static int sum(int x, int y){

        return x + y;
    }
    static double sum(int x, int y){

        return x + y;
    }
}
```

Однако перегрузкой это не будет считаться. Более того такая программа некорректна и попросту не скомилируется, так как метод с одним и тем же количеством, и типом параметров определен несколько раз.

Литература

Гуськова О.И. Объектно ориентированное программирование Java : учебное пособие / О.И.Гуськова. – Москва : МПГУ, 2018. – 240 с.

Тема 9. Средства языка для организации работы в сети. Основные классы и интерфейсы реализации сетевого взаимодействия

Распределенная обработка данных. Основы работы в сети. Клиент-сервер. Proxy-серверы. Адресация Internet. Сетевые классы и интерфейсы. Обзор сокетов. Зарезервированные сокеты. Сокеты TCP/IP клиентов. Сокеты TCP/IP серверов. Дейтаграммы. Использование URL. Основные классы и интерфейсы реализации сетевого взаимодействия

Что такое прокси-сервер?

Прокси-сервер — это дополнительное звено между вами и интернетом. Некий посредник, который отделяет человека от посещаемого сайта. Создает условия, при которых сайт думает, что прокси — это и есть реальный человек. Только не вы.

Такие посредники довольно многофункциональны и используются в нескольких сценариях:

Для обеспечения конфиденциальности. Чтобы сайты не знали, кто именно их посещает.

Для повышения уровня безопасности при выходе в сеть. Базовые атаки будут направлены именно на прокси.

Еще он нужен, чтобы получать доступ к контенту, который существует только в определенной локации.

Чтобы ускорить доступ к некоторым ресурсам в интернете.

Ну и для того, чтобы получить доступ к заблокированным страницам. Сайтам, мессенджерам и так далее.

Все за счет того, что прокси подменяет IP-адрес, а трафик проходит через дополнительный сервер, на котором могут быть кэшированные данные или организованы дополнительные механизмы защиты данных.

Так как proxy-сервера отвечают за подмену IP, стоит немного пояснить, что он вообще делает и почему замена IP-адреса решает вышеописанные проблемы с доступом к сайтам и сервисам.

IP-адрес говорит сайтам и веб-приложениям, где вы находитесь. Что ставит под угрозу конфиденциальность и безопасность.

Его же используют, чтобы блокировать доступ к контенту. Зачастую на основе физического расположения.

Поэтому люди используют proxy и прячутся за посторонними адресами, чтобы избегать блокировок и не так сильно светиться в интернете. Но опять же

есть исключения, когда proxy-сервер в открытую делится данными о пользователе с сайтом и используется только для ускорения передачи запросов.

Типы прокси-серверов

Косвенно я уже упомянул о том, что proxy бывают разными. Зачастую тип сервера сопоставим с задачами, которые он выполняет. Но для начала мы обсудим именно базовую типизацию proxy, а потом более подробно поговорим о том, какие проблемы эти серверы решают.

Прозрачные

Такой прокси-сервер не утаивает от посещаемого сайта никакой информации. Во-первых, он честно сообщит ему о том, что является прокси, а во-вторых, передаст сайту IP-адрес пользователя по ту сторону сервера. С подобным типом можно встретиться в публичных заведениях, школах.

Анонимные

Более востребованный тип прокси. В отличие от первого, он тоже заявляет посещаемому ресурсу о своей proxy-сущности, но личные данные клиента не передает. То есть будет предоставлять обезличенную информацию для обеих сторон. Правда, неизвестно, как поведет себя сайт, который на 100% знает, что общается с proxy.

Искажающие

Такие прокси тоже идентифицируют себя честно, но вместо реальных пользовательских данных передают подставные. В таком случае сайты подумают, что это вполне себе реальный человек, и будут вести себя соответствующе. Например, предоставлять контент, доступный только в конкретном регионе.

Приватные

Вариант для параноиков. Такие прокси регулярно меняют IP-адреса, постоянно выдают фальшивые данные и заметно сокращают шансы веб-ресурсов отследить трафик и как-то связать его с клиентом.

Другие подкатегории

Прокси-серверы отличаются друг от друга и технически. Существуют:

HTTP-прокси. Самые распространенные. Используются для веб-браузинга. Но они небезопасные, поэтому лучше выбирать другие.

HTTPS. То же самое, что и HTTP, только с шифрованием. Можно смело использовать для выхода на заблокированные сайты типа Pandora или Hulu.

SOCKS. Вариация протокола, работающая с разными типами трафика. Более гибкая и безопасная.

Зачем нужен прокси-сервер?

На плечи proxy возлагают много задач. Сейчас подробно обсудим каждую.

Фильтрация доступных ресурсов

Распространенный сценарий использования в общественных сетях. С помощью такого сервера можно наблюдать за трафиком и при необходимости

его «фильтровать». Это как родительский контроль. Только масштабы иные. Подобный proxy запросто могут поднять в крупной компании, чтобы сотрудники не лезли в Твиттер, пока занимаются делами. Поэтому при входе в соцсеть может вылезти предупреждение с просьбой заняться работой. Ну или вместо этого начальник просто зафиксирует все время пребывания в Фейсбуке, а потом вычтет это из зарплаты. С детьми ситуация примерно такая же. Можно ограничить их свободу в сети на время выполнения домашнего задания, к примеру.

Ускорение работы интернета

На прокси-серверах могут храниться кэшированные копии сайтов. То есть при входе на определенный сайт вы получите данные именно с proxy. С большой долей вероятности, через прокси загрузятся они заметно быстрее. Так происходит, потому что загруженность популярного сайта, на который вы хотите зайти, пострадает меньше, если большое количество людей будет заходить на него через шлюз в виде прокси-сервера.

Сжатие данных

Тоже весьма практичный сценарий. Помогает заметно снизить количество затрачиваемого трафика. На некоторых прокси установлены инструменты, которые сжимают весь запрашиваемый контент перед тем, как перенаправить его к конечному пользователю. По такому принципу работает «Турбо-режим» в браузерах Opera и Яндекса. Сжатие происходит на прокси-сервере, только он загружает полную версию медиа-контента и берет на себя всю нагрузку. А клиент уже скачивает те же данные, только в облегченном виде. Поэтому люди с лимитированным трафиком от этого выигрывают.

Конфиденциальность

Если возникают беспокойства за частную жизнь, то можно настроить приватный или анонимный шлюз, который будет всячески скрывать информацию о компьютере, который сделал первоначальный запрос (уберет его IP-адрес как минимум). Ими пользуются как отдельные личности, уставшие от слежки рекламистов, так и крупные корпорации, не желающие мириться со шпионажем со стороны конкурентов, например. Это, конечно, не панацея, но самые примитивные проблемы, связанные с конфиденциальностью, прокси решить может. А еще он не требует большого количества ресурсов и времени на реализацию.

Безопасность

Прокси может обезопасить не только частную жизнь, но и защитить от реальных угроз вроде вирусов. Можно настроить шлюз таким образом, чтобы он не принимал запросы с вредоносных ресурсов. И превратить получившийся прокси в своего рода массовый «антивирус», через который можно выпускать всех сотрудников компании, не переживая, что те нарвутся на какую-нибудь серьезную угрозу. Конечно, это не защитит пользователей на 100%, но зато даст небольшой прирост безопасности. А он тоже дорогого стоит. Поэтому proxy, используемые именно для защиты, не такая уж редкость.

Доступ к запрещенному контенту

Еще шлюз можно использовать, чтобы обойти региональные запреты. Это работает как с веб-страницами, так и с веб-приложениями. Можно смотреть заграничную библиотеку Netflix, слушать американский музыкальный сервис Pandora, смотреть что-то в Hulu и так далее. Можно заходить на сайты, которые блокируются конкретно в вашей стране. Или случайно заблокированные провайдером. Причем это могут быть совсем безобидные сайты. Я, например, долго не мог зайти на форум sevenstring.com. Ну и всем известная история с Телеграмом, который из недолгого забвения вытащили как раз таки proxy-серверы.

Сравнение прокси с VPN

VPN лучше как в плане безопасности, так и в плане удобства, но такая сеть чаще стоит приличных денег. Зачастую VPN сложнее в настройке и работают не так быстро. Сами посудите, вам обязательно нужен клиент для работы с виртуальными сетями или как минимум разрешения для браузера. Через proxy же можно подключаться, не устанавливая на компьютер ничего.

Риски, которые несет с собой использование прокси

Да, риски есть, причем серьезные. Придется потратить чуть больше времени на изучение proxy-серверов, прежде чем выбрать какой-то из них и начать использовать.

Например, стоит взять во внимание тот факт, что бесплатные прокси зачастую не очень хорошо подходят для решения вопросов безопасности. Чтобы как-то зарабатывать, владельцы шлюзов ищут иные пути для этого. Они продают пользовательские данные. Помогают распространять таргетинговую рекламу. Но даже этих денег не хватает, чтобы обеспечить высокую безопасность и скорость работы сервера, поэтому бесплатные варианты бывают тормозными и небезопасными.

Также стоит понимать: использование прокси-сервера равняется передаче личных данных третьему лицу. Обычно с ними знакомятся только провайдер связи и владельцы страниц, которые вы посещаете. Теперь появится еще одна сторона, у которой будет доступ ко всему вашему трафику. Не факт, что он будет шифроваться или храниться в безопасности. И неизвестно, на каких условиях proxy-сервер может взаимодействовать с государством.

Способы доступа к ресурсам сети из программных приложений.

Java поддерживает протокол TCP/IP, во-первых, расширяя свой интерфейс потоков ввода-вывода, описанного в предыдущей главе, и во вторых, добавляя возможности, необходимые для построения объектов ввода-вывода при работе в сети.

InetAddress

Java поддерживает адреса абонентов, принятые в Internet, с помощью класса InetAddress. Для адресации в Internet используются служебные функции, работающие с обычными, легко запоминающимися символическими именами, эти функции преобразуют символические имена в 32-битные адреса.

Фабричные методы

В классе InetAddress нет доступных пользователю конструкторов. Для создания объектов этого класса нужно воспользоваться одним из его фабричных методов. Фабричные методы — это обычные статические методы, которые возвращают ссылку на объект класса, которому они принадлежат. В данном случае, у класса InetAddress есть три метода, которые можно использовать для создания представителей. Это методы getLocalHost, getByName и getAllByName.

В приведенном ниже примере выводятся адреса и имена локальной машины, локального почтового узла и WWW-узла компании, в которой работает автор.

```
InetAddress Address = InetAddress.getLocalHost();
System.out.println(Address);
Address = InetAddress.getByName("mailhost");
System.out.println(Address);
InetAddress SW[] = InetAddress.getAllByNarne(“www.starwave.com");
System.out.println(SW);
```

У класса InetAddress также есть несколько нестатических методов, которые можно использовать с объектами, возвращаемыми только что названными фабричными методами:

getHostName() возвращает строку, содержащую символическое имя узла, соответствующее хранящемуся в данном объекте адресу Internet.

getAddress() возвращает байтовый массив из четырех элементов, в котором в порядке, используемом в сети, записан адрес Internet, хранящийся в данном объекте.

toString() возвращает строку, в которой записано имя узла и его адрес.

Дейтаграммы

Дейтаграммы, или пакеты протокола UDP (User Datagram Protocol) — это пакеты информации, пересылаемые в сети по принципу “fire-and-forget” (выстрелил и забыл). Если вам надо добиться оптимальной производительности, и вы в состоянии минимизировать затраты на проверку целостности информации, пакеты UDP могут оказаться весьма полезными.

UDP не предусматривает проверок и подтверждений при передаче информации. При передаче пакета UDP по какому-либо адресу нет никакой гарантии того, что он будет принят, и даже того, что по этому адресу вообще есть кому принимать такие пакеты. Аналогично, когда вы получаете дейтаграмму, у вас нет никаких гарантий, что она не была повреждена в пути или что ее отправитель все еще ждет от вас подтверждения ее получения.

Java реализует дейтаграммы на базе протокола UDP, используя для этого два класса. Объекты класса DatagramPacket представляют собой контейнеры с данными, а DatagramSocket — это механизм, используемый при передаче и получении объектов DatagramPacket.

Сокеты “для клиентов”

TCP/IP-сокеты используются для реализации надежных двунаправленных, ориентированных на работу с потоками соединений точка-

точка между узлами Internet. Сокеты можно использовать для соединения системы ввода-вывода Java с программами, которые могут выполняться либо на локальной машине, либо на любом другом узле Internet. В отличие от класса DatagramSocket, объекты класса Socket реализуют высоконадежные устойчивые соединения между клиентом и сервером.

В пакете java.net классы Socket и ServerSocket сильно отличаются друг от друга. Первое отличие в том, что ServerSocket ждет, пока клиент не установит с ним соединение, в то время, как обычный Socket трактует недоступность чего-либо, с чем он хочет соединиться, как ошибку. Одновременно с созданием объекта Socket устанавливается соединение между узлами Internet. Для создания сокетов вы можете использовать два конструктора:

Socket(String host, int port) устанавливает соединение между локальной машиной и указанным портом узла Internet, имя которого было передано конструктору. Этот конструктор может возбуждать исключения UnknownHostException и IOException.

Socket(InetAddress address, int port) выполняет ту же работу, что и первый конструктор, но узел, с которым требуется установить соединение, задается не строкой, а объектом InetAddress. Этот конструктор может возбуждать только IOException.

Из объекта Socket в любое время можно извлечь информацию об адресе Internet и номере порта, с которым он соединен. Для этого служат следующие методы:

getInetAddressQ возвращает объект InetAddress, связанный с данным объектом Socket.

getPort() возвращает номер порта на удаленном узле, с которым установлено соединение.

getLocalPort() возвращает номер локального порта, к которому присоединен данный объект.

После того, как объект Socket создан, им можно воспользоваться для того, чтобы получить доступ к связанным с ним входному и выходному потокам. Эти потоки используются для приема и передачи данных точно так же, как и обычные потоки ввода-вывода, которые мы видели в предыдущей главе:

getInputStream() возвращает InputStream, связанный с данным объектом.

getOutputStream() возвращает OutputStream, связанный с данным объектом.

close() закрывает входной и выходной потоки объекта Socket.

Приведенный ниже очень простой пример открывает соединение с портом 880 сервера “timehost” и выводит полученные от него данные.

```
import java.net.*;
import java.io.*;
class TimeHost {
public static void main(String args[]) throws Exception {
int c;
Socket s = new Socket("timehost.starwave.com",880);
```

```
InputStream in = s.getInputStream();
while ((c = in.read()) != -1) {
System.out.print( (char) c);
}
s.close();} }
```

Сокеты “для серверов”

Как уже упоминалось ранее, Java поддерживает сокеты серверов. Для создания серверов Internet надо использовать объекты класса ServerSocket. Когда вы создаете объект ServerSocket, он регистрирует себя в системе, говоря о том, что он готов обслуживать соединения клиентов. У этого класса есть один дополнительный метод accept(), вызов которого блокирует подпроцесс до тех пор, пока какой-нибудь клиент не установит соединение по соответствующему порту. После того, как соединение установлено, метод accept() возвращает вызвавшему его подпроцессу обычный объект Socket.

Два конструктора класса ServerSocket позволяют задать, по какому порту вы хотите соединяться с клиентами, и (необязательный параметр) как долго вы готовы ждать, пока этот порт не освободится.

ServerSocket(int port) создает сокет сервера для заданного порта.

ServerSocket(int port, int count) создает сокет сервера для заданного порта. Если этот порт занят, метод будет ждать его освобождения максимум count миллисекунд.

URL

URL (Uniform Resource Locators — однородные указатели ресурсов) — являются наиболее фундаментальным компонентом “Всемирной паутины”. Класс URL предоставляет простой и лаконичный программный интерфейс для доступа к информации в Internet с помощью URL.

У класса URL из библиотеки Java - четыре конструктора. В наиболее часто используемой форме конструктора URL адрес ресурса задается в строке, идентичной той, которую вы используете при работе с браузером:

URL(String spec)

Две следующих разновидности конструкторов позволяют задать URL, указав его отдельные компоненты:

URL(String protocol, String host, int port, String file)

URL(String protocol, String host, String file)

Четвертая, и последняя форма конструктора позволяет использовать существующий URL в качестве ссылочного контекста, и создать на основе этого контекста новый URL.

URL(URL context, String spec)

В приведенном ниже примере создается URL, адресующий www-страницу (поставьте туда свой адрес), после чего программа печатает свойства этого объекта.

```
import java.net.URL;
class myURL {
public static void main(String args[]) throws Exception {
URL hp = new URL("http://coop.chuvashia.edu");
```

```
System.out. println("Protocol: " + hp.getProtocol());
System.out.printin("Port: " + hp.getPort());
System.out.println("Host: " + hp.getHost());
System.out.println("File: " + hp.getFile());
System.out.println("Ext: " + hp.toExternaLForm());} }
```

Для того, чтобы извлечь реальную информацию, адресуемую данным URL, необходимо на основе URL создать объект URLConnection, воспользовавшись для этого методом openConnection().

Что такое URL адрес сайта?

Аббревиатура URL по-английски расшифровывается как Uniform Resource Locator, что в переводе на русский язык обозначает «единый указатель ресурсов». По-русски эту аббревиатуру обычно произносят как «у-эр-эл», «ю-ар-эл», или просто «урл». Попробуем разобраться подробнее в том, что такое URL. Каждый документ (веб-страница) в сети Интернет имеет определенное местонахождение, на которое можно точно указать. С помощью URL адреса как раз и указывается точный путь к определенной вебстранице. Аналогично тому, как указывается путь к любому файлу на компьютере, URL адрес строится по определенной схеме, которая обычно выглядит приблизительно так:

http://name.ru/papka/document.html

Где http – указывает на тип протокола, по которому осуществляется передача данных, name.ru – означает доменное имя сайта, papka представляет собой папку, а document.html – конкретную страницу, на которую и ведет данный URL адрес.

Поскольку наш URL адрес http://name.ru/papka/document.html является вымышленным, дается только для примера, и, соответственно, ни к какой вебстранице не ведет, то, попытавшись перейти по нему, мы попадем на страницу, содержащую информацию об ошибке. Выглядеть она может по-разному, однако мы обязательно встретим надпись «404 not found». «Not found» в переводе означает «не найдено», а появление страницы 404 означает, что URL адрес вебстраницы был введен не полностью, неверно (с ошибкой или опечаткой), либо запрошенная страница больше не находится по данному адресу, так как была удалена или переименована.

Ошибка 404 часто возникает при переходе по ссылке, найденной на другой странице, в том случае, если ссылка является устаревшей. Автор сайта мог переместить нужный нам документ, переименовать его или удалить. Что же делать, если при переходе возникает 404 страница? Во-первых, проверить правильность URL адреса, если он нам известен. Исправить ошибки или опечатки и попробовать перейти снова. Если же ошибка 404 возникает при переходе по ссылке на незнакомый ресурс, следует попробовать перейти на главную и воспользоваться поиском по сайту – возможно, что нужная информация все же найдется.

Кстати, многие разработчики сайтов заботятся о том, чтобы страница 404 на их сайте не выглядела устрашающе безнадежно. Здесь размещают

юмористический текст с забавной картинкой, чтобы подбодрить заблудившегося пользователя, а также ссылки на главную сайта, строку поиска либо карту сайта. Если страница 404 выглядит недружелюбно и ссылок для перехода на ней нет, можно попробовать вручную сократить URL адрес, оставив только имя сайта – в нашем примере это будет http://name.ru/ и таким образом попытаться попасть на главную страницу сайта, откуда можно будет перейти на искомую страницу.

URLConnection

URLConnection — объект, который мы используем либо для проверки свойств удаленного ресурса, адресуемого URL, либо для получения его содержимого. В приведенном ниже примере мы создаем URLConnection с помощью метода openConnection, вызванного с объектом URL. После этого мы используем созданный объект для получения содержимого и свойств документа.

```
import java.net.*;
import java.io.*;
class localURL {
public static void main(String args[]) throws Exception {
int c;
URL hp = new URL("http", "127.0.0.1", 80, "/");
URLConnection hpCon = hp.openConnection();
System.out.println("Date: " + hpCon.getDate());
System.out.println("Type: " + hpCon.getContentType());
System.out.println("Exp: " + hpCon.getExpiration());
System.out.println( "Last M: " + hpCon.getLastModified());
System.out.println("Length: " + hpCon.getContentLength());
if (hpCon.getContentLength() > 0) {
System.out.println("=== Content ===");
InputStream input = hpCon.getInputStream();
int i=hpCon.getContentLength();
while (((c = input. read()) != -1) && (--i > 0)) {
System.out.print((char) c);}
input.close();}
else {
System.out.println("No Content Available");
}} }
```

Эта программа устанавливает HTTP-соединение с локальным узлом по порту 80 (у вас на машине должен быть установлен Web-сервер) и запрашивает документ по умолчанию, обычно это - index.html. После этого программа выводит значения заголовка, запрашивает и выводит содержимое документа.

Литература

Гуриков, С.Р. Информатика : учебник / С.Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2021. – 463 с. – (Высшее образование: Бакалавриат)

Тема 10. Библиотеки и средства внедрения визуальных компонентов для организации GUI-интерфейсов пользователя. Обработка событий

Архитектура Модель-Представление-Контроллер (MVC). Создание графического интерфейса при помощи встроенных классов. Компоненты и контейнеры. Использование элементов управления, менеджеров компоновки и меню. Элементы управления. Основные понятия. Добавление и удаление элементов управления. Реагирование на элементы управления. Понятие менеджера компоновки. Работа с меню и диалоговыми окнами. Обработка событий. Модель делегирования событий. Источники событий. Блок прослушивания событий. Классы событий. Элементы-источники событий. Интерфейсы прослушивания событий. Использование модели делегирования событий.

Знакомство с паттерном MVC (Model-View-Controller)

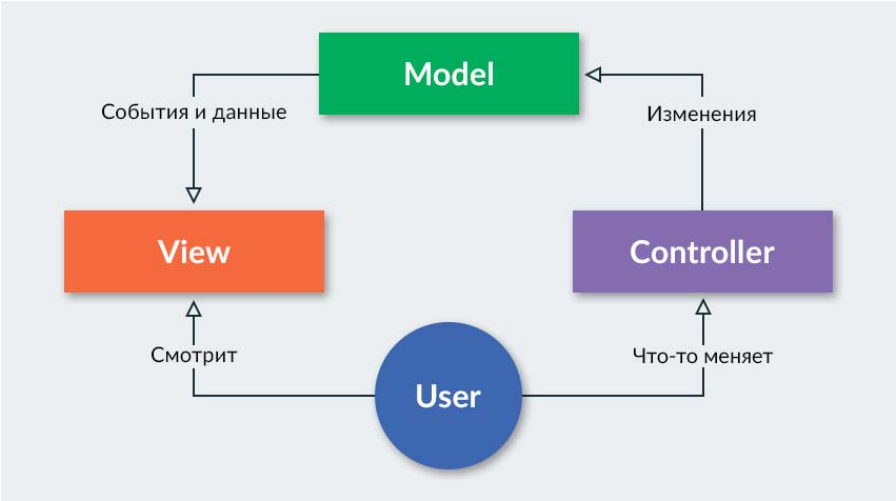
Идеи MVC сформулировал Трюгве Реенскауг (Trygve Reenskaug) во время работы в Xerox PARC в конце 70-х годов. В те времена для работы с ЭВМ было не обойтись без ученой степени и постоянного изучения объемной документации. Задача, которую Реенскауг решал совместно с группой очень сильных разработчиков, заключалась в том, чтобы упростить взаимодействие рядового пользователя с компьютером. Необходимо было создать средства, которые с одной стороны были бы предельно простыми и понятными, а с другой — давали бы возможность управлять компьютером и сложными приложениями. Реенскауг работал в команде, которая занималась разработкой портативного компьютера "для детей всех возрастов" — Dynabook, а также языка SmallTalk под руководством Алана Кея (Alan Kay). Именно тогда и там закладывались понятия дружелюбного интерфейса. Работа Реенскауга совместно с командой во многом повлияла на развитие сферы IT. Приведем интересный факт, который не относится к MVC напрямую, но иллюстрирует значимость тех разработок. В 2007 году после презентации Apple iPhone, Алан Кей сказал: “Когда вышел Macintosh, в Newsweek спросили, что я о нем думаю. Я сказал: это первый персональный компьютер, достойный критики. После презентации Стив Джобс подошел и спросил: достоин ли iPhone критики? И я ответил: сделайте его размером пять на восемь дюймов, и вы завоюете мир”. Спустя 3 года, 27 января 2010 года, Apple представила iPad диагональю 9,7 дюйма. То есть Стив Джобс почти буквально следовал совету Алана Кея. Проект, над которым работал Реннскауг велся на протяжении 10 лет. А первая публикация об MVC от его создателей вышла в свет еще через 10 лет. Мартин Фаулер, автор ряда книг и статей по архитектуре ПО, упоминает, что он изучал MVC по работающей версии SmallTalk. Поскольку информации об MVC из первоисточника долго не было, а также по ряду других причин, появилось большое количество различных трактовок этого понятия. В результате многие считают MVC схемой или паттерном проектирования. Реже MVC называют составным паттерном или комбинацией нескольких паттернов, работающих совместно для реализации сложных приложений. Но на самом деле, как было сказано ранее, MVC — это прежде всего набор архитектурных

идей/принципов/подходов, которые можно реализовать различными способами с использованием различных шаблонов... Далее мы попробуем рассмотреть основные идеи, заложенные в концепции MVC.

Что такое MVC: основные идеи и принципы

- VC — это набор архитектурных идей и принципов для построения сложных информационных систем с пользовательским интерфейсом;
- MVC — это аббревиатура, которая расшифровывается так: Model-View-Controller.

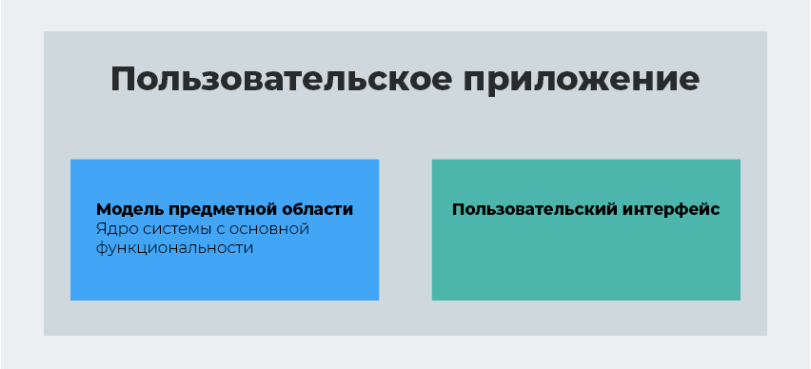
MVC — это именно **набор архитектурных идей и принципов** для построения сложных систем с пользовательским интерфейсом. Но для удобства, чтобы каждый раз не повторять: “Набор архитектурных идей…”, мы будем называть MVC паттерном. Начнем с простого. Что же скрывается за словами Model-View-Controller? При разработке систем с пользовательским интерфейсом, следуя паттерну MVC нужно разделять систему на три составные части. Их, в свою очередь, можно называть модулями или компонентами. Говори как хочешь, но дели на три. У каждой составной компоненты будет свое предназначение. **Model.** Первая компонента/модуль — так называемая модель. Она содержит всю бизнес-логику приложения. **View.** Вторая часть системы — вид. Данный модуль отвечает за отображение данных пользователю. Все, что видит пользователь, генерируется видом. **Controller.** Третьим звеном данной цепи является контроллер. В нем хранится код, который отвечает за обработку действий пользователя (любое действие пользователя в системе обрабатывается в контроллере). Модель — самая независимая часть системы. Настолько независимая, что она не должна ничего знать о модулях Вид и Контроллер. Модель настолько независима, что ее разработчики могут практически ничего не знать о Виде и Контроллере. Основное предназначение Вида — предоставлять информацию из Модели в удобном для восприятия пользователя формате. Основное ограничение Вида — он никак не должен изменять модель. Основное предназначение Контроллера — обрабатывать действия пользователя. Именно через Контроллер пользователь вносит изменения в модель. Точнее в данные, которые хранятся в модели. Приведем еще раз схему, которую тебе уже показывали на лекции:



Из всего этого можно сделать вполне логичный вывод. Сложную систему нужно разбивать на модули. Опишем кратко шаги, как можно добиться подобного разделения.

Шаг 1. Отделить бизнес-логику приложения от пользовательского интерфейса

Ключевая идея MVC состоит в том, что любое приложение с пользовательским интерфейсом в первом приближении можно разбить на 2 модуля: модуль, отвечающий за реализацию бизнес-логики приложения, и пользовательский интерфейс. В первом модуле будет реализован основной функционал приложения. Данный модуль будет ядром системы, в котором реализуется модель предметной области приложения. В концепции MVC данный модуль будет нашей буковкой M, т.е. моделью. Во втором модуле будет реализован весь пользовательский интерфейс, включая отображение данных пользователю и логику взаимодействия пользователя с приложением. Основная цель такого разделения — сделать так, чтобы ядро системы (Модель в терминологии MVC) могла независимо разрабатываться и тестироваться. Архитектура приложения после подобного разделения будет выглядеть следующим образом:



Шаг 2. Используя шаблон Наблюдатель, добиться еще большей независимости модели, а также синхронизации пользовательских интерфейсов

Здесь мы преследуем 2 цели:

1. Добиться еще большей независимости модели.

2. Синхронизировать пользовательские интерфейсы.

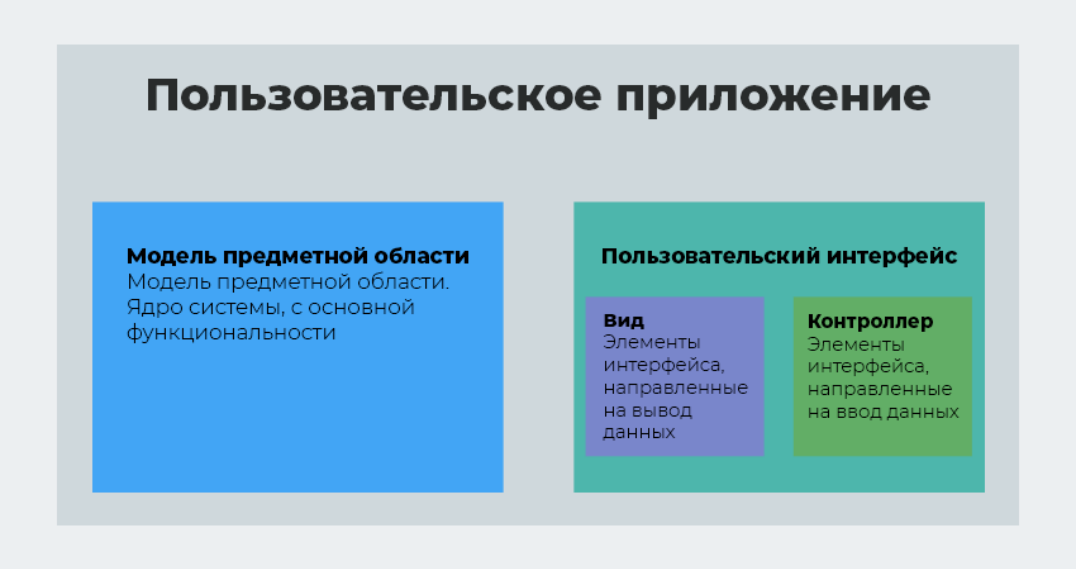
Понять, что подразумевается под синхронизацией пользовательских интерфейсов, поможет следующий пример. Предположим, мы покупаем билет в кино через интернет и видим количество свободных мест в кинотеатре. Одновременно с нами покупать билет в кино может кто-то еще. Если этот кто-то купит билет раньше нас, нам бы хотелось увидеть, что количество свободных мест на наш сеанс уменьшилось. А теперь поразмышляем о том, как это может быть реализовано внутри программы. Предположим, у нас есть ядро системы (наша модель) и интерфейс (веб страница, на которой мы осуществляем покупку). На сайте 2 пользователя одновременно выбирают место. Первый пользователь купил билет. Второму пользователю необходимо отобразить на странице эту информацию. Как это должно произойти? Если мы из ядра системы будем обновлять интерфейс, наше ядро, наша модель, будет зависима от интерфейса. При разработке и тестировании модели придется держать в голове различные способы обновления интерфейса. Чтобы достичь этого, необходимо реализовать шаблон Наблюдатель. С его помощью модель рассылает уведомления об изменениях всем подписчикам. Интерфейс, являясь таким подписчиком, получит уведомление и обновится. Шаблон Наблюдатель позволяет модели с одной стороны информировать интерфейс (вид и контроллер) о том, что в ней произошли изменения, а с другой — фактически ничего о них “не знать”, и тем самым оставаться независимой. С другой стороны, это позволит синхронизировать пользовательские интерфейсы.

Шаг 3. Разделение интерфейса на Вид и Контроллер

Продолжаем делить приложение на модули, но уже на более низком уровне иерархии. На этом шаге пользовательский интерфейс (который был выделен в отдельный модуль на шаге 1) делится на вид и контроллер. Сложно провести строгую черту между видом и контроллером. Если говорить о том, что вид — это то, что видит пользователь, а контроллер — это механизм, благодаря которому пользователь может взаимодействовать с системой, можно обнаружить некоторое противоречие. Элементы управления, например, кнопки на веб-странице или виртуальная клавиатура на экране телефона, это по сути часть контроллера. Но они так же видны пользователю, как и любая часть вида. Здесь скорее речь идет о функциональном разделении. Основная задача пользовательского интерфейса — обеспечить взаимодействие пользователя с системой. Это означает, что у интерфейса всего 2 функции:

- выводить и удобно отображать пользователю информацию о системе;
- вводить данные и команды пользователя в систему (передавать их системе);

Данные функции и определяют то, как нужно делить интерфейс на модули. В итоге, архитектура системы выглядит так:



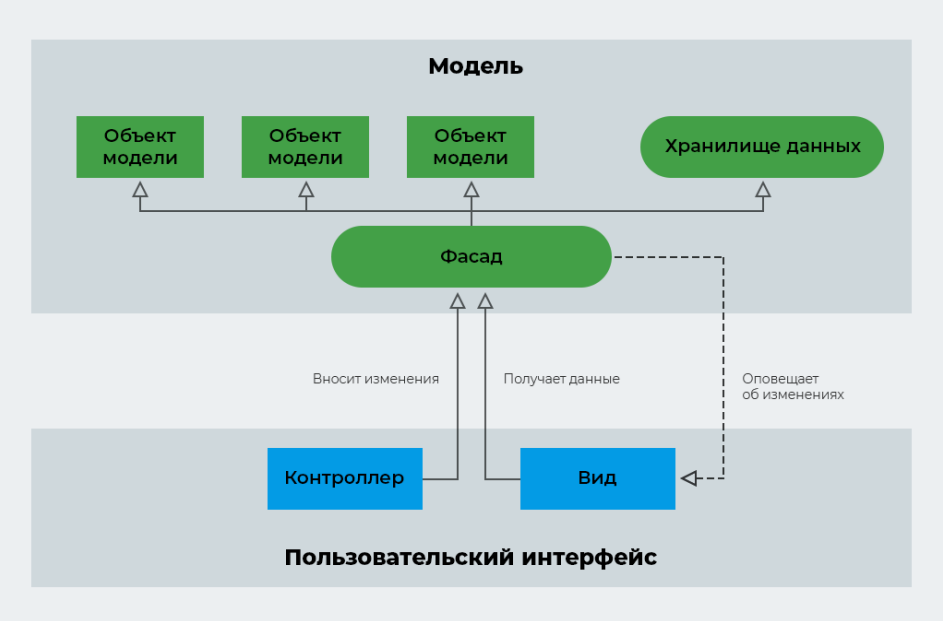
Итак, у нас появилось приложение из трех модулей, которые называются Модель, Вид и Контроллер. Резюмируем:

1. Следуя принципам MVC, систему нужно разделять на модули.
2. Самым важным и независимым модулем должна быть модель.
3. Модель — ядро системы. Нужна возможность разрабатывать и тестировать ее независимо от интерфейса.
4. Для этого на первом шаге сегрегации системы нужно разделить ее на модель и интерфейс.
5. Далее, с помощью шаблона Наблюдатель, укрепляем модель в ее независимости и получаем синхронизацию пользовательских интерфейсов.
6. Третьим шагом делим интерфейс на контроллер и вид.
7. Все, что на ввод информации от пользователя в систему — это в контроллер.
8. Все что на вывод информации от системы к пользователю — это в вид.

Немного о взаимосвязи Вида и Контроллера с Моделью

Когда пользователь вводит информацию через контроллер, он тем самым вносит изменения в модель. По крайней мере, пользователь вносит изменения в данные модели. Когда пользователь получает информацию через элементы интерфейса (через Вид), пользователь получает информацию о данных модели. Как это происходит? Посредством чего Вид и Контроллер взаимодействуют с моделью? Ведь не может быть так, что классы Вида напрямую используют методы классов Модели для чтения/записи данных, иначе ни о какой независимости Модели не может быть и речи. Модель представляет тесно связанный между собой набор классов, к которым, по-хорошему, ни у Вида, ни у Контроллера не должно быть доступа. Для связи Модели с Видом и Контроллером необходимо реализовать шаблон проектирования Фасад. Фасад модели будет той самой прослойкой между Моделью и интерфейсом, через которую Вид получает данные в удобном формате, а Контроллер изменяет

данные, вызывая нужные методы фасада. Схематично, в итоге, все будет выглядеть так:



MVC: в чем профит?

Основная цель следования принципам MVC — отделить реализацию бизнес-логики приложения (модели) от ее визуализации (вида). Такое разделение повысит возможность повторного использования кода. Польза применения MVC наиболее наглядна в случаях, когда пользователю нужно предоставлять одни и те же данные в разных формах. Например, в виде таблицы, графика или диаграммы (используя различные виды). При этом, не затрагивая реализацию видов, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных). Если следовать принципам MVC, можно упростить написание программ, повысить читаемость кода, сделать легче расширение и поддержку системы в будущем.

Литература

1. MV-паттерны для проектирования веб-приложений. Режим доступа: https://bool.dev/blog/detail/mv-patterny-dlya-proektirovaniya-web-prilozheniy.

2. ASP.NET Core MVC. Введение в MVC. Режим доступа: https://metanit.com/sharp/aspnet5/3.1.php.

Тема 11. Проектирование и разработка приложений в архитектуре клиент-сервер с организацией взаимодействия с базой данных

Понятия и терминология ODBC-JDBC. Связь JDBC и ODBC. Настройка базы данных. DriverManager. Создание соединения с источником данных. URL и ODBC-JDBC. Более сложные соединения. Драйвер JDBC-ODBC Bridge. Получение метаданных для множества результатов

Руководство по JDBC. Драйверы.

Драйвер – это сущность, которая реализует определённые интерфейсы JDBC API для взаимодействия с сервером базы данных (далее – БД).

Например, именно драйвер даёт нам возможность открывать соединения и выполнять SQL – запросы и т.д.

Разработчики БД должны имплементировать интерфейс java.sql.Driver для того, чтобы их БД могла работать с JDBC.

Типы драйверов

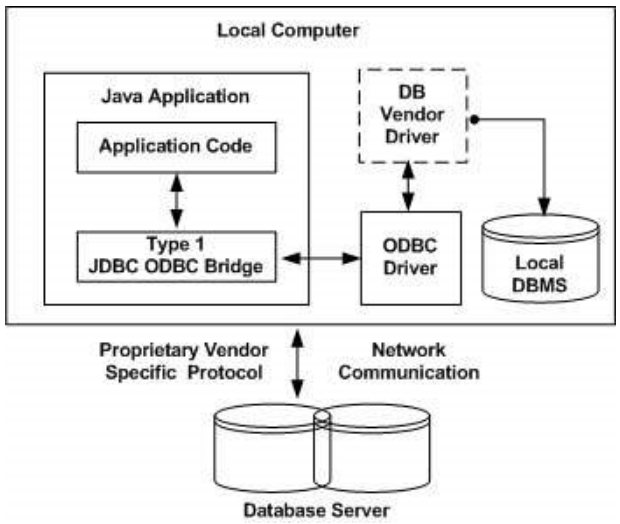
Java работает с различным оборудованием и с различными операционными системами (далее – ОС), именно поэтому существуют различные драйверы реализующие JDBC для различных платформ.

Существует 4 типа драйверов для JDBC.

Рассмотрим их по отдельности:

Тип 1. JDBC – ODBC транслятор

Этот тип драйвера транслирует JDBC в установленный на каждой машине клиентской машине ODBC. Использование ODBC требует конфигурации DSN, который является целевой БД.

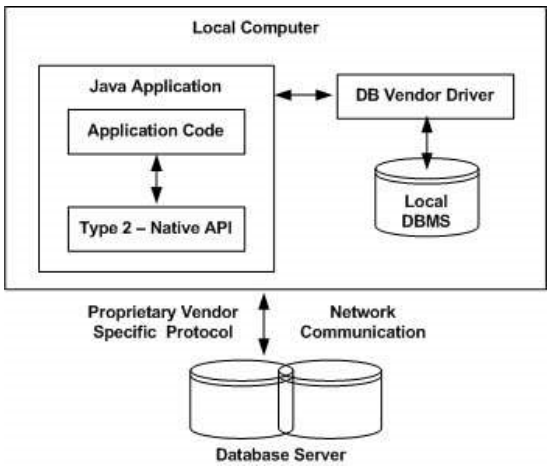


jdbc-driver-type1

Изначально, именно этот тип драйверов был наиболее используемым, так как большинство БД поддерживало только ODBC.

Тип 2. JDBC – нативный API

В этом драйвере JDBC API преобразовывается в уникальный для каждой БД нативный C/C++ API. Его принцип работы крайне схож с драйвером первого типа.

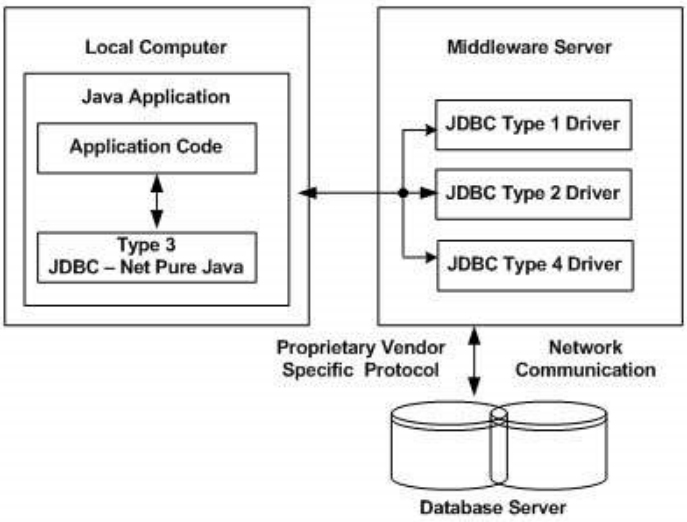


jdbc-driver-type2

Если мы меняем БД, то нам необходимо изменить и нативный API, который будет работать с конкретной БД.

Тип 3. JDBC драйвер на основе библиотеки Java

Этот тип драйверов использует трёхзвенный подход для получения доступа к БД. Для связи с промежуточным сервером приложения используется стандартный сетевой сокет. Информация, полученная от этого сокета транслируется промежуточным сервером в формат, который необходим для конкретной БД и направляется в сервер БД.



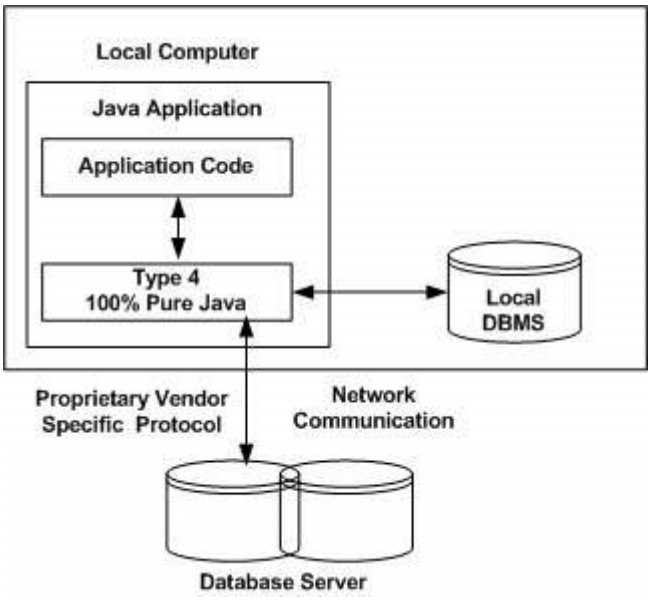
jdbc-driver-type3

Этот подход является крайне гибким, так как нет необходимости устанавливать ПО на стороне клиента и один драйвер способен обеспечить доступ к различным типам БД.

Тип 4. Чистая Java.

Этот тип драйверов разработан полностью с использованием языка программирования Java и работает с БД через сокетное соединение. Главное

его преимущество – наибольшая производительность и, обычно, предоставляется разработчиком БД.



jdbc-driver-type4

Другое его преимущество – невероятная гибкость. Нам не нужно устанавливать никакого дополнительного программного обеспечения (далее – ПО).

Ярким примером такого драйвера является MySQL Connector/J.

Если мы используем такие БД, как MySQL, Oracle и т.д., то наиболее предпочтительным будет использование драйвера типа 4.

Если наше приложение использует различные виды БД, то тип 3 будет более приемлемым.

Если для нашей БД ещё нет драйверов типа 3 или 4, то мы будем вынуждены использовать дврайвер типа 2.

Драйвер типа 1, обычно не используется в коммерческой разработке.

Клиентский ODBC-драйвер Empress был специально модифицирован для работы с мостом JDBC-ODBC. Это предоставляет разработчикам Java-приложений интерфейс к базам данных Empress. Важным компонентом для доступа к базе данных с помощью Java является протокол Java Database Connectivity (JDBC), Java-версия протокола Open Database Connectivity (ODBC), который обеспечивает доступ к клиент/серверным СУБД.

Настоящий документ вкратце описывает, как создавать Java-приложения, имеющие доступ к базам данных Empress. Он базирован и содержит цитаты из документов компании Sun Microsystems, Inc.:

Документация к бета-версии JDK 1.1
Комментарии к релизу JDBC-ODBC версии 1.1001
Об интерфейсе JDBCTM

JDBC является интерфейсом прикладного программирования Java (JavaTM API) для выполнения SQL-выражений. Он состоит из набора классов и интерфейсов, написанных на языке программирования Java, что облегчает отправку SQL-запросов практически к любой реляционной базе данных. Используя JDBC API, можно написать программу, которая сможет отправлять SQL-запросы к нужной базе данных. При этом использование Java устранит необходимость в создании множества различных программ под различные платформы. Применение Java и JDBC позволяет программистам создать программу один раз и затем использовать ее повсюду.

Язык программирования Java, будучи надежным, безопасным, простым в использовании и понимании и автоматически загружаемым через сеть, является превосходной языковой основой для создания приложений для работы с базами данных. Необходимо лишь задать механизм взаимодействия Java-приложений с различными базами данных. Этим механизмом и является интерфейс JDBC.

JDBC расширяет возможности языка программирования Java. Так, использование Java и интерфейса JDBC API позволяет создавать веб-страницы, содержащие апплеты, которые получают информацию из удаленной базы данных. Для предприятия использование JDBC позволит подключить всех его работников к одной или нескольким внутреннии базам данных через Интранет, даже если они используют различные платформы, будь то Windows, Mac OS или UNIX. С увеличением количества Java-разработчиков потребность в легком доступе Java-программ к базам данных возрастает.

Менеджеры информационно-управляющих систем признают преимущества совместного использования Java и JDBC, поскольку благодаря этому значительно упрощается и удешевляется распространение информации. Можно продолжать использовать уже установленные базы данных и леко получать доступ к информации, даже если она хранится в различных СУБД. Значительно уменьшается время разработки новых приложений. Существенно упрощаются инсталляция и последующие обновления продукта. Применение Java и JDBC позволяет один раз написать приложение или обновление для него и, выложив его на сервер, обеспечить доступ к этой последней версии любому пользователю. Для тех, кто занимается предоставлением информационных услуг, совместное применение Java и JDBC открывает широкие возможности для предоставления своим внешним клиентам самой свежей информации.

Что делает JDBC?

Интерфейс JDBC позволяет:
установить связь с базой данных;
отослать SQL-запрос;
обработать результат.

Ниже приведен фрагмент кода, который показывает пример функциональности JDBC:

```
Connection con = DriverManager.getConnection ( "jdbc:odbc:wombat", "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
int x = getInt("a");
String s = getString("b");
float f = getFloat("c");
}
```

В качестве резюме можно сказать, что JDBC API является интерфейсом между Java и основными абстракциями и понятиями SQL. Он построен на основе ODBC, поэтому программисты, знакомые с ODBC, смогут быстро освоить и JDBC. Интерфейс JDBC сохраняет основные характеристики ODBC: оба интерфейса основаны на интерфейсе X/Open уровня запросов SQL (X/Open SQL Call Level Interface). Разница заключается в том, что JDBC, основываясь на Java, позволяет воспользоваться всеми преимуществами этого языка программирования и чрезвычайно прост в использовании.

Для соединения используются следующие компоненты:

Java-приложение, использующее классы JDBC.

Менеджер драйверов JDBC. Он составляет основу архитектуры JDBC и довольно компактен и прост; главная его функция - соединенить Java-приложение с корректным драйвером JDBC.

Мост JDBC-ODBC позволяет использовать драйверы ODBC в качестве драйверов JDBC. Мост JDBC-ODBC является драйвером JDBC, который позволяет выполнять операции JDBC, преобразуя их в операции ODBC. Таким образом, для ODBC-сервера он выглядит как обычный ODBC-клиент. Мост позволяет использовать JDBC с любой базой данных, для которой существует ODBC-драйвер. Он реализован в виде Java-пакета sun.jdbc.odbc и содержит штатную библиотеку, используемую для обеспечения доступа к ODBC. Этот мост является совместной разработкой компаний Intersolv и JavaSoft. Он написан на языке Java и использует для вызова ODBC собственные Java-методы. Единственная разница между версиями для различных платформ заключается в использовании другой штатной библиотеки.

ODBC-драйвер Empress является ODBC-драйвером, обрабатывающим вызовы ODBC и взаимодействующим с сервером Empress Connectivity. Сервер Empress Connectivity может быть запущен на том же компьютере, на котором выполняется приложение. Обычно он запускается на одной из сетевых машин, доступных для клиентской станции.

Литература

Типы драйверов JDBC. Режим доступа: https://java-blog.ru/osnovy/tipy-drayverov.

Тема 12. Основы применения расширенных языков гипертекстовой разметки документов и разработки клиентских и серверных скриптов.

Введение в языки гипертекстовой разметки документов. Основные элементы и структура языков. Понятие и особенности разработки и использования клиентских и серверных скриптов. Запуск и выполнение клиентских и серверных скриптов. Способы организации и хранения данных и механизмы их представления

HTML (HyperText Markup Language) представляет язык разметки гипертекста, используемый преимущественно для создания документов в сети интернет. HTML начал свой путь в начале 90-х годов как примитивный язык для создания веб-страниц, и в настоящий момент уже трудно представить себе интернет без HTML. Подавляющее большинство сайтов так или иначе используют HTML.

В 2014 году официально была завершена работа над новым стандартом - HTML5, который фактически произвел революцию, привнеся в HTML много нового.

Что именно привнес HTML5?

HTML5 определяет новый алгоритм парсинга для создания структуры DOM

добавление новых элементов и тегов, как например, элементы video, audio и ряд других

переопределение правил и семантики уже существовавших элементов HTML

Фактически с добавлением новых функций HTML5 стал не просто новой версией языка разметки для создания веб-страниц, но и фактически платформой для создания приложений, а область его использования вышла далеко за пределы веб-среды интернет: HTML5 применяется также для создания мобильных приложений под Android, iOS, Windows Mobile и даже для создания десктопных приложений для обычных компьютеров (в частности, в ОС Windows 8/8.1/10).

В итоге, как правило, HTML 5 применяется преимущественно в двух значениях:

HTML 5 как обновленный язык разметки гипертекста, некоторое развитие предыдущей версии HTML 4

HTML 5 как мощная платформа для создания веб-приложений, которая включает не только непосредственно язык разметки гипертекста, обновленный HTML, но и язык программирования JavaScript и каскадные таблицы стилей CSS 3.

Кто отвечает за развитие HTML5? Этим занимается World Wide Web Consortium (сокращенно W3C - Консорциум Всемирной Паутины) - независимая международная организация, которая определяет стандарт HTML5 в виде спецификаций. Текущую полную спецификацию на английском языке можно посмотреть по адресу https://www.w3.org/TR/html5/. И надо отметить, что организация продолжает работать над HTML5, выпуская обновления к спецификации.

Поддержка браузерами

Надо отметить, что между спецификацией HTML5 и использованием этой технологии в веб-браузерах всегда был разрыв. Большинство браузеров стало внедрять стандарты HTML5 еще до их официальной публикации. И к текущему моменту большинство последних версий браузеров поддерживают большинство функциональностей HTML5 (Google Chrome, Firefox, Opera, Internet Explorer 11, Microsoft Edge). В то же время многие старые браузеры, как например, Internet Explorer 8 и более младшие версии, не поддерживают стандарты, а IE 9, 10 поддерживает лишь частично.

При этом даже те браузеры, которые в целом поддерживают стандарты, могут не поддерживать какие-то отдельные функции. И это тоже надо учитывать в работе. Но в целом с поддержкой данной технологии довольно хорошая ситуация.

Для проверки поддержки HTML5 конкретным браузером можно использовать специальный сервис http://html5test.com.

Необходимые инструменты

Что потребуется для работы с HTML5? В первую очередь, текстовый редактор, чтобы набирать текст веб-страниц на html. На данный момент одним из самых простых и наиболее популярных текстовых редакторов является Notepad++, который можно найти по адресу http://notepad-plus-plus.org/. К его преимуществам можно отнести бесплатность, подсветка тегов html. В дальнейшем я буду ориентироваться именно на этот текстовый редактор.

Также стоит упомянуть кроссплатформенный текстовый редактор Visual Studio Code. Данный редактор обладает несколько большими возможностями, чем Notepad++, и кроме того, может работать не только в ОС Windows, но и в MacOS и в операционных системах на основе Linux.

И также потребуется веб-браузер для запуска и проверки написанных веб-страничек. В качестве веб-браузера можно взять последнюю версию любого из распространенных браузеров – Google Chrome, Mozilla Firefox, Microsoft Edge, Opera.

Прежде чем переходить непосредственно к созданию своих веб-страниц на HTML5, рассмотрим основные строительные блоки, кирпичики, из которых состоит веб-страница.

Документ HTML5, как и любой документ HTML, состоит из элементов, а элементы состоят из тегов. Как правило, элементы имеют открывающий и закрывающий тег, которые заключаются в угловые скобки. Например:

```
<div>Текст элемента div</div>
```

Здесь определен элемент div, который имеет открывающий тег и закрывающий тег . Между этими тегами находится содержимое элемента div. В данном случае в качестве содержимого выступает простой текст "Текст элемента div".

Элементы также могут состоять из одного тега, например, элемент
, функция которого - перенос строки.

```
<div>Текст <br /> элемента div</div>
```

Такие элементы еще называют пустыми элементами (void elements). Хотя я использовал закрывающий слеш, но его наличие согласно спецификации необязательно, и равнозначно использованию тега без слеша:

Каждый элемент внутри открывающего тега может иметь **атрибуты**. Например:

```
<div style="color:red;">Кнопка</div>
<input type="button" value="Нажать">
```

Здесь определено два элемента: div и input. Элемент div имеет атрибут **style**. После знака равно в кавычках пишется значение атрибута: style="color:red;". В данном случае значение "color:red;" указывает, что цвет текста будет красным.

Второй элемент - элемент input, состоящий из одного тега, имеет два атрибута: type (указывает на тип элемента - кнопка) и value (определяет текст кнопки)

Существуют глобальные или общие для всех элементов атрибуты, как например, style, а есть специфические, применяемые к определенным элементам, как например, type.

Кроме обычных атрибутов существуют еще булевые или логические атрибуты (boolean attributes). Подобные атрибуты могут не иметь значения. Например, у кнопки можно задать атрибут disabled:

```
<input type="button" value="Нажать" disabled>
```

Атрибут disabled указывает, что данный элемент отключен.

Глобальные атрибуты

В HTML5 есть набор **глобальных атрибутов**, которые применимы к любому элементу HTML5:

- **accesskey**: определяет клавишу для быстрого доступа к элементу
- **class**: задает класс CSS, который будет применяться к элементу
- **contenteditable**: определяет, можно ли редактировать содержимое элемента
- **contextmenu**: определяет контекстное меню для элемента, которое отображается при нажатии на элемент правой кнопкой мыши
- **dir**: устанавливает направление текста в элементе
- **draggable**: определяет, можно ли перетаскивать элемент

- **dropzone**: определяет, можно ли копировать переносимые данные при переносе на элемент
- **hidden**: скрывает элемент
- **id**: уникальный идентификатор элемента. На веб-странице элементы не должны иметь повторяющихся идентификаторов
- **lang**: определяет язык элемента
- **spellcheck**: указывает, будет ли для данного элемента использоваться проверка правописания
- **style**: задает стиль элемента
- **tabindex**: определяет порядок, в котором по элементам можно переключаться с помощью клавиши TAB
- **title**: устанавливает дополнительное описание для элемента
- **translate**: определяет, должно ли переводиться содержимое элемента

Но, как правило, из всего этого списка наиболее часто используются три: **class**, **id** и **style**.

Пользовательские атрибуты

В отличие от предыдущей версии языка разметки в HTML5 были добавлены пользовательские атрибуты (custom attributes). Теперь разработчик или создатель веб-страницы сам может определить любой атрибут, предваряя его префиксом *data-*. Например:

```
<input type="button" value="Нажать" data-color="red" >
```

Здесь определен атрибут data-color, который имеет значение "red". Хотя для этого элемента, ни в целом в html не существует подобного атрибута. Мы его определяем сами и устанавливаем у него любое значение.

Одинарные или двойные кавычки

Нередко можно встретить случаи, когда в html при определении значений атрибутов применяются как одинарные, так и двойные кавычки. Например:

```
<input type='button' value='Нажать'>
```

И одинарные, и двойные кавычки в данном случае допустимы, хотя чаще применяются именно двойные кавычки. Однако иногда само значение атрибута может содержать двойные кавычки, и в этом случае все значение лучше поместить в одинарные:

```
<input type="button" value='Кнопка "Привет мир"'>
```

Элементы являются кирпичиками, из которых складывается документ html5. Для создания документа нам надо создать простой текстовый файл, а в качестве расширения файла указать **.html*

Создадим текстовый файл, назовем его *index* и изменим его расширение на **.html**.

Затем откроем этот файл в любом текстовом редакторе, например, в Notepad++. Добавим в файл следующий текст:

```
<!DOCTYPE html>
<html>

</html>
```

Для создания документа HTML5 нам нужны в первую очередь два элемента: DOCTYPE и html. Элемент **doctype** или Document Type Declaration сообщает веб-браузеру тип документа. <!DOCTYPE html> указывает, что данный документ является документом html и что используется html5, а не html4 или какая-то другая версия языка разметки.

А элемент html между своим открывающим и закрывающим тегами содержит все содержимое документа.

Внутри элемента html мы можем разместить два других элемента: **head** и **body**. Элемент head содержит метаданные веб-страницы - заголовок веб-страницы, тип кодировки и т.д., а также ссылки на внешние ресурсы - стили, скрипты, если они используются. Элемент body собственно определяет содержимое html-страницы.

Теперь изменим содержимое файла index.html следующим образом:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Документ HTML5</title>
  </head>
  <body>
    <div>Содержание документа HTML5</div>
  </body>
</html>
```

В элементе head определено два элемента:

- элемент title представляет заголовок страницы
- элемент meta определяет метаинформацию страницы. Для корректного отображения символов предпочтительно указывать кодировку. В данном случае с помощью атрибута charset="utf-8" указываем кодировку utf-8.

В пределах элемента элемента body используется только один элемент - div, который оформляет блок. Содержимым этого блока является простая строка.

Поскольку мы выбрали в качестве кодировки utf-8, то браузер будет отображать веб-страницу именно в этой кодировке. Однако необходимо чтобы сам текст документа также соответствовал выбранной кодировке utf-8. Как правило, в различных текстовых редакторах есть соотвествующие настройки для установки кодировки. Например, в Notepad++ надо зайти в меню **Кодировки** и в открывшемся списке выбрать пункт **Преобразовать в UTF-8 без BOM**:

После этого в статусной строке будет можно будет увидеть **UTF-8 w/o BOM, что будет указывать, что нужная кодировка установлена.**

Сохраним и откроем файл *index.html* в браузере:

Таким образом, мы создали первый документ HTML5. Так как мы указали в элементе title заголовок "Документ HTML5", то именно такое название будет иметь вкладка браузера.

Так как указана кодировка utf-8, то веб-браузер будет корректно отображать кириллические символы.

А весь текст, определенный внутри элемента body мы увидим в основном поле браузера.

При создании документа HTML5 мы можем использовать два различных стиля: HTML и XML.

Стиль HTML предполагает следующие моменты:

Начальные открывающие теги могут отсутствовать у элементов

Конечные закрывающие теги могут отсутствовать у элементов

Только пустые элементы (void elements) (например, br, img, link) могут закрываться с помощью слеша />

Регистр названий тегов и атрибутов не имеет значения

Можно не заключать значения атрибутов в кавычки

Некоторые атрибуты могут не иметь значений (checked и disabled)

Специальные символы не экранируются

Докмент должен иметь элемент DOCTYPE

Это так называемый "разрешительный" стиль, основанный на послаблениях при создании документа.

Документ HTML5 также может быть описан с помощью синтаксиса XML. Такой стиль еще называют "XHTML". Он используется, если заголовок content-type имеет значение application/xml+xhtml. Для данного стиля характерны следующие правила:

Каждый элемент должен иметь начальный открывающий тег

Непустые элементы (non-void elements) с начальным открывающим тегом также должны иметь конечный закрывающий тег

Любой элемент может закрываться с помощью слеша />

Названия тегов и атрибутов регистрозависимы, как правило, используются в нижнем регистре

Значения атрибутов должны быть заключены в кавычки

Атрибуты без значений не допускаются (checked="checked" вместо просто checked)

Специальные символы должны быть экранированы

Сравним два подхода. Подход HTML5:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset=utf-8>
      <title>Заголовок</title>
   </head>
   <body>
      <p>Содержание документа HTML5<br>
```

```
        <input type=button value=Нажать >
    </body>
</html>
```

И аналогичный пример с использованием подхода XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta charset="utf-8">
        <title>Заголовок</title>
    </head>
    <body>
        <p>Содержание документа HTML5<br />
        <input type="button" value="Нажать" /></p>
    </body>
</html>
```

При использовании синтаксиса XHTML нам также надо указать пространство имен для данного документа: <html xmlns="http://www.w3.org/1999/xhtml">

Выбор конкретного стиля при написании html-документов зависит от предпочтений программиста или веб-дизайнера. Нередко используется смешенный стиль, который заимствует правила из первого, и из второго стилей.

В то же время надо учитывать, что наличие у элемента закрывающего и открывающего тегов снижает вероятность, что элемент будет неправильно интерпретирован браузером.

Также заключение значений атрибутов в кавычки поможет избежать потенциальных ошибок. Так, атрибут class может принимать несколько значений подряд. Например: . Но если мы опустим кавычки, то в качестве значения будет использоваться "navmenu", а "bigdesctop" браузер будет пытаться интерпретировать как отдельный атрибут.

Если же возникают затруднения, насколько правильной является создаваемая разметка html, то ее можно проверить с помощью валидатора по адресу https://validator.w3.org:

Литература

Гуриков, С.Р. Информатика : учебник / С.Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2021. – 463 с. – (Высшее образование: Бакалавриат)

Тема 13. Разработка web-приложений с организацией обработки клиентских запросов.

Технологии расширения функциональных возможностей Web серверов. Обработка клиентского запроса: чтение параметров данных формы. Обработка клиентского запроса: заголовки HTTP-запроса. Генерация ответов сервера: коды состояния HTTP. Генерация ответов сервера: заголовки HTTP-ответов.

Обработка cookies. Элементы сценариев web-страниц. Веб-ориентированные способы и модели структурирования и хранения данных, а также механизмы представления этой информации в браузере (клиенту)

Что такое протокол передачи данных

Так называют общепринятое соглашение, благодаря которому разработчики разных сервисов отправляют информацию в едином виде. Например, используя Google Chrome, ты можешь получить информацию и с Facebook, и с Twitter, потому что разработчики передают ее с помощью стандартного протокола HTTP, а твой браузер умеет его обрабатывать. Единые правила очень удобны и самим разработчикам серверных частей: существует очень много библиотек, которые могут за тебя преобразовать информацию и отправить по необходимому протоколу. Изначально HTTP задумывался как протокол передачи HTML-страниц. Долгое время так и было, но сейчас программисты частенько передают по нему и строки, и медиафайлы. В общем, этот протокол универсальный и гибкий, и использовать его действительно просто. А как это делать — сейчас разберемся.

Структура HTTP

Сразу стоит отметить, что HTTP-протокол состоит только из текста. Ну а нас больше всего интересует структура, в которой расположен этот текст. Каждое сообщение состоит из трех частей:

1. Стартовая строка (Starting line) — определяет служебные данные.
2. Заголовки (Headers) — описание параметров сообщения.
3. Тело сообщения (Body) — данные сообщения. Должны отделяться от заголовков пустой строкой.

По HTTP-протоколу можно отправить запрос на сервер (request) и получить ответ от сервера (response). Запросы и ответы немного отличаются параметрами.

Как выглядит простой HTTP-запрос

```
GET / HTTP/1.1
Host: javarush.ru
User-Agent: firefox/5.0 (Linux; Debian 5.0.8; en-US; rv:1.8.1.7)
```

В стартовой строке указаны:

- GET — метод запроса;
- / — путь запроса (path);
- HTTP/1.1 — версия протокола передачи данных.

Затем следуют заголовки:

- Host — хост, которому адресован запрос;
- User-Agent — клиент, который отправляет запрос.

Тело сообщения отсутствует. В HTTP-запросе обязательны только стартовая строка и заголовок Host. Теперь разберем все по порядку. HTTP-

запрос должен содержать какой-то метод. Всего их девять: GET, POST, PUT, OPTIONS, HEAD, PATCH, DELETE, TRACE, CONNECT. Самые распространенные — GET и POST. Этих двух методов на первых порах будет достаточно. GET — запрашивает контент из сервера. Поэтому у запросов с методом GET нет тела сообщения. Но при необходимости можно отправить параметры через path в таком формате: https://cdn.javarush.ru/images/article/155cea79-acfd-4968-9361-ad585e939b82/original.pngsend?name1=value1&name2=value2 Здесь: javarush.ru — хост, /send — путь запроса, ? — разделитель, обозначающий, что дальше следуют параметры запроса. В конце перечисляются параметры в формате ключ=значение, разделенные амперсандом. POST — публикует информацию на сервере. POST-запрос может передавать разную информацию: параметры в формате ключ=значение, JSON, HTML-код или даже файлы. Вся информация передается в теле сообщения. Например:

```
POST /user/create/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 28
Host: javarush.ru

{
 "Id": 12345,
 "User": "John"
}
```

Запрос отправляется по адресу javarush.ru/user/create/json, версия протокола — HTTP/1.1. Accept указывает, какой формат ответа клиент ожидает получить, Content-Type — в каком формате отправляется тело сообщения. Content-Length — количество символов в теле. HTTP-запрос может содержать много разных заголовков. Подробнее с ними можно ознакомиться в спецификации протокола.

HTTP-ответы

После получения запроса, сервер его обрабатывает и отправляет ответ клиенту:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 98

<html>
 <head>
  <title>An Example Page</title>
 </head>
 <body>
```

```
    <p>Hello World</p>
  </body>
</html>
```

Стартовая строка в респонсе содержит версию протокола (HTTP/1.1), Код статуса (200), Описание статуса (OK). В заголовках — тип и длина контента. В теле ответа — HTML-код, который браузер прорисует в HTML-страницу.

Response Status Codes

С телом сообщения и заголовками все ясно, а о кодах статусов стоит сказать пару слов. Response Status Codes всегда трехзначные, и первая цифра кода указывает категорию ответа:

- 1xx — информационный. Запрос получен, сервер готов к продолжению;
- 2xx — успешный. Запрос получен, понятен и обработан;
- 3xx — перенаправление. Следующие действия нужно выполнить для обработки запроса;
- 4xx — ошибка клиента. Запрос содержит ошибки или не отвечает протоколу;
- 5xx — ошибка сервера. Сервер не смог обработать запрос, хотя был составлен верно;

Вторая и третья цифры в коде детализируют ответ. Например:

- 200 OK — реквест получен и успешно обработан;
- 201 Created — реквест получен и успешно обработан, в результате чего создан новый ресурс или его экземпляр;
- 301 Moved Permanently — запрашиваемый ресурс был перемещен навсегда, и последующие запросы к нему должны происходить по новому адресу;
- 307 Temporary Redirect — ресурс перемещен временно. Пока к нему можно обращаться, используя автоматическую переадресацию;
- 403 Forbidden — запрос понятен, но нужна авторизация;
- 404 Not Found — сервер не нашел ресурс по этому адресу;
- 501 Not Implemented — сервер не поддерживает функциональность для ответа на этот запрос;
- 505 HTTP Version Not Supported — сервер не поддерживает указанную версию HTTP-протокола.

Вдобавок к статус-коду ответа также отправляется описание статуса, благодаря которому интуитивно понятно, что значит конкретный статус. HTTP-протокол очень практичен: в нем предусмотрено большое количество хедеров, используя которые можно настроить гибкое общение между клиентом и сервером. Все хедеры запросов и ответов, методы запросов и статус-коды ответов невозможно рассмотреть в одной статье. При необходимости можешь почитать официальную спецификацию протокола, которая описывает все нюансы. HTTP-протокол принято использовать на порту 80, поэтому когда ты видишь адрес, который заканчивается портом 80, можешь быть уверен, что к нему нужно обращаться по HTTP. С развитием технологий и активным

перемещением персональных данных в интернет пришлось задуматься о том, как обеспечить дополнительную защиту информации, которую клиент передает серверу. В результате появился протокол HTTPS.

В чем отличие между HTTPS и HTTP

HTTPS синтаксически идентичен протоколу HTTP, то есть использует те же стартовые строки и заголовки. Единственные отличия — дополнительное шифрование и порт по умолчанию (443). HTTPS шифруется между HTTP и TCP, то есть между прикладным и транспортным уровнями. Если забыл, что это такое, загляни в статью о модели OSI. Современный стандарт шифрования — по протоколу TLS. В эту темы мы слишком углубляться не будем, но запомни, что шифрование происходит перед тем, как информация попадает на транспортный уровень. В HTTPS шифруется абсолютно вся информация, кроме хоста и порта, куда отправлен запрос. Для перевода сервера на использование HTTPS протокола вместо HTTP, нам не нужно менять код сервера. Включение этой фичи происходит в контейнерах сервлетов, о которых мы поговорим в следующих статьях. А на сегодня все. Впрочем, погоди. Чтобы пощупать HTTP-запросы, открой Google Chrome, нажми F12, выбери вкладку Network. Тут будут отображаться все реквесты и респонсы, отправленные/полученные браузером.

Технология cookie

Cookies — это небольшие текстовые файлы, которые хранят персонализированную информацию о пользователе на его компьютере. По данным, которые в них записаны, сайты мгновенно узнают посетителя и загружают настроенные им параметры.

Например, когда покупатель интернет-магазина заходит на сайт, браузер отправляет на жесткий диск его компьютера некую информацию — например, местоположение. Если пользователь создает на сайте личный кабинет, данные его авторизации тоже сохраняются в память устройства. Эти записи — и есть файлы куки.

Куки-файлы на сайте

Cookie-файлы знают, где вы находитесь, на каком говорите языке, какой у вас компьютер и сколько примерно вам лет. Но нет, это не шпионы. Cookie-файл больше похож на библиотечную карточку или запись в журнале посетителей. Например, когда человек впервые приходит в фитнес-клуб, он оставляет о себе данные на ресепшен, и при каждом новом визите ему не нужно регистрироваться заново.

Для чего нужны cookies:
хранят логин и пароль, ускоряют авторизацию
запоминают, какие товары лежат в корзине
сохраняют настройки интерфейса, языка и уведомлений
позволяют предзагружать страницы для ускорения поиска

помогают персонализировать рекламу и рекомендовать интересные материалы

позволяют маркетологам анализировать информацию о пользователях и управлять расходами на рекламу более эффективно

помогают определить ошибки в работе сайта

Седержимое файлов куки — просто текст

В цифровой среде эти крохотные текстовые файлы помогают и пользователям, и маркетологам. С их помощью специалисты анализируют количество посетителей сайта, их действия и источники трафика. Для этих целей куки очень удобны: сбор данных идет автоматически, их структура простая и не требует серьезных манипуляций.

Посетители всегда соглашаются на куки добровольно, поэтому имеют право в любое время отказаться от них. Но выключать эти файлы браузер не рекомендует — из-за этого некоторые веб-страницы могут работать неправильно, а ряд функций будет недоступен.

Cookie в середине 90-х создала команда американских инженеров Netscape Communications. В то время едва зарождалась интернет-торговля, и корпорациям нужны были мощные и безопасные серверы для их сайтов. На этом и специализировалась компания Netscape. Ее создатели предсказали спрос на сетевую безопасность, и компания начала производить защищенные браузеры и веб-серверы.

Когда разработчики Netscape писали программу для корзины покупок, они заметили проблему. Персонализировать покупателей было невозможно: они как будто покупали товары через уличный автомат. Клиенты оставались «безымянными» строчками кода, что сильно мешало бизнесу интернет-ритейлинга.

Это препятствие и подтолкнуло компьютерных инженеров Лу Монтулли и Джона Джанандреа к созданию куки. Новая технология позволила идентифицировать посетителей сайтов и сохранять данные о них, а впоследствии — использовать эту информацию для настройки рекламы.

До внедрения cookie Интернет был, по сути, приватным.

Лоуренс Лессиг, американский общественный активист, писатель, профессор права в Гарвардском университете

Потенциально куки и правда могут навредить — все-таки в них записана персональная информация. С другой стороны, люди и так оставляют Интернете много данных, которые компрометируют их приватность: например, IP или MAC-адрес. Так или иначе, если вы не хотите, чтобы с помощью ваших данных поисковики выдавали рекомендации и персонализировали рекламу, всегда можно заблокировать куки в настройках браузера. Но заходить на часто посещаемые ресурсы будет уже не так удобно и быстро.

Не так давно заговорили о вероятности полной отмены cookies как рекламного инструмента. Браузеры Mozilla и Safari уже блокируют около трети

этих текстовых файлов, — если они исходят от сайта, который пользователь не посещает. Даже Европейский суд запретил устанавливать галочку в графе согласия на куки по умолчанию. Эта мера призвана сделать выбор людей в отношении файлов-печенек более осознанным.

Маркетологи, впрочем, не напуганы, хотя и понимают значимость проблемы. Согласно исследованию Future of Marketing от Econsultancy, половину опрошенных беспокоят последствия отказа от куки. В то же время 72% специалистов уверены, что для cookie существуют «жизнеспособные альтернативы».

Рецепт cookie

Структура файлов-печенек очень простая. Они состоят из текста, в котором записаны данные идентификации, пользовательские настройки, веб-страницы, которые посетил владелец устройства, местоположение или тип компьютера.

Если бы эти текстовые файлы действительно были печеньками, то информация, хранимая в них, была бы тестом и начинкой. Чтобы приготовить печенье, нужно отнести все ингредиенты на кухню, — то есть, перейти на сайт. Потом тесту придают форму, — на этом этапе посетитель придумывает данные аутентификации.

Осталось передать данные на сервер, чтобы тот обработал информацию и вернул ее в виде фрагмента данных. В кулинарной версии этому шагу соответствует выпекание печенек в духовом шкафу, а после выкладка их на красивое блюдо.

Если все печенье не съедят сразу, можно убрать его в буфет — а cookies будут записаны в долговременную память вашего компьютера. Дорогу к буфету забыть сложно, а cookie-файл можно отыскать примерно по такому пути: Windows: C:\Users\Имя_Пользователя\AppData.

Что в начинке разных видов cookie

Вся классификация файлов-печенек укладывается в три основные группы. Каждая группа составляют единство по одному из признаков, а именно — времени, безопасности и цели. Бывает и так, что в одном и том же куки пересекаются задачи разных видов из разных групп.

По параметру времени

Сессионные. Самые недолговечные. Удаляются сразу после того, как пользователь покидает ресурс или заходит на страницу в режиме инкогнито.

Постоянные. Хранятся на компьютере определенное время или до тех пор, пока их не удалит человек.

Вечные, или Evercookie. После удаления из браузера их можно восстановить. Их использование запрещено.

По параметру безопасности

Защищенные. Скажем сразу: чтобы зашифровать куки, нужно владеть навыками программирования. В результате шифрования пользователю

возвращается информация, которую нельзя раскрыть или изменить. Но помните, что у всех методов криптографии есть уязвимости.

Сторонние, или трекеры. Не самые безопасные с точки зрения конфиденциальности данных, но не всегда представляют угрозу. Их записывает ресурс, который пользователь не посещал. Таким ресурсом может быть рекламный баннер сайта-партнера, который собирает информацию о посетителе интернет-магазина. В таком случае сторонний cookie не опасен.

Супер-куки. Это фрагменты данных, подобные обычным куки. Обнаружить их сложно, — они не хранятся в памяти компьютера, — но все же возможно. Хранят информацию о действиях юзера в сети, умеют отслеживать его персонализированные данные даже в приватном режиме и умеют получать доступ к источникам трафика.

По параметру цели

Технические. Необходимы для нормальной работы сайта. Отвечают за поиск ошибок или тестирование новых функций на странице.

Идентификационные. С ними удобнее пользоваться сайтом, так как эти файлы запоминают идентификационные данные, а также выбор языка, региона и валюты.

Аналитические. Созданы для изучения структуры посетителей и их характеристик. Обычно Яндекс.Метрика или Google.Analytics записывают эти текстовые файлы, чтобы предоставить их клиенту — например, интернет-магазину.

Рекламные. Предназначены для настройки таргетированных объявлений. Они помогают определить, с какого ресурса пользователь перешел на рекламное объявление. Могут быть сторонними.

Осталось разобраться, как удалить файлы cookie и зачем это вообще делать.

Во-первых, долгое хранение куки может быть небезопасно для приватности данных. Чем меньше персональной информации хранится на компьютере, тем проще будет жить, если она вдруг утечет в сеть.

Всегда есть риск, что злоумышленники установят на ваш ноутбук вредоносную программу и получат доступ к личной информации или данным банковской карты. В Интернете никто от этого не застрахован, хотя часто значение имеет банальная осмотрительность.

Как удалить cookie

Чистить куки нужно раз в несколько месяцев. Будьте готовы, что эта процедура дизавторизирует вас на некоторых сайтах, но есть легкий способ избежать этого.

Перед тем, как удалить cookies, перейдите в раздел «Пароли» в настройках браузера. Напротив каждого ресурса, на котором вы зарегистрированы, скрыты данные аутентификации. Нажмите на иконку глаза

справа, чтобы увидеть их, перепишите пароли или сохраните их в надежном месте.

Файлы куки — удобный и полезный инструмент, который отлично подходит для настройки рекламы и маркетингового анализа. Простому юзеру он экономит время и килобайты памяти в его мозге — благодаря куки не нужно держать в голове кучу данных аутентификации.

Если вы не доверяете куки, то можете хранить логины и другие данные авторизации в специальной программе или обычном блокноте. Есть и другой вариант — время от времени заботиться о компьютере. Периодически устраивайте генеральную уборку и стряхивайте крошки от печенек с жесткого диска.

Литература

Гуриков, С.Р. Информатика : учебник / С.Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2021. – 463 с. – (Высшее образование: Бакалавриат)

Тема 14. Концепция распределенной обработки данных и технологии удаленной обработки данных.

Понятие и архитектура распределенной системы. Протоколы и программная реализация удаленного вызова процедур. Объектно-ориентированные вызовы удаленных методов. Архитектура решений, основанных на Web Services. Протоколы и стандарты. Публикация и развертывание служб.

Веб-сервисы

Прежде всего, веб-сервисы (или веб-службы) — это технология. И как и любая другая технология, они имеют довольно четко очерченную среду применения.

Если посмотреть на веб-сервисы в разрезе стека сетевых протококолов, мы увидим, что это, в классическом случае, не что иное, как еще одна надстройка поверх протокола HTTP. С другой стороны, если гипотетически разделить Интернет на несколько слоев, мы сможем выделить, как минимум, два концептуальных типа приложений — вычислительные узлы, которые реализуют нетривиальные функции и прикладные веб-ресурсы. При этом вторые, зачастую заинтересованы в услугах первых. Но и сам Интернет — разнороден, т. е. различные приложения на различных узлах сети функционируют на разных аппаратно-программных платформах, и используют различные технологии и языки. Чтобы связать все это и предоставить возможность одним приложениям обмениваться данными с другими, и были придуманы веб-сервисы. По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.

Именно с появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

Протоколы веб-сервисов

На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

SOAP (Simple Object Access Protocol) — на самом деле это тройка стандартов SOAP/WSDL/UDDI

REST (Representational State Transfer)

XML-RPC (XML Remote Procedure Call)

На самом деле, SOAP произошел от XML-RPC и является следующей ступенью его развития. В то время как REST — это концепция, в основе которой лежит скорее архитектурный стиль, нежели новая технология, основанный на теории манипуляции объектами CRUD (Create Read Update Delete) в контексте концепций WWW.

Безусловно, существуют и иные протоколы, но, поскольку они не получили широкого распространения, мы остановимся в этом кратком обзоре на двух основных — SOAP и REST. XML-RPC ввиду того, что является несколько «устаревшим», мы рассматривать подробно не будем [1].

Классический подход (SOAP)

Веб-сервис идентифицируется строкой URI. Веб-сервис имеет программный интерфейс, представленный в машинно-обрабатываемом формате WSDL. Другие системы взаимодействуют с этим веб-сервисом путем обмена сообщениями протокола SOAP. В качестве транспорта для сообщений используется протокол HTTP. Описание веб-сервисов и их API могут быть найдены средствами UDDI. Концептуальная схема технологии приведена на рис. 1., а связь между протоколами — на рис. 2.

Рисунок 1 — Концепция веб-сервиса

SOAP (Simple Object Access Protocol) — протокол обмена сообщениями между потребителем и поставщиком веб-сервиса;

WSDL (Web Services Description Language) — язык описания внешних интерфейсов веб-службы;

UDDI (Universal Discovery, Description and Integration) — универсальный интерфейс распознавания, описания и интеграции, используемый для формирования каталога веб-сервисов и доступа к нему.

Рисунок 2 — Протоколы веб-сервисов

Все спецификации, используемые в технологии, основаны на XML и, соответственно, наследуют его преимущества (структурированность, гибкость и т.д.) и недостатки (громоздкость, медлительность).

SOAP

SOAP (изначально Simple Object Access Protocol, а в версии 1.2 официальная расшифровка аббревиатуры отсутствует) — простой протокол доступа к объектам (компонентам распределенной вычислительной системы), основанный на обмене структурированными сообщениями. Как любой текстовый протокол, SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTPS и др., но чаще всего SOAP используется поверх HTTP.

Все сообщения SOAP оформляются в виде структуры, называемой конвертом (envelop), включающей следующие элементы:

Идентификатор сообщения (локальное имя).
Опциональный элемент Header (заголовок):
Ноль или более ссылок на используемые пространства имен;
Ноль или более свойств, доступных в этом пространстве имен.
Обязательный элемент Body (тело сообщения)
Ноль или более ссылок на используемые пространства имен;
Дочерние элементы тела сообщения
Пример сообщения SOAP:

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

<env:Header>

<n:alertcontrol xmlns:n="http://example.org/alertcontrol">

```
    <n:priority>1</n:priority>
    <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Get up at 6:30 AM</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

WSDL

Язык описания веб-сервисов (Web services Description Language, WSDL) предназначен для унифицированного представления внешних интерфейсов веб-службы. Текущая версия протокола WSDL 2.0 и она имеет некоторые отличия от предыдущих версий.

Таблица 1. Основные элементы протокола WSDL.

Элемент WSDL 1.1	Элемент WSDL 2.0	Краткое описание
PortType	Interface	Представляет описание интерфейса веб-сервиса (список операций и их параметров).
Service	Service	Список системных функций
Binding	Binding	Специфицирует интерфейсы и задает параметры связывания с протоколом SOAP: стиль связывания (RPC/Document) и транспорт (SOAP). Эта секция доступна и для каждой из операций
Operation	Operation	Определяет операцию, представляемую веб-сервером. WSDL-операция — это аналог традиционным функциям и процедурам.
Message	не использ.	Сообщение, связанное с определенной операцией. Содержит информацию, необходимую для выполнения данной операции. Каждое сообщение может состоять из нескольких логических частей, описывающих типы данных и имена атрибутов. В версии 2.0 было исключено, т.к. была внедрена поддержка XML Schema для всех элементов.

Элемент WSDL 1.1	Элемент WSDL 2.0	Краткое описание
Types	Types	Описание данных в соответствии с XML Schema.

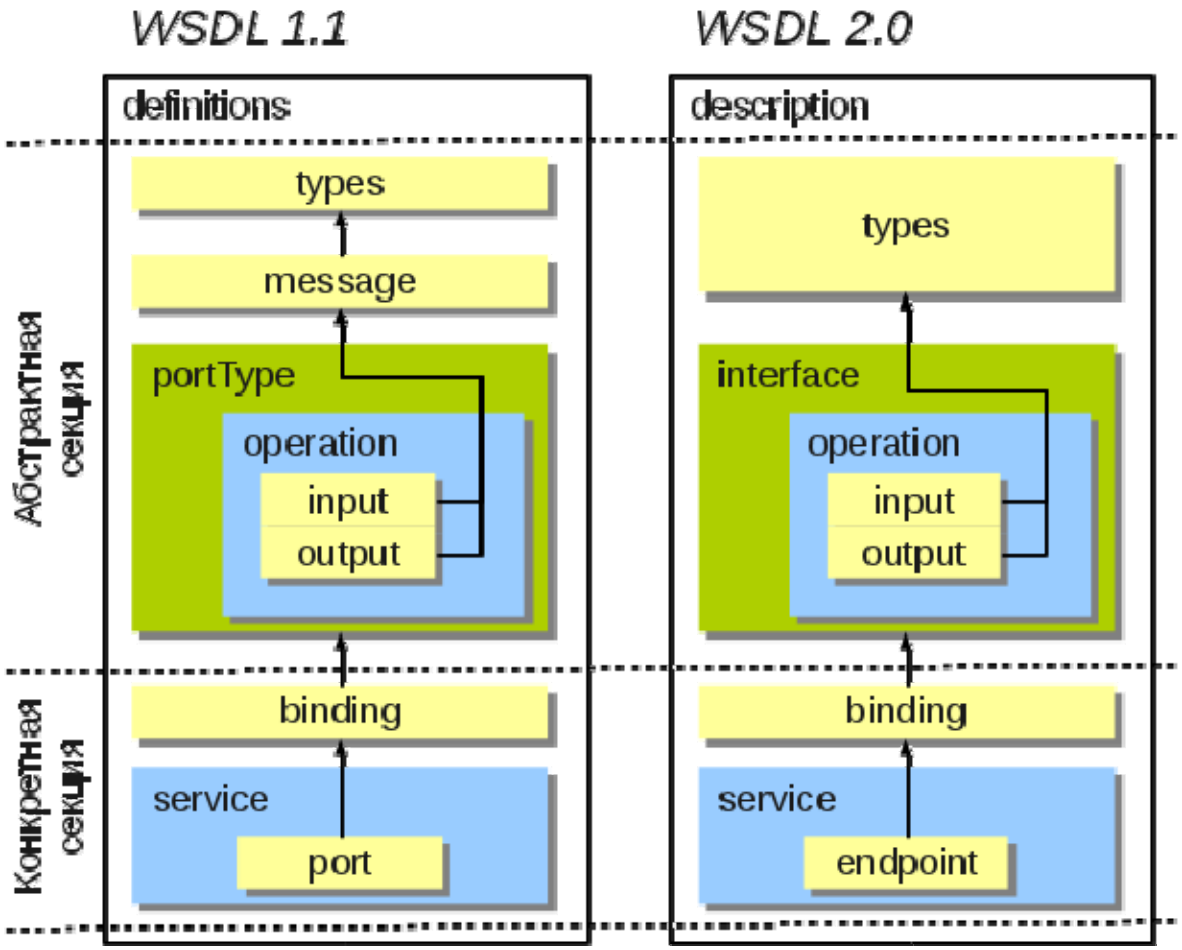


Рисунок 3 — Структура протокола WSDL

В спецификации WSDL 1.1 было определено 4 шаблона обмена сообщениями (типы операций):

Однонаправленные операции (One-way): операция может принимать сообщение, но не будет возвращать ответ.

Запрос-ответ (Request-response): операция может принять запрос и должна вернуть ответ.

Вопрос-ответ (Solicit-response): операция может послать запрос и будет ждать ответ на него.

Извещение (Notification): операция может послать сообщение, но не будет ожидать ответ.

В версии WSDL 2.0 эти шаблоны изменены и расширены в сторону поддержки сообщений об ошибках (например, шаблон Robust-in-only), но для обратной совместимости поддерживаются типы WSDL 1.1.

UDDI

Universal Description, Discovery and Integration (UDDI, универсальный интерфейс распознавания, описания и интеграции) — открытый стандарт, утвержденный OASIS, определяющий методы публикации и обнаружения сетевых программных компонентов сервис-ориентированной архитектуры (SOA). В практической реализации UDDI представляет собой сетевой реестр (службу каталогов), представляющий данные и метаданные о веб-сервисах.

UDDI опирается на отраслевые стандарты HTTP, XML, XML Schema (XSD), SOAP и WSDL. Концептуальная связь между UDDI и другими протоколами стека веб-сервисов показана на рис. 4.

Рисунок 4 — Место UDDI в стеке протоколов веб-служб

Функциональное назначение реестра UDDI — представление данных и метаданных о веб-службах. Он может использоваться как в сети общего пользования, так и в пределах внутренней инфраструктуры организации. Реестр UDDI предлагает основанный на стандартах механизм классификации, каталогизации и управления веб-службами, позволяющий применять их (веб-сервисы) другими приложениями. Этот механизм включает средства для поиска сервиса, вызова этой службы, а также для управления метаданными об этой службе.

Ключевыми функциями UDDI являются публикация информации о службе в реестре и поиск этой информации сторонними приложениями. Наряду с этими, реализованы и типовые для службы каталогов функции: представление модели хранимых данных и структуры информационной базы, отношения между объектами реестра, репликация, обеспечение безопасности и т.д. Все основные функции реестра представлены в виде веб-сервисов и доступны через API UDDI [2].

Архитектура REST

REST (Representational state transfer) — это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

А теперь тоже самое более наглядно:

Отсутствие дополнительных внутренних прослоек означает передачу данных в том же виде, что и сами данные. Т.е. мы не заворачиваем данные в XML, как это делает SOAP и XML-RPC, не используем AMF, как это делает Flash и т.д. Просто отдаем сами данные.

Каждая единица информации однозначно определяется URL — это значит, что URL по сути является первичным ключом для единицы данных. Т.е. например третья книга с книжной полки будет иметь вид /book/3, а 35 страница в этой книге — /book/3/page/35. Отсюда и получается строго заданный формат. Причем совершенно не имеет значения, в каком формате находятся данные по адресу /book/3/page/35 — это может быть и HTML, и отсканированная копия в виде jpeg-файла, и документ Microsoft Word.

Как происходит управление информацией сервиса — это целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же HTTP. Так вот, для HTTP действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия CRUD (Create-Read-Updtae-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST.

Вот как это будет выглядеть на примере:

GET /book/ — получить список всех книг

GET /book/3/ — получить книгу номер 3

PUT /book/ — добавить книгу (данные в теле запроса)

POST /book/3 — изменить книгу (данные в теле запроса)

DELETE /book/3 — удалить книгу

ВАЖНОЕ ДОПОЛНЕНИЕ: Существуют так называемые REST-Patterns, которые различаются связыванием HTTP-методов с тем, что они делают. В частности, разные паттерны по-разному рассматривают POST и PUT. Однако, PUT предназначен для создания, реплейса или апдейта, для POST это не определено (The POST operation is very generic and no specific meaning can be attached to it). Поэтому данный пример будет правильным и в таком виде, и в виде если поменять местами POST и PUT.

Использование REST для построения Web-сервисов.

Как известно, web-сервис — это приложение работающее в World Wide Web и доступ к которому предоставляется по HTTP-протоколу, а обмен информации идет с помощью формата XML. Следовательно, формат данных передаваемых в теле запроса будет всегда XML.

Для каждой единицы информации (info) определяется 5 действий. А именно:

GET /info/ (Index) — получает список всех объектов. Как правило, это упрощенный список, т.е. содержащий только поля идентификатора и названия объекта, без остальных данных.

GET /info/{id} (View) — получает полную информацию о объекте.

PUT /info/ или POST /info/ (Create) — создает новый объект. Данные передаются в теле запроса без применения кодирования, даже urlencode.

POST /info/{id} или PUT /info/{id} (Edit) — изменяет данные с идентификатором {id}, возможно заменяет их. Данные так же передаются в теле запроса, но в отличие от PUT здесь есть некоторый нюанс. Дело в том, что POST-запрос подразумевает наличие urldecoded-post-data. Т.е. если не применять кодирования — это нарушение стандарта. Тут кто как хочет – некоторые не обращают внимания на стандарт, некоторые используют какую-нибудь post-переменную.

DELETE /info/{id} (Delete) — удаляет данные с идентификатором {id}.

В данном примере /info/ — может и базироваться на какой-то другой информации, что может быть (и должно) быть отражено в URL: /data/4/otherdata/6/info/3/ , например.

Как видно, в архитектура REST очень проста в плане использования. По виду пришедшего запроса сразу можно определить, что он делает, не разбираясь в форматах (в отличие от SOAP, XML-RPC). Данные передаются без применения дополнительных слоев, поэтому REST считается менее ресурсоемким, поскольку не надо парсить запрос чтоб понять что он должен сделать и не надо переводить данные из одного формата в другой [3].

Литература

Гуриков, С.Р. Информатика : учебник / С.Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2021. – 463 с. – (Высшее образование: Бакалавриат)