JRE. Вы как разработчик написали код, отправляете его в компилятор. Для этого JDK и был нужен. На выходе получаете байт‑код. И после этого в этот байт код своему пользователю передаете, пользователь запустил у себя, он запускает ваше приложение. Запуск приложения на Java — это запуск JVM и загрузка в JVM вашего приложения, ваших байт‑кодов. Ваши байт‑коды, записаные в виде некоторых файлов. После этого JVM у пользователя загружает классы, проверяет и либо интерпретирует, либо компилирует ваши байт‑коды в нативный код машины и пользуясь ресурсами операционной системы это приложение пользователь исполняет.

1.2.2 Обзор платформы Java

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | JVM:  Interpreter;  Just-in-Time (JIT) compiler;  Byte code verifier;  Garbage collector (GC).  Javac:  превращает \*.java в \*.class;  \*.class можно собрать в \*.jar (zip архив) |

Картинка

В продолжении темы про Java‑платформу еще несколько слов о том, из чего состоит виртуальная Java‑машина и как она работает. В состав виртуальной машины входит интерпретатор. Его я уже упоминал в одном из предыдущих занятий. Когда вы написали байт‑код. Точнее, когда вы написали код, который был превращен компилятором в байт‑код и этот байт‑код вы передали внутрь JVM, JVM должна этот байт‑код понять. На деле он ей понятен в том смысле, что этот код написан для нее. Как раз код, содержащий инструкции для этой JVM. Для того, чтобы исполнить приложение JVM должна превратить этот код в нативный код того устройства, на котором она установлена. И сделать она это может 2-мя способами. 1-ый способ — это проинтерпретировать. То есть у нас есть с одной стороны некий байт‑код, с другой стороны мы должны получить команды для устройства. И мы можем бежать по этому байт‑коду и интерпретировать его инструкции в команды нативные. Можем сделать по‑другому. Можем скомпилировать, быстро скомпилировать, как английское название этого компилятора just‑in‑time. То есть он получает байт код, компилирует его и потом виртуальная машина исполняет уже результат, который получился после компиляции. Насколько я знаю, современные Java‑машины в общем добирают между интерпретатором и компилятором, причем могут еще вести статистики в зависимости от того, когда и сколько каких ресурсов было потрачено на ту или иную стадию. Кроме компиляторов и интерпретаторов в состав Java‑машины входит еще верификатор этого байт‑кода, то есть перед тем, как начать исполнять мы его проверяем. И состав Java машины входит сборщик мусора. Это отдельная большая служба, о которой я расскажу несколько раз коротко, но чуть более подробно на одном из следующих занятий. Если совсем коротко, то сборщик мусора — это та служба, которая анализирует ваши объекты и находит те из них, которые можно безопасно удалить, то есть те, которые ваше приложение уже не использует. Вы поработали с объектами, потом все ссылки на этот объект переставили на что‑то другое или просто обнулили и объект остался без ссылки на него. И собственно сборщик мусора — эта та часть JVM, которая с некоторой периодичностью проверяет ваши объекты, находят в ней живые и те, которые можно удалить. Еще пару слов про работу компилятора. Я упоминал его в том ключе, что он делает для вас байт‑код. Для каждого вашего Java‑файла компилятор создает файл с байт‑кодом. То есть процесс компиляции в Java — это процесс прохода дерева вашей файловой системы для проектной, в которой лежат ваши Java‑файлы. В каждом Java‑файле по одному Java‑классу. И по результатам компиляции у вас получится аналогичное же дерево файловой системы, в которой просто Java‑файл будет заменен на файл с расширение .class. И внутри этого файла будет байт‑код. При желании вы можете открыть и посмотреть, что у вас там получилось. Обычно все эти класс файлы целиком как они есть в дереве собирают в специальный ZIP‑архивы и называют их JAR‑фалами. То есть это Java­‑архив. В нем в той же иерархии, в которой у вас в проекте будут лежать файлы с байт‑кодом. Здесь еще пару слов. На картинке я здесь показал возможность сборки байт‑кода существует не только из Java. В Java машину должен поступить байт‑код. JVM может жить на разных устройствах и этот байт‑код будет интерпретировать. Откуда байт‑код взялся, JVM в принципе не так важно. Если он понимаемый ею байт код, то совершенно не важно на каком языке был оригинал написан. Так вот кроме Java compiler, которым мы будем пользоваться, есть возможность еще собрать байт‑код из Python. То есть вы написали код на Python, собрали в байт‑код. Если приложение написано одинаково (и то и другое), вы можете даже совершено одинаковый байт‑код получить. То есть вы получили здесь байт‑код, получили здесь байт‑код и потом вы этот байт‑код можете отправить в какую‑то из Java‑машин, которая находится уже на конкретном устройстве и в этот момент они уже совсем не будут знать о том на каком языке был исходник написан.

1.2.3 Редакции Java

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Java SE (Standard Edition)  Стандартная редакция  JVM + библиотека классов  Java EE (Enterprise Edition)  Java SE  Спецификация  API + runtime environment services  Java ME (Micro Edition)  Мобильные телефоны, бытовые приборы |

Несколько слов о редакциях Java. В литературе вы можете встретить добавочные 2 буквы к слову Java. В большинстве случаев SE или EE, реже ME. E в этом сокращении означает редакция, то есть Edition. Всего из распространенных редакций у нас 3. Это Standard Edition (SE), Enterprise Edition (EE) и Micro Edition (ME). Мы с вами будем в этом курсе разбирать Java SE, то есть у нас будет Java Standard Edition. Часто SE расшифровывают не как Standard Edition, а как Server Edition. На самом деле это ошибка, потому что в стандартную редакцию входят стандартные библиотеки Java. Больше половины из стандартной библиотеки Java посвящена работе с UA, то есть это пользовательские интерфейсы и совсем не сервер. Поэтому правильно говорить Standard Edition. И в SE входит JVM и библиотека классов, то есть то самое, что входит в состав JRE когда вы скачиваете, то есть некий минимальный набор возможностей для вашего приложения. То есть вы пишите приложение, запускаете его внутри Java машины. Она использует библиотеки, которые есть. И это вот называется Java SE. Кроме SE есть Java Enterprise Edition (EE). Задумка Enterprise в том, что когда вы запустили JVM вы внутрь JVM кроме собственного приложения можете еще поселить некий набор сервисов, которым ваше приложение может пользоваться, более того который могут вызывать ваше приложение. То есть вместо того, чтобы как в случае с SE, запустить свое приложение, которое использует статические библиотеки, вы запускаете приложение внутри некого окружения. И с этим окружением, которое не просто статическое, как библиотеки, которое живет рядом с вами, вы с этим окружением работаете. JE в текущий курс не входит. Поэтому то, что я сейчас об этом скажу об Enterprise, это будет все об Enterprise в этом курсе скажу. Если разбирать по составу, что входит в EE, то входит SE и плюс еще некий набор описаний как именно должны сервисы внутри JVM быть устроено, то есть некая спецификация того, как это все должно выглядеть. Существует также, как в случае с Java есть несколько разных вариантов Java есть оракловская Java, есть открытая Java. Также существует множество Enterprise решений джавовских, которые просто от разных компаний, удовлетворяют JE спецификаций. И есть еще Micro Edition (ME). Скорей всего она была популярна в начале 2000‑х. Сейчас все меньше. Она как раз нужна для бытовых приборов, для телефонов. Но и на практически всех современных смартфонах вы встретите уже не ME, а SE. В этом смысле ME есть, но скорее всего в своей работе с Java вы никогда не встретите.

1.2.4 Сборщик мусора

|  |  |
| --- | --- |
| 1.  2.  3. | Garbage collectior (GC)  Поиск ненужных объектов  Освобождение памяти |

Несколько слов о том, что такое сборщик мусора и как он работает. Сборщик мусора — это часть JVM, которая нужна для того, чтобы собирать за вами мусор. В процессе разработки приложения вы создаете объекты. Создаете вы их в куче. Куча — это специальное место в памяти, которое нужно для того, чтобы в нем создавать объекты. То, где вы создаете объекты, и где вы создаете ссылки на эти объекты, это другой участок памяти, который называется стек. Если сильно забежать вперед на каждый ваш поток у вас будет по стеку. И стек назван стеком, потому что вы каждый раз заходя в функцию подстраиваете ее в стек исполнения этих функций. И когда создаете переменную, вы создаете ее на стеке. Выходя из функции, вы теряете те переменные, которые вы внутри функции создали, если вы их не сохранили, допустим, в объекте. Процесс создания объектов в Java состоит из создания ссылки на стеке и создания объектов кучи. Когда вы пишите в своем приложении A a = new A(), то в левой части создаем ссылку, а в правой — объект. Ссылка живет на стеке, а объект — в хиппи. В процессе работы вы можете переставить свою ссылку на что‑то другое, например, занулить ее: a = null. Если кроме этой ссылки на объект больше никто не ссылался, то объект окажется без ссылки на него. Он будет занимать некую область в памяти, но добраться до него из приложения будет уже невозможно, потому что ссылка на ней и осталась. Вот, собственно, чтобы понять, что есть участки памяти, в которых лежат ненужные объекты и удалить их для этого и есть сборщик мусора. Он не входит в состав нашего приложения, он входит в состав Java‑машины. То есть он как бы живет рядом с нашим приложением, периодически останавливая и анализируя наше приложение. Что значит анализируя наше приложение? Он просматривает наши объекты и для каждого объекта, который он находит он пробегает по ссылкам, которые на него указывают. Если для объекта этот путь существует, то есть мы нашли путь до стека. Это означает, что объект живой. Если для объекта пути к стеку нет, никак из объекта по ссылкам на стек не попасть, то это недостижимый объект и из можно удалить. В ранних версиях Java сборщик мусора не понимал ссылки друг на друга. Но это давно поправили как раз тем способом, которым я сейчас рассказал. То есть анализ идет по всем ссылкам, которые есть на объект. И если есть путь до стека, то объект живой.

1.2.5 Наименования

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | Классы в Java собирают в пакеты (packages)  Пакеты обычно собирают в \*.jar файлы (zip архив)  Класс может быть однозначно идентифицирован по полному имени packageName.directoryName.ClassName  Имена пакетов и директорий принято начинать с маленькой буквы  Имена классов: MySuperClassName  Имена функций: getMyVariable()  Имена переменных: myVariable  Пакеты не могут иметь циклических зависимостей |

Перед тем, как перейти к следующей части занятия, на которой мы будем уже разбирать непосредственно разработку приложений и настройку окружения, я хотел бы остановиться и несколько слов сказать о правилах именования сущностей в Java. Если вы писали до этого на C++, то вы должны были столкнуться с таким понятием как name convention и кодпольси. И часто разработчик, приходя на новое место работы, первое что ему выдают рабочее место и выдают еще документ на несколько десятков страниц, в котором подробно описана как именно принято в этой компании именовать переменные, как организовывать свой код, чтобы он у всех был одинаковый. Разработчикам Java повезло. Как‑то так удачно получилось, что name convention у Java один, один на всех, вообще на всех разработчиков, поэтому приходя в другую компанию, где тоже пишут на Java или просматривая чьи‑то исходники вы увидите там точно такое же по формату код, который вы будете писать сами, если вы будете эти правила использовать. Основных этих правил немного. Первое с чего в этой теме нужно начать это то, что организация даже не столько имен переменных и классов, сколько организация самих файлов. Как вы помните каждый класс в Java лежит в отдельном файле и называется точно так же как название файла. Собственно, вот эти классы или файлы собирают в пакеты и потом при сборке приложения собирают еще в ZIP‑архивы. Собственно, ZIP‑архив называют как называется приложение, а пакеты называются маленькой буквой, как называется модуль, который отвечает за некую функциональность. Допустим модуль работает с базами данных, или модуль работает с интерфейсом. У каждого вашего класса есть короткое имя и есть полное имя. Полное имя состоит из полного пути к нему от корня вашего проекта. Пример на слайже. Если перейти к правилам наименования, то по адресу можно догадаться о правилах наименования. Это так называемый подход верблюда. По английский звучит чуть более благороднее, а смысл в том, что мы каждое слово начинаем с большой буквы, то есть у нас получается некие такие горбы в названии. Если вы хотите создать некую сущность, она у вас что‑то делает. В моем случае ClassName, то вы в каждое слово из которых эта сущность состоит начинаете с новой буквы, не делая между ними никаких пробелов, ни тире ни нижних подчеркиваний. Просто пристыковываете имена друг другу. И классы мы называем с большой буквы, а директории и пакеты с маленькой буквы. То есть начинаем с маленькой, потом большая, в случае классов — все большие. Как я уже и сказал функции принято назвать глаголами, то есть это должно быть что‑то, что указывает на действие, которое происходит. В данном случае — getMyVariable(). То есть я говорю, что переменную нужно вернуть. И функции мы начинаем с маленькой буквы. Потом так же, собственно, у нас появляется еще один верблюд. Имена переменных — это некое существительное, обычно так же как и имя класса. Обычно в единственном числе. Но только в отличие от класса оно с маленькой буквы. В примере всего один горб myVariable. Если вы этим простым правилом будете в организациях своих кодов соответствовать, то в общем любой из Java разработчиков легко поймет то, что вы написали.

1.3 Java. Запуск сервера

1.3.1 Настройка окружения. Инструменты

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | JDK 1.8  IntelliJ IDEA  Maven  Git  Jetty  freemarker | Java Development Kit 8  среда разработки приложения на Java  система сборки проекта  система контроля версий  библиотека для web сервера  шаблонизатор |

Основная задача курса — получение практических навыков. Поэтому для нас очень важно сразу же на первом же занятии настроить окружение, в котором мы будем работать. Я предлагаю вам использовать следующие инструменты. В разработке и в своих примерах я буду использовать именно этот набор. В первую очередь нам с вами понадобится JDK. Вы уже знаете, что такое JDK и для чего он нужен разработчику и пользователю Java. Мы с вами будем использовать 8 версию Java. Правда не все из того, что мы будем делать, будет использовать элементы именно 8 Java. Часть будет 7 и даже будут какие‑то элементы из 6 Java. По желанию, вы можете разобраться со всеми особенностями на 8 Java, полностью перейти на нее и использовать особенности именно 8. Потом нам с вами понадобиться среда разработки. Я предлагаю вам использовать IntelliJ IDEA, бесплатную версию. Практически всего функционала бесплатной версии нам будет с вами достаточно. На мой взгляд — это одно из лучших сред разработки для Java, может быть вообще для любого языка, который вообще существует. Нам с вами понадобиться Maven. Maven мы будем с вами в основном выкачивать зависимости. То есть выкачивать библиотеки, которые нам понадобятся в добавок к стандартным библиотекам Java. Частично будем собирать проекты Maven, но я думаю, что все‑таки основная работа по сборке у нас будет происходить в IDEA. И нам понадобиться система контролей версий. Я вам рекомендую Git. И мы скорей всего будем использовать GitHub, как всемирно известный бесплатный и удобный ресурс для хранения открытых проектов. Дальше нам с вами понадобиться уже для работы в этом уроке 2 библиотеки. Библиотека Jetty. Мы на ней будем строить веб­‑сервер. И нам понадобиться freemarker. Такой простой, но удобный надежный шаблонизатор.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | https://www.oracle.com/technetwork/java/  javase/downloads  Lunix  export JAVA\_HOME = your java directory в файле /etc/profile  Windows  Control Panel -> System -> Advanced добавьте в переменные среды JAVA\_HOME = your java directory |

Итак, первое, что вам нужно будет сделать перед тем как начать работать, — это настроить компьютер, на котором вы будете работать. Вам на нем понадобиться JDK и в зависимости от операционной системы вы можете выкачать ту, которая вам нужна и поставить ее. На слайде приведены настройки, которые вам нужно сделать в своей операционной системе для того, чтобы у вас программы, запускаемые операционной системой, знали где искать виртуальную машину.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | https://www.jetbrains.com/idea/download  Community Edition FREE  Linux: запуск через bin/idea.sh  Windows: обычная установка и запуск |

Не сложно скачать приложение для разработки. Это среда, в которой вы будете в основном писать коды. Компилятору все равно как вы будете писать код. Можете писать его в любом текстовом редакторе. Но удобно, когда редактор, в который вы пишите, еще помогает вам в вашей работе. IDEA, на мой взгляд, помогает лучше всего. Ничего особенного в запуске нет. Сама IDEA написана на Java, и ей для работы нужна JDK. То есть если вы не прошли предыдущий шаг установки виртуальной машины себе, то IDEA запустить тоже не сможете. В Linux, когда вы запускаете через ShelScript, первым делом он пытается найти себе куда вы установили JDK. В принципе, вы можете открыть в текстовом редакторе и посмотреть, что он делает. Вам просто надо будет настроить себе пути и запомнить их в config, чтобы при старте не сбросились.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | Распакуйте архив в инсталляционную директорию.  Windows: C:\ProgramFiles\maven\  Linux: /opt/maven  Установите переменную окружения  M2\_HOME = инсталляционная директория  Установите переменную окружения PATH  Windows: в переменной PATH добавьте к списку директорий строку %M2\_HOME%\bin  Linux: можно добавить строку export PATH=$PATH$M2\_HOME/bin в файл /etc/profile  Запустите в командной строке mvn -version |

Дальше нам понадобиться Maven. Maven — это система контроля зависимости и сборки проектов. То есть это приложение, помогающее вам выкачивать нужные вам версии библиотек и собирать проект. Значит у Maven есть большой центральный репозиторий. Вы можете набрать Maven в поисковике и первой же ссылкой получите ссылку на этот репозиторий. Вы можете посмотреть какие в нем библиотеки лежат. Их там невероятное количество. Все что нам может понадобиться в Maven репозитории в центральном есть. Наше использование Maven сведется к тому, что мы будем говорить, что нам нужно выкачать. Maven нам будет выкачивать сам. Вам понадобиться интернет, чтобы выкачивать библиотеки Maven как минимум при первом старте, при первом открытии вашего проекта в среде разработки. Настройка чуть сложнее чем все предыдущие, хотя ничего особенно страшного нет, нужно просто проставить несколько переменных окружения. В Linux и в Windows будет немного отличаться, но я думаю, что любой уверенный пользователь этих операционных систем разберется. И в конце, чтобы понять, что вы все сделали правильно, вы можете просто в командной строке набрать mvn -version и Maven вам покажет какая версия была установлена. Если при запуске командной строки вы видите mvn не найден, значит что‑то у вас пошло не так. Последние версии IntelliJ IDEA содержит в себе Maven уже встроено. Все эти действия, которые на этом слайде перечислены, нужны, если вы собираетесь собирать проекты вне IDEA. В принципе я вам рекомендую все это проделать. Если вдруг сразу не получилось, будем использовать то, что у вас в IDEA. Если понадобиться больше, потом до настроите.

1.3.2 Создание проекта

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | Project Object Model — pom.xml  <groupId> адрес проекта </groupId>  <artefactId> название приложения  </artefactId>  <version> версия приложения </version>  <dependencies>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artefactId>jetty-server</artefactId>  <version>9.3.0.M1</version>  <dependency>  </dependencies> |

Итак, если вы успешно проделали все предыдущие пункты, то у вас на рабочей машине уже установлена JDK нужной версии, есть среда разработки, стоит Maven, и вы готовы уже начать создавать проект. Я вам предлагаю первым делом зайти на сайт JetBrains, разработчика IntelliJ IDEA, и почитать как именно нужно создавать проекты под управлением Maven. После того, как вы пройдете все шаги, которые вам рекомендует в инструкции, у вас получится пустой проект, и в нем будет файл с именем pom.xml (Project Object Model). В нем вы будете указывать какие именно библиотеки вы хотите подключить к своему проекту. И как именно вы хотите свой проект собирать, если вы хотите его собирать отдельно не из IDEA. В каждом pom файле вы найдете блок с описанием вашего проекта. Он будет содержать groupId, artefactId и version. И в этот же pom файл вы сможете записывать зависимости, которые вы хотите, чтобы у вашего проекта были на библиотеки других разработчиков. И при включении зависимости вы также должны указать groupId, artefactId и version того, что вы хотите выкачать. На слайде 8‑14 я привел пример зависимости на библиотеку Jetty, на Jetty сервер с версией 9.3.0.M1. Естественно время идет и версии будут у библиотек расти. Те версии, которые я указываю в своих слайдах, это те версии, которые сейчас у меня в примерах, которые у меня работают. В принципе вы всегда можете работать с теми версиями, которые есть у меня. Если вам нужно только библиотеки для примеров. Если вы в дальнейшем планируете разрабатывать проект дальше курса и использовать его как‑то еще в добавок тому, как мы рассказываем в курсе, то вам хорошо бы было следить за версиями библиотек и расставить зависимости на последнее.

После того как вы все это сделаете, у вас уже будет проект, в нем уже будут зависимости и будет самое время уже потренироваться выложить этот проект в систему контроля версии. Посмотрите, что нам предлагает GitHub и выкачайте себе непосредственно Git на машину, поставьте его здесь. На этом слайде ссылки, которые вам понадобиться. В дальнейшем я буду показывать везде ссылки на свои примеры, которые тоже лежат на GitHub. Заходите в мой профиль, смотрите какие у меня есть проекты. Выкачивайте любые примеры, которые вам нужны, разбирайте. Для обмена файлами друг с другом нам Git и будет нужен.

1.3.3 Jetty

Мы с вами разобрались с тем, что такое Java, историю Java, разобрали внутреннее устройство Java‑машины, посмотрели какие ей нужны знания для успешного прохождения курса, настроили себе окружение, выкачали все нужные библиотеки, создали первый проект под управлением Maven. Следующее наше с вами действие — перейти непосредственно к разработке, то есть оставшиеся 3 части я буду рассказывать о хетпати библиотеках. Решение, которое мы будем использовать уже непосредственно для того, чтобы создать проект и сделать некое подобие HelloWorld, которое у нас будет в проекте. Только этот HeloWorld будет уровня общения пользователя из веба с нашим сервером.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Java based http server  создание динамических страниц  пересылка статических файлов (html страницы, картинки…)  <dependencies>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artefactId>jetty-server</artefactId>  <version>9.3.0.M1</version>  <dependency>  </dependencies> |

Итак, первое что нам с вами понадобиться это библиотека Jetty. Перед тем, как рассказать про Jetty я хочу рассказать пару слов о том, как у нас вообще будет устроена работа с пользователем. В этой части курса, части посвященной бекэнду, мы с вами будем заниматься непосредственно разработкой сервера, того, что реагирует на запросы пользователей. У нас будет то место нашего веб‑приложения, которое отвечает за обработку запросов за формирование ответов пользователю на основе данных, которые пришли от пользователя, а также на основе нами собственно вычисленных данных или данных, взятых из базы. Мы с вами существенно облегчим себе работу тем, что не будем писать свой собственный клиент. Клиент у нас — это браузер. Вы будете отдельно программировать свое приложение во второй части нашего курса так, чтобы браузер показывал пользователю красиво и удобно то, что мы ему вернули. Наша задача будет обработать запросы и сформировать ответы. Упрощение, которое я предлагаю сделать в данном случае, — это кроме того, что мы не пишем свой собственный браузер, то есть в качестве клиента мы используем уже готовый браузер, мы с вами также не будем писать протокол общения между клиентом и сервером, то есть в качестве протокола у нас будет http запрос. Та часть нашего сервера, которая непосредственно будет работать с браузером и с его запросами, то есть работать в смысле принимать эти запросы — это будет некий http сервер. В качестве этого http сервера я вам предлагаю использовать Jetty. То есть смотрите, наша с вами задача будет писать обработчики запросов пользователя. То, как именно эти запросы уходят от пользователя, от его браузера и прилетают к нам, мы с вами писать не будем. То есть ни клиент, ни протокол общения, ни непосредственно уровень приема запросов мы с вами разрабатывать не будем. Вместо них будем использовать уже готовое решение. Нам и без этого достаточно что сделать. Та часть нашего сервера, которая будет принимать запросы пользователя, запросы из разных браузеров, это и есть http сервер, в нашем случае Jetty. Первое что вам нужно будет сделать для того, чтобы начать общаться с браузером — это подключить себе его в ваше приложение. <dependency> на него я уже показывал. Вам нужно будет открыть pom файл вашего проекта, вставить <dependency> как есть, может быть зайти на сайт Maven и посмотреть какая сейчас версия последняя, поменяйте версию, посмотрите. Те версии, которые у меня здесь, я их протестировал. Если вы выставляете свои версии, там могут начаться какие‑то особенности. Например, какие‑то классы могут показаться уже deprecated. То есть вам придется самостоятельно согласовывать версии. С первого раза я вам предлагаю поставить мои версии, а потом, если вы выходите уже на новый уровень разработки, включать следующие. После того, как вы это включили, в вашем приложении появится возможность использовать Java бейст http сервер, то есть библиотека, которая вашему приложению позволяет обрабатывать запросы, которые в ваш сервер будут поступать из браузера. И наша задача в результате будет сведена к тому, что мы должны будем написать у себя в коде обработчики для этих событий. Мы с помощью Jetty. Jetty нам просто скажет унаследуйте класс от определенных базовых типов, реализуйте некий набор функций и в том случае, если пользователь что‑то захочет от вас получить, то его запрос приведет к вызову функции, которую вы напишите. Это и является обработчиком. Еще раз. У нас клиент серверное приложение, мы самостоятельно с вами клиент не пишем, мы программируем для клиента в другой части курса. Клиент этот, браузер, отправляет запрос по протоколу, который мы тоже не пишем. Для того чтобы понимать этот протокол и этот запрос у нас с вами в составе сервера будет хетпати библиотека Jetty, она получит эти запросы и по виду этих запросов поймет какие именно функции вызвать в нашем приложении. В эти функции она заодно еще отправит request от пользователя. request — это уже название класса. В этом request будет информация о запросе. После того как мы из этого request всю нужную информацию возьмем и обработаем ее. Например, если будет request на авторизацию или регистрацию пользователя. Сделаем нужные действия и в ответ отправим response. response — это тоже название класса. Что значит отправим? Значит, что мы из своей функции вернем response, либо мы его поменяем и Jetty сама поймет, как его поменяли. В ответ нам нужно будет сказать код результата и строку. Это строка, которая в браузере будет превращена в страницу, которую браузер отработает. Еще раз. Если вы работали с http серверами, такими как Apache или Nginx, то вы в общем то представляете что это. Вы поднимаете некий сервис. Он обрабатывает запросы. По запросам возвращает в случае Apache и Nginx статический файл. Jetty тоже умеет возвращать статические файлы. Вы можете настроить Jetty так, что она будет запросы на статику перенаправлять в некоторую директорию. В этой директории искать файлы с соответствующим названием и возвращать их. Кроме того, что Jetty умеет возвращать статику, она также умеет, собственно, динамические страницы. Это то самое, о чем я сейчас рассказывал. Динамические страницы — это страницы, созданные на основе запросов. И наша с вами задача будет написать обработчики для создания динамических запросов. Сама обвязка, которая приводит к вызову этих обработчиков, эту часть мы поручим Jetty.

1.3.4 Jetty

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Server:  Connector  handler(s)  ThreadPool | принимает http запросы (request)  возвращают response  pool потоков для обработки запроса |

Теперь, когда мы уже понимаем зачем нам библиотека Jetty и смогли ее подключить в наш проект, несколько слов о том, как она работает и как она устроена. На сайте Jetty есть статья об устройстве и работе Jetty. И там есть эта замечательная диаграмма, которая есть на этом слайде. К ней подпись, что это взгляд на Jetty с высоты птичьего полета. Очень в крупные блоки собраны части, из которых Jetty состоит. Нам для ознакомления вполне достаточно такого взгляда с высоты птичьего полета. Если начать разбирать внутренне устройство, первое о чем нужно сказать — это Connector. Эта та часть Jetty сервера, которая непосредственно принимает запрос от браузеров, от разных пользователей. То есть если пользователь что‑то запрашивает, он должен задать адрес, по которому он запрашивает, адрес и потом документ, который он запрашивает. После того, как он сформировал запрос, запрос будет понят браузером и отправлен на машину, которой этот адрес принадлежит. На этой машине на определенном порту, стандартный порт для http запросов 80, должен висеть обработчик, http сервер, который весит на порту и слушает запросы, которые на этот порт приходят. Jetty сделает так, что на наш сервер при старте тоже будет цепляться к некому порту. Цепляться и слушать запросы. И вот та часть, которая непосредственно слушает запросы пользователя, которая работает с портом и с запросами — это connector. После того, как данные в connector, то есть мы сформировали некий набор запросов. Дальше процесс исполнения приводит к тому, что мы вызываем, то есть мы должны неким образом этот вызов должен быть понят Jetty, либо превращен в отдачу статических данных обратно пользователю, как ответ на тот запрос, который от пользователя пришел. Либо мы должны обработать этот запрос каких‑то из наших обработчиков самостоятельно записанным. Для того, чтобы обработать запрос у Jetty есть TreadPool. Это часть сервера, в которой лежат заготовленные потоки. Когда мы дойдем до многопоточности, эта часть будет чуть более понятна. Пока что представьте себе это как остановленные линии конвейера. То есть у вас завод, в нем много стоящих неработающих линий конвейера и когда приходят пользователи, запрос пользователя сделать что­‑то, Jetty выбирает какую‑то из этих линий, в данном случае поток из пула, запускает его и в нем исполняет задачу. ThreadPool — это набор заготовленных элементов, которые могут исполнять что‑то. Взяв запрос и взяв поток из пула сервер может в нем вызвать непосредственно обработчик. То есть Hendler — это то, что непосредственно обработает запрос. Это может быть опять же запрос к файловой системе, которая приведет к отдаче статического файла, либо это запрос к нашему коду для того, чтобы в нашем коде сформировать ответ и вернут его пользователю.

1.3.5 Jetty

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public class Main {  public static void main(String[] args)  throws Exception {  Frontend frontend = new Frontend();  Server server = new Server(8080);  ServletContextHandler context =  new ServeletContextHandler(  ServeletContextHandler  .SESSIONS);  server.setHandler(context);  context.addServelet(  new ServeletHolder(frontend),  '/authform');  server.start();  server.join();  }  } |

После того как мы подключили Jetty, разобрались зачем вообще она нам нужна и из чего и каких больших блоков она состоит и как работает, самое время уже посмотреть участок кода, который позволит нам Jetty запустить. Здесь я привел класс Main и в нем функцию main(). Название класса Main произвольное. В принципе функция main() может быть в любом из Java классов. А вот сигнатура main() уже обязательная. То есть вы хотите создать точку входу в вашу приложение, вы должны написать 2 и передать аргумент. В этом примере первым делом я создаю обработчик. Frontend (4), который здесь указан, — это мой собственный класс. Как я уже говорил, чтобы создать объект в Java нужно вызвать new. Что такое Frontend я вам покажу чуть позже. Здесь пока рассказ о том, как запустить сервер. Server (6) — сервер Jetty. Это основной класс библиотеки Jetty. И при старте сервера мы должны еще ему сказать самую важную для него информацию, это то, на каком порту он будет стартовать. В данном случае в этой строке я создаю объект Jetty сервера и говорю ему, что он должен работать на порту 8080. Это обычный тестовый порт для разработки приложения. Дальше мне понадобится создание сервлет‑контейнера ServletContextHandler и в него мне нужно будет передать то, что будет обрабатывать запросы. То есть, когда я говорю здесь frontend (13), это объект, который я ранее создал. Я говорю, что этот обработчик должен обрабатывать запрос, который придет на запрос документа authform (14). Кога пользователь захочет обратиться к моему серверу, он наберет адрес. В моих примерах это будет localhost. Он наберет порт. То есть запрос от пользователя может выглядеть следующим образом. У нас будет http://localhost:8080. И после этого пользователь еще должен будет написать документ, который он хочет запросить. Это будет authform. Если пользователь все это сделает и если сервер запущен в таком виде, то запрос от пользователя придет внутрь обработчика объекта класса Frontend.

1.3.6 Jetty. Сервлеты

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Servlet  ServletContainer | класс, расширяющий возможности сервера  как aplet, только на web сервере  java решение для создания динамических страниц  объект, который обрабатывает http запрос (request) и возвращает ответ (response) — html страницу  часть web сервера, содержащая сервелеты  связывает URL с сервелетом |

Дальнейший разбор работы с Jetty и с обработчиками невозможен без упоминаний сервлетов. Потому что на предыдущем примере в общем то я уже сказал слова сервлет и сервлет‑контейнер. Мы с вами, используя Jetty, разрабатывая веб‑сервер, будем писать сервлеты. То есть наша задача использовать готовый веб‑сервер и объяснить ему какие сервлеты для чего использовать. Название сервлет не очень удачно. Изначально было 2 родственных понятия сервлет и апплет. То есть апплет появился раньше, как некая сущность, расширяющая возможности браузера. И так как апплет был на фронтендной стороне, то решили что давайте на серверной стороне тоже будем называть то, что расширяет возможности сервера, словом сервлет. Если вы настраивали допустим Apache, который для динамических страниц PHP, то примерно можете представить себе, что такое сервлет. Apache задуман как для отдачи статических страничек. Но его можно расширить неким скриптовым языком, например, PHP, который позволит создавать странички динамические. И на каждую страничку вы писали файлик, собственно, название файла .php. И вот это прямой аналог сервлета. Jetty сервер нам дает возможность отдавать статические документы и он вообще нужен для того, чтобы принимать запросы. А мы еще можем ему в добавок дать возможность обрабатывать запросы динамические. Пользователю не важно, как мы запросы обрабатываем. Это мы у себя знаем, что они динамические. Пользователь просто присылать нам некоторые данные. Он присылает нам request. А мы для этих request можем писать код, который не просто отдаст ему готовую страницу, а который на основе данных этих request залазит в базу, соберет какие‑то материалы и отдаст ему. Это как раз и есть действие сервлета. У нас сервлет — это обработчик, привязанный к некому запросу. А запрос со стороны пользователя — это запрос документа. Пользователь запрашивает документ, а мы знаем, что этот документ на самом деле не документ в файловой системе какой‑то файл, а запрос на динамическое создание контента. Мы с вами будем на каждый пользовательский запрос писать сервлет либо на группу запросов и в нашем понимании сервлет это будет наследник некого класса, который будет содержать функции, которую мы напишем для того, чтобы запрос обработать. После обработки мы должны сформировать страницу так, как будто Jetty прочитал эту страницу из файла. И пользователю, и Jetty на самом деле все равно откуда взялась строка, которой мы даем в качестве response. Эту строку можно получить чтение файла, а можно получить динамически, например запросом к базе. И в response и том и в другом случае у Jetty будет строка. Наша задача по запросу пользователя создавать строки для разных запросов разные и эти строки в виде response пользователю отдавать. И вот за то, чтобы формировать динамический контент пользователю мы будем использовать сервлет. На этом слайде я упоминаю еще севрлет‑контейнер. Это та часть сервера, которая содержит мапинг документа, который пользователь запрашивает на сервлет, который этот документ обрабатывает.

1.3.7 Jetty

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | <dependencies>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artefactId>jetty-webapp</artefactId>  <version>9.3.0.M0</version>  <dependency>  </dependencies> |

Перед тем, как начать использовать сервлеты, нам эти сервлеты нужно подключить к себе. На этом слайде я привел зависимость. Эту зависимость вы точно так же, как зависимость на Jetty сервер, должны включить в блок с зависимостями своего pom документа. После того, как вы это сделаете, среда разработки спросит у вас хотите ли вы выкачать версию. Соглашайтесь, она вам выкачает добавку к Jetty, которая непосредственно работает с сервлетами.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public class Frontend extends HttpServlet {  private Sring login = '';  public void doGet(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException,  IOException {  ...  }  public void doPost(  HttpServletRequest request,  HttpServletResponse response)  throws ServletException,  IOException {  ...  }  } |

После того, как все это будет сделано, вы можете уже написать вы можете написать свой собственно первый сервлет. Это тот самый класс Frontend, который я уже упоминал. Это то самое название, которое было у нас в примере одного из предыдущих шагов. И как я обещал он будет наследовать некий базовый класс. В нашем случае это HttpServelet. Вот как раз HttpServelet — это класс, который лежит в библиотеке, который вот только что с зависимостями выкачали себе. В каждом HttpServelet можно разместить 2 функции doGet и doPost. Функции, которые будут вызваны Jetty в тот момент, когда на этот сервлет придет Get или POST запрос. Смотрите еще раз. У нас есть Jetty сервер. Jetty сервер цепляется к некому порту и слушает запросы. Внутри Jetty сервера есть настройка, которая говорит с какой директории брать статические документы. И есть настройка, которая позволяет привязывать запросы некого документа к некоторому сервлету. Если вы вернетесь к предыдущему примеру, вы увидите, что сторону фронтедн я привязал к запросу на AU форм. То есть это некая страничка. Если пользователь ее запросит. Скорее даже так. Пользователь запросит ее при отправке формы. То есть пользователь запрашивает наш сервер. В индексе получает статический документ с формой. В форме заполняет поля авторизационные. И в форме указано куда пойдет запрос, на какой адрес. И это адрес как раз AU форм. После того как пользователь форму заполнил и нажимает отправить, браузер подсоединяется к нашему серверу, тому порту на котором висит Jetty и слушает. Jetty находит в потоке пулов свободный поток, поручает этому потоку обработать запрос и в зависимости от того как именно вы отправляли форму get или post запросом в этом потоке будет вызвана либо функция doGet, либо функция doPost. В этих функциях как вы можете видеть нам пришло еще 2 переменных. Это request и response. И в doGet, и в doPost. И в request нам Jetty сообщает все те переменные, которые нам могут понадобиться пользователи. Те, которые он сам им передает, и те которые является его переменными окружения и в response мы с вами должны будем записать ответ, который мы хотим пользователю отдать. То есть в response мы проставляем ключ результата. И в response мы пишем строку, которая потом превратится в страницу у пользователя. Браузер получит эту строку и отобразит ее как страницу.

1.3.8 Шаблонизатор

Картинка

В заключении этого урока я хочу вам рассказать еще об одной библиотеке, которая упростит нашу работу по созданию страниц. Как я уже упоминал раньше пользователь от нас ждет response. Браузер пользователя джет response. В response мы пользователю даем строку. Браузеру все равно откуда взялась строка. Прочитана она из файла. То есть что такое отдать пользователю текстовой файл или статическую html‑страницу — это прочитать содержимое текстовой страницы и отдать ее в браузер. То есть в общем‑то браузеру в любом случае прилетает строка. Просто мы знаем, что в этой строке страница. И браузер понимает, что это страница. А можно эту строку взять и сформировать программно. Не читать из файла, а прям взять и целиком написать у себя в приложении строку, захаркодить ее и собственно отдавать из класса. Это можно так делать, но это будет не удобно. При работе с вашими страницами на самом деле, при создании динамического контента в большинстве случаев не нужно формировать целиком всю страницу. Вам нужно в ней поменять какой‑то набор полей. То есть написать, пользователь авторизуется, и вы должны после авторизации написать ему «Здравствуй» и указать его имя. Та часть страницы, которая содержит заголовок, кодировку, разметку, указания на стили — она не будет меняться от запроса к запросу. У вас меняться будет только слово имя пользователя. И все. И для того, чтобы не держать у себя в коде всю страницу со всеми разметками и с строками, которые вы не собираетесь менять, есть шаблонизатор. Это библиотеки, позволяющие вам совместить данные, полученные из самого файла, и данные, полученные каким‑то динамическим путем. То есть у вас, с одной стороны, есть файл, обычный файл, который лежит на диске, то есть в этом файле вы при старте вашего сервера говорите где эти файлы искать. С другой стороны, у вас есть поля, которые вы пользователю хотите вернуть. Библиотека, в нашем случае FreeMarker позволяет вам совместить плюсы от файла и значение полей. Все это вместе перемешать, пользователю вернуть уже сформированную страницу. В принципе совсем не обязательно использовать ее именно для генерации страниц. То есть FreeMarker, он может вам понадобиться, когда вы просто захотите формировать строки, часть которых создано статически и лежит в файликах и часть которых вы создадите динамически.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | <html>  <head>  <title>Пример шаблона</title>  </head>  <body>  <p>Имя пользователя: ${userName}</p>  <p>Id пользователя: ${userId}</p>  </body>  </html> |

Такая строка может быть. Этот файлик лежит на диске. В этом файлике у вас указаны поля, которые вы хотите динамически подменять. Вот такая нотация ${имяПеременной} она довольно распространена и относится не только к FreeMarker. Вы, используя другие шаблонизаторы, тоже ее встретите. Вы создаете страницу. В странице пишите места, которые должны быть заменены при формировании уже результирующего документа. После того, как вы свой код включите в библиотеку, вы сможете взять файлик, взять переменные, совместить их, получить строку, положить ее в response и отдать ее пользователю. Подробности использования FreeMarker и Jetty и использования севрелетов в Jetty я покажу при разборе примера

Видеоразбор примера.

2 Авторизация

2.1 Постановка задачи, дженерики

2.1.1 Ознакомление

Добрый день. Давайте начнем второе занятие курса. По традиции мы начнем его с разбора плана урока, который нам с вами предстоит. В первую очередь мы с вами посмотрим задачу, которую нам нужно сделать. Мне кажется, это очень правильно начинать с занятия с описания того, что должно быть сделано в занятии и потом в процессе разбора тем занятий подчеркивать чем именно материал, который мы смотрим поможет нам в решении этой задачи. Первое же что нам понадобиться для решения нашей задачи — это дженерики. Без дженериков мы с вами не сможем перейти к контейнерам и коллекциям в Java. Коллекции — это будет третий пункт нашего урока. Разберем коллекции стандартной библиотеки. Не все, но многие те, которые нам с вами понадобятся. Потом вернемся к вебевой части нашей разработки, посмотрим что такое Http сессии и как нам с ней работать. Далее вернемся снова к базовым основам языка. Разберемся с наследованием, разберем что такое объекты и классы. В Java эти два ключевых слова они имеют некое особое значение больше, чем в большинстве других языков. Потом разберем очень важную тему — примитивные типы. Это собственно те базовые элементы, на которых построено любое приложение. Если вы начнете спускаться по иерархии с самого высокого уровня абстракции до самого нижнего, то в конечном итоге вы обнаружите внизу примитивные типы и ссылки. Обязательно нам надо будет с вами еще рассмотреть массивы и строки, их особенности в Java. И посмотреть особенность работы Java с памятью, сколько место Java выделяет под наши объекты.

2.1.2 Задача

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Сессия пользователя  1. Регистрация  2. Авторизация  3. Игра  4. Выход |

Итак, давайте разберем задание, которое нам предстоит в этом уроке сделать. Я предлагаю вам в качестве задания написать веб сервер, на котором нужно будет авторизоваться. Заготовка, которая нужна для любого дальнейшего использования этого веб‑сервера. Как бы вы не планировали его использовать в качестве игрового сервера или в качестве CRM‑системы. В общем не важно. Даже если вы пишите собственную социальную сеть. Вам в любом случае понадобиться механизм регистрации и аутентификации пользователя. Наше задание с вами сейчас — это сделать первый прототип авторизации на нашем сервере. Как видно на слайде — первая наша задача будет сделать регистрацию пользователей. То есть то, что позволит пользователю сказать, что он хочет нашим сервисом в дальнейшем пользоваться, чтобы система узнавала его. Авторизация подразумевает, что пользователь уже есть в нашей системе, что мы его уже когда‑то запомнили. И он хочет нам представиться. Дальше следует некий пункт. Содержательное взаимодействие пользователя с сервером. И это мы пока с вами в этом занятии не будем. То есть третья часть у нас остается пока не написанной. В эту 3-тью часть можно будет вставить какое‑то наполнение, под которым я подразумеваю некое взаимодействие с пользователем, которое вы в дальнейшем напишите. Четвертый момент — это выход пользователя из системы. Когда он говорит я в общем‑то уже пользоваться вашим сервисом и хочу, чтобы все остальные пользователи, если запросят онлайн я или нет, уже знали, что я вышел.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Упрощения  1. Без записи в базу данных  2. Без интерфейсов  3. Потоками управляет Jetty  4. «Потокоопасные» карты |

В процессе работы нам с вами, так как мы только начинаем разработку придется сделать некие упрощение системы. Не все мы будем делать полноценно и сразу. Мы вместо некоторых сервисов сделаем пока заглушки. И самое важное здесь это то, что мы пока ничего не будем писать в базу. Базу мы разберем четь позже. У нас просто будет запись в runtime. То есть вы подняли сервер, пользователь пришел, зарегистрировался, залогинился, мы его узнали. Потом мы погасили сервер и запустили снова. Он приходит, мы уже его не знаем, потому что у нас runtime информация была потеряна при старте. Чтобы такой потери не происходило как раз и нужны базы. Мы их разберем позже. Пока что у нас никаких баз не будет. Пока что мы с вами не будем делить сервер на интерфейс и реализацию. Это некая архитектурное упрощение системы. Оно подходит для небольших проектов и пока для первого прототипа нам тоже подойдет. Хотя в дальнейшем я буду настаивать на том, чтобы для всех сервисов, которые мы пишите вы обязательно производили разделение и особенно будет понятно зачем это нужно, когда мы с вами подойдем к тестированию. У нас 3-ий пункт в упрощениях. Мы будем использовать в работе потоки, которые нам даст Jetty. Когда я рассказывал про Jetty, я говорил, что у нее есть отдельный пул потоков, сравнивал пул потоков с некой лентой конвейеров. Когда есть некая задача вы можете запустить конвейер, обработать ее, остановить конвейер. Под этими конвейерами потоками я подразумеваю что это даст нам Jetty. У нас будет поток Main, но вы в нем произведем всю инициализацию, которая нам нужна, а потом просто скажем просто подождать. Так как тема многопоточности пока еще не является для вас, ну мы просто не можем к ней приступить, потому что еще только в самом начале пути, то нам придется здесь использовать поток безопасной карты. То есть в том виде, в каком мы с вами напишем сервер, его можно в общем то использовать в продакшн тоже, хотя если вы все таки планируете это делать, то вам обязательно нужно будет послушать тему про потоки, иначе вы получите некоторые странности в поведении сервера, если к вам придет более одного пользователя одновременно. Веб‑сервер очевидно должен выдерживать нагрузку больше, чем один пользователь в один промежуток времени.

2.1.3 Задача

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Регистрация  Frontend  1. Страница с вводом: логин, email, пароль  Backend  1. Сервлет обработки запроса  2. Класс AccountService с методом регистрации  3. Класс UserProfile с полями: логин, email, пароль  4. Карта логин -> UserProfile |

И теперь разберем страницы, которые нам с вами понадобятся. Когда на прошлом занятии я говорил вам про сервлеты в общем то я и говорил про эти самые страницы. То есть вспоминайте материал прошлого урока и на нем мы говорили, что для того чтобы кастомно каким‑то образом нестатически обработать запрос пользователя мы должны к этому запросу подключить некий сервлет. Если пользователь у нас просто запрашивает некую статическую страницу, текст или картинку, то в общем то нам просто можно отдать ему файлик с содержимым и все. А если пользователь хочет от нас некого действия. Например, как в случае с регистрацией он нам присылает данные, где среди этих данных вычисляет свое имя, email, пароль. И пользователь хочет, чтобы мы не просто вернули ему некий документ, а чтобы мы произвели у себя на сервере некие действия, которые приведут к запоминанию нами его информации. Вот это запоминание, код, который будет производить обработку данных о пользователе, этот код будет содержаться в сервлетах. На фронтенде пользователю мы должны предоставить страницу. Страница, на которой будет форма и в этой форме он сможет заполнить логин, емайл и пароль. Если вы пока не знаете что такое форма, то собственно обратитесь к либо к нашему, либо какому‑то другому источнику информации, посвященной html‑страницам. Форма — это один из стандартных элементов html. Так вот, собственно, на этой странице, которую мы пользователю уже должны отдать. Это может быть, например статическая страница, это может быть индекс html, которую мы ему, пользователю, возвращаем по первому заходу. И на ней он в форму заполняет нужные нам данные и отправляет. Что значит отправляет. Отправляет его в браузер, запрашивает наш сервер на определенном порту поднятый и мы от браузера получаем от пользователя request. И мы на этот request пишем свой сервлет. В этом сервлете (действия происходят на бекенде) получаем объект request. Нам с вами для того, чтобы, то есть мы из этого request сможем извлечь логин, email, пароль. Это будут строковые переменные. И после этого нам с вами нужно эти строковые переменные у себя запомнить. Для того, чтобы работать с этими данными не внутри сервлета, а в некой службе на сервере мы с вами создадим класс аккаунт сервис и у этого аккаунт сервиса будут специальные методы, которые предназначены для регистрации пользователя. То есть мы вызовем внутри сервлета аккаунт сервис, передадим ему параметры, которые получили из request. Внутри аккаунт сервиса мы с вами создадим еще один объект UserProfile. Этот UserProfile будет содержать поля логин, email и пароль. И каждый раз, когда пользователь захочет, каждый новый пользователь захочет зарегистрироваться, мы на него будем создавать UserProfile. И этот UserProfile пусть хранит в специальной карте. Мы создадим Map. Ключ в карте будет логин пользователя, строка. И ключ будет указывать на Value. Это UserProfile.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Регистрация  Авторизация  Frontend  1. Страница с вводом: логин, пароль  Backend  1. Сервлет обработки запроса  2. Класс AccountService с методом авторизации  3. Класс UserProfile в карте по логину  4. Карта HttpSession -> UserProfile |

Следующая страница, которая нам с вами понадобиться это страница авторизации. То есть когда пользователь уже прошел регистрацию и уже вызвал наш сервлет. В этом сервлете мы уже вызвали аккаунт сервис. В аккаунт сервис уже передали данные регистрации и запомнили там его. Запомнили, означает записали UserProfile в карту. Запись UserProfile, логин пользователя на UserProfile — это и есть собственно наш аналог записи в базу. То есть в будущем вы в этом месте мы не будем сохранять его в runtime, в карту логин на профиль, вы будем создавать запись в базе. Пока что у нас там заглушка рантаймовая и мы храним данные пользователя в аккаунт сервисе. Когда пользователь произвел эти действия он приходит к нам с авторизацией, и мы ему даем снова страницу. Опять же это может быть страница результата какого‑то сервлета, может быть просто статический файл. И на нем, на этой странице мы спрашиваем логин и пароль. То есть там тоже форма. На этой форме логин и пароль. Мы этот логин и пароль с пользователя спрашиваем. Он нам с request его запрашивает, отправляет. Мы на бекенде уже пишем еще один сервлет. Я предлагаю вам пока, чтобы не путаться, на каждый такой запрос заводить отдельный класс. Каждый раз, когда я говорю сервлет это означает что это наследник http сервлета, предназначен для работы с конкретным данным запросом. Сервлет обработки запросов в нем тоже будет ссылка на объект аккаунт сервиса, то есть у нас 2 сервлета, оба держат ссылку на один и тот же аккаунт сервис. И в аккаунт сервис мы еще добавим метод авторизации. Мы в этот метод авторизации положим логин и пароль, который достали из request пользователя и найдем в карте по логину UserProfil. То есть у нас смотрите, аккаунт сервис, в нем карта, в карте по ключу логина мы можем спросить UserProfil. Если мы его не нашли, значит профиль еще не зарегистрировался. Мы можем ему вернуть ответ зарегистрируйтесь сначала. Если пользователя мы нашли в этой карте, то мы приветствуем пользователя, говорим здравствуй пользователь и можем взять, например из этой карты еще его email. Написать здравствуй пользователь, мы знаем твой email. Мы поприветствовали его. Кроме того, нам хорошо здесь будет в аккаунт сервисе же хранить карту http сессии на UserProfile. Зачем это нужно, я расскажу четь позже.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Учетная запись  Frontend  1. Страница после логина  Backend  1. Сервлет обработки запроса страницы  2. Класс HttpSession из request  3. Класс UserProfile в карте по HttpSession |

И когда пользователь прошел авторизацию, мы можем показать ему страницу с его учетной записью. То есть это та страница, которую фронтенд покажет после логина пользователя. Jetty уже будет узнавать этого пользователя, она будет знать что это то же самый пользователь, потому что у него в куках будет прописан ключ для поиска http сессии. Мы разберем это позже в теме про http сессию. Пока что мы знаем, что есть такая страница с учетной записью, и на этой странице мы можем поприветствовать пользователя. То есть смотрите, когда пользователь обновляет эту страницу, запрос опять же приходит к нам на сервлет учетной записи, мы проверяем что пользователь залогинен. И мы можем получить http сессию из request. То есть в request Jetty нам заодно будет передавать не просто данные пользователя он нам отправит, но еще и некий объект, представляющий браузер пользователя. И мы сможем найти по http сессии юзерский профиль, то есть мы считаем, что пользователь уже залогинелся и мы его, у него есть соответсвия http сессии на UserProfile. То есть смотрите у зарегистрированных пользователей есть соответствие логин на UserProfile. У залогиненых пользователей есть соответствие http сессии, или http сессии toString дали нам ключ http сессии на юзерский профиль. И соответственно первой проверкой, проверкой если пользователь в карте логин на UserProfile мы узнаем зарегистрирован он или нет, то поиском по http сессии мы узнаем залогинен он или нет. И если мы уже знаем, что он залогинен, у нас есть объект профиля, в этом профиле мы уже сложили 3 поля, которые мы о нем уже знаем. Потом мы в него можем добавлять другую информацию о пользователе, например, результаты предыдущих взаимодействий с системы, время последнего захода, в общем все что хотите.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Выход  Frontend  1. Кнопка выхода  Backend  1. Сервлет обработки запроса  2. Класс HttpSession из request  3. Удаление UserProfile в карте по HttpSession |

После того, как пользователь вошел в систему, авторизовался, посмотрел свой профиль, как‑то по взаимодействовал с системой, мы должны дать возможность еще выйти из системы. И для того, чтобы пользователь имел возможность это сделать нам нужна кнопка выхода на фронтеде. Это тоже будет некая форма, которая отправит запрос на определенный сервлет. То есть нужно сделать на бекенде сервлет, который обработает выход пользователя. И если мы получили от пользователя запрос на выход, то мы вместе с этим запросом получили от него Http сессию. Как в случае, если он пользуется сервисом и ходит по странице. И когда мы по этой http сессии мы можем найти, у нас есть карта, в которой этот пользователь по http сессии может быть найден, может быть найден его профиль. Нам с вами просто нужно будет удалить из этой карты по сессии юзерский профиль, чтобы при следующем запросе мы увидели что пользователь не авторизован. Нам не нужно его удалять из записи логин на UserProfile, потому что в данном случае мы удалим запись о том, что он зарегистрировался. Нам нужно удалить запись о том, что он залогинен. Просто в карте сессия на профиль нужно будет по ключу этой карты запись удалить.

2.1.4 Generics

Теперь когда мы разобрались с задачей, которая нам предстоит, нам нужно разобрать инструменты, которые мы с вами будем использовать. И я уже упоминал использование карт для написания нашего пока еще не очень функционального, но уже интересного сервера и подойти к разбору карт невозможно в Java без разбора дженериков. Эту тему нам с вами сейчас нужно освоить. Тема на самом деле довольно сложная, поэтому в общем может быть с первого раза нужно, чтобы вы ухватили основную идею. Если вы уже использовали другие языки, если вы знакомы с C++, то там есть понятие тимплейтов, шаблонов. Это нечто похожее на то, что у нас есть в Java и то, что мы называем дженериками.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Generic programming — тип переменно как параметр  Проверка типов во время компиляции  Вставка разных типов в коллекцию — ошибка в runtime  Нет необходимости в использовании ключевого слова cast  Универсальные алгоритмы  N алгоритмов, M типов данных. N \* M реализаций?  Алгоритмы работают с шаблонами => N реализаций  Упрощение кода  Меньше кода  Понятнее |

Итак, как разработчики и создатели языков вообще пришли к идет дженериков. Если почитать документацию, выложенную на сайте Oracle, посвященную дженерикам, то вы найдете в первую очередь несколько раз подчеркнутую авторами идею, что дженерики нужны нам для того, чтобы контролировать ошибки во время компиляции, а не во время исполнения. Во‑первых, здесь надо сказать, что дженерики, вся тема дженериков она связана с темой типизации. То есть в Java мы проверяем типы значений, то есть у нас строгая проверка типов. Каждая переменная у нас обязательно должна иметь некий тип и мы должны в случае, если мы пытаемся привести одну переменную к другой и система не понимает, как это сделать, у нас будет ошибка в runtime, то есть никаких скрытых от разработчика способов перевода от типа одного значения в другой тип значения у нас нет. Мы обязаны везде точно сказать какого конкретно типа будет переменная, если у нас идет приведение типов одного в другое, то мы должны сказать каким именно образом будет происходить. Смотрите, если вернуться обратно к дженерикам, то самый простой способ представить себе задумку дженериков это представьте что у вас есть функция, в которую вы передаете параметры. Параметры каждый раз могут быть разными. Дженерики — это способ передачи в качестве параметра типа переменой. То есть если у вас в случае с функцией есть функция и набор значений, которые в функцию вы передаете, то в случае использования дженериков у вас есть функция или класс, в который перед тем, как его использовать у себя в коде, вы должны сказать какого типа вы хотите использовать переменные в этом классе. То есть дженерики — это написание кода таким образом, чтобы тот разработчик, который хочет использовать код с шаблонами должен был перед тем, как начать его использовать, явным образом сказать какого типа будут переменные в коде, который он будет использовать. Приведу пример. Очень важная для нас тема — коллекции. И в Java коллекции в стандартной библиотеке типизированы. То есть вы должны сказать какого именно типа вы будете хранить переменные внутри коллекции. И если вы попытаетесь в коллекцию. Если вы создаете коллекцию без типизации, то есть это такой диприкейтит способ, но пока еще поддерживаемый. Можете создать коллекция без указания типов. И вы можете попытаться положить в него объекты разных типов и при этом вы получите ошибку в runtime. Если вы явным образом скажете какого типа коллекции вы хотите использовать, например, коллекцию строк или коллекцию чисел, то есть если вы создали коллекцию строк, то вы не сможете в эту коллекцию строк положить числа, причем вы поймете, что вы делаете ошибку не на стадии запуска приложения. А в процессе компиляции. То есть дженерики нам, как подчеркивают разработчики языка, в первую очередь нужны для того, чтобы проверять правильность кода, который мы пишем на стадии компиляции. Второе бонус, который нам дают дженерики, это возможность создать неких универсальных алгоритмов. Универсальных алгоритмов и универсальных структур данных. То есть опять же если вернуться к списку или к карте, то так как эти типы данных основаны на дженериках, вы при использовании можете указать какого типа будут эти карты и списки. И вам не нужно писать десять-сотни карт по каждой по карте на комбинацию ключ-значение. Вы можете использовать одну стандартную. То есть нам нужно стандартной библиотеке держать меньше вариантов для структур данных для алгоритмов. Ну и как результат мы получаем, что нам нужно меньше кода написать. Разработчики языка говорят, что код становится понятнее. Я согласен с этим на половину. Код понятнее для того, что освоил тему дженериков. То есть если вы потренировались, потратили некое время на примеры использования дженериков, вы хорошо поймете эту тему и код, который на них основан, если нет, то он для вас будет более загадочным, чем код без дженериков.

2.1.5 Generics. Класс с шаблонным типом

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26. | public class GenericExample<T> {  private T value;  public GenericExample(T value) {  this.value = value;  }  public T getT() {  return value;  }  public static void main(String[] args) {  GenericExample<Integer> intObject =  new  GenericExample<Integer>(1);  Integer valueInteger  = intObject.getT();  GenericExample<String> stringObject =  new GenericExample<String>(  'word');  String valueString =  stringObject.getT();  }  } |

Самое время я думаю нам разобрать пример того, как работать с дженериками. Может быть вам он покажется сложным. Мы наверняка вернемся к этой теме, когда будем разбирать работу с базами данных. То есть я активно буду использовать дженерики как минимум в теме про базы, поэтому к ней хорошо бы эту тему разобрать. Если перейти непосредственно к материалу, то у нас есть 2 способа создания шаблонных, кода, основанного на шаблонах. Это шаблонный класс, то есть дженерик класс. И дженерик метод. На первом примере, который сейчас на слайде, я привел дженерик класс. Первая отличительная по сравнению с обычными классами у этого примера это наличие в угловых скобочках имени типа. В принципе здесь можно использовать какую‑то другую букву. Принято здесь использовать T. Если хотите вы можете задавать сюда свои. То есть вы можете писать GenericExample<W> или <G>. В общем это на ваше усмотрение. Опять же по конвенш, код конвеншн, принято здесь использовать одну заглавную букву. После того как вы в имени класса указали что он использует шаблоны, вы можете указанную в угловых скобках тип, то есть смотрите то что здесь написано в угловых скобках это тип, который нужно задать, если вы хотите использовать GenericExample. Вы можете начать его использовать. Перове самое заметное здесь это использование в качестве типа переменной. Тот, кто захочет использовать ваш GenericExample, он должен будет задать вместо T сказать это будет GenericExample от строки или от числа. И когда он это сделает, то есть, когда скажет что T = String. Как скажет, я покажу в этом же примере. Однозначно будет понятно какого именно типа у вас будет переменная value. То есть Вы уже задали тип, сказали что это будет строка и все, в этот момент вы определили что value у вас это строка. Так же вы можете использовать в качестве параметров функции, тоже использовать T в качестве типа переменной. Здесь у меня приведен конструктор, который запоминает value. И можете использовать его в качестве возвращаемого значения. То есть мы написали класс, класс дескриптор, в котором есть переменная. Вы эту переменную можете создать в конструкторе и потом при желании ее получить. Чтобы не придумывать дугой пример, я здесь же еще в этот самый GenericExample приложил функцию main(), статическую. В Java функцию main() можно поставить куда угодно, можно сделать в Java сколько хотите функций main(), в каких хотите классах, потому что вы указываете какой main() использовать при запуске. Почему бы и не использовать тогда в GenericExample тоже свой собственный main(). И в этом main() я могу использовать собственный же класс. Могу использовать любой другой, могу использовать этот же. И синтаксис вы можете посмотреть создания GenericExample от 2‑х разных типов: от Integer и от String. Вот это задача в угловых скобках типа GenericExample она определят то, какое именно тип данных GenericExample будет использовать. В первом примере я говорю GenericExample от Integer и однозначно определяю T, какой T будет. А во втором случае говорю строка. Ну и тоже однозначно определяю всю дальнейшую работу. И дальше посмотрите на примеры. Я создаю GenericExample из new, указываю какой тип. Если вы используете 7 или 8 Java, то вам можно здесь не писать тип, указать просто 2 угловые скобки, так называемый dimond, придуманный в 8 версии. Компилятор сам поймет, что здесь должен быть Integer, потому что он видит слева Integer. И мы можем передать Integer на вход. После этого мы можем опять же взять через get() взять и получить значение Integer. Точно то же самое можно видеть в примере со строкой, то есть вы передаете строку при создании GenericExample и можете эту строку через get() получить. То есть смотрите, вы написали один класс‑контейнер. Тип контейнера вы задали, то есть сказали это может быть любой, который захочет тот, кто будет этот контейнер использовать. T есть и оно может быть любое. Любое, какое захочет тот, кто будет писать код, используя GenericExample. И вы можете в этом контейнере хранить какие‑то значения. В данном примере Integer или String.

2.1.6 Generics. Метод с шаблонным типом

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class GenericExample {  public static <T> T getTheFirst(  List<T> list) {  Return list.get(0);  }  public static void main(String[] args) {  List<Integer> listOfInts =  new ArrayList<Integer>();  listOfInts.add(0);  Integer intValue =  getTheFirst(listOfInts);  List<String> listOfString =  new ArrayList<String>();  listOfStrings.add(  "Java is the best!");  String stringValue =  getTheFirst(listOfStrings);  }  } |

Кроме варианта создания класса с шаблонами типа можно написать свой собственный метод с шаблонным типом. То есть это вариант, когда вы говорите использовать шаблон, использовать некую переменную типа не для всего класса, а для конкретного метода. Как можете видеть в этом примере я переписал GenericExample. У него больше нет типизации, то есть сам класс больше не требует задания типа для работы. В то же время время мы можем в этом GenericExample написать функции, которые потребуют. Собственно, сами функции внутри себя позволят нам работать с некими заранее не заданными типами, то есть при написании GenericExample функция GenericExample я говорю, что я здесь буду при использовании этого класса использовать некий тип, но я пока задавать его не буду. И чтобы это сказать и в случае с инететом тоже самое относится и к работе с классами в случае если вы это хотите сделать при работе с метолами вы первым делом еще до того как вы начнете работать с сигнатурой метода с возвращаемым значением и с параметрами должны в угловых скобках сказать что это будет метод с параметром. Те из вас, кто в школе может быть помнит решение уравнение с параметром. Вспомните там были квадратные уравнения, часто очень любили их в старших классах задавать, где нужно было решить параметрическое уравнение. Тогда, конечно, все казалось ужасно. После университета я уверен у многих из вас этот ужас пропал. Ничего с ним страшного нет. Параметр — это просто еще одна переменная. Смотрите на эти T просто как на тоже на параметр, только параметр не переменная в runtime, в параметр — это тип во время написания кода. То есть шаблонные типы, шаблонные классы, шаблонные методы — это просто классы и методы, которые перед тем как начать их использовать в коде требует обозначение чему равен параметр. Вы должны задать этот параметр. После этого можете параметр использовать везде внутри функции. Вы можете возвращаемое значение типа T возвращать. Вы можете взять, например, получить на вход List. Смотрите, вот это идет уже использование списка. Здесь я говорю мне на вход в getTheFirst() дайте пожалуйста список от того же типа, который будет возвращать функцию. И после этого я просто у листа беру и получаю нулевой элемент. Код не очень правильно задали конечно, потому что мне сперва надо убедиться что элемент есть. Но для простоты я здесь этого не делаю. То есть если это был продакшн код, я бы конечно спросил еще а если там элемент. В общем сейчас она вернет null, если элемента нет. И когда мы этот код написали, мы можем посмотреть как его использовать. Пример опять же я взял здесь сделал void main(). И в этом main() создаю список, список от Integer, то есть вот этот List это уже не мой класс, это класс из библиотеки. И ArrayList в общем тоже. Эти классы я беру из стандартной библиотеки. Они требуют для создания указания с каким типом данных они будут работать. То есть это список от Integer. И после этого список от Integer я могу добавить элемент. То есть вызвать у объекта listOfInts функцию add() и добавить в нее элемент. И после этого я могу взять и вызвать собственную функцию getTheFirst() и передать в нее ранее созданных Integer. И вернет она мне в код тоже Integer. Потому что она поймет что так как мы передали в функцию лист от Integer, то и вернуть нам надо тоже Integer. И то же самое будет в случае со строками. То есть вы создаете список строк, и здесь компилятор понимает, что надо чтобы была строка. Если написать не строка, а например Integer, то компилятор подсветит всю строку, и скажет что у он не понимает как из строки получить Integer. В этом примере выглядит не очень запутано. Сразу понятно, что список от строк нельзя скастить Integer без специальных условий. А если же у вас более сложный код, где вы этот список передаете между методами, то в какой‑то момент вы можете перестать понимать с каким именно контейнером вы работаете. И тогда в этом месте вам компилятор подскажет. Он скажет вы пытаетесь использовать список с не тем типом данным, который ожидаете. Это бонус от проверки типизации, о котором я собственно говорил в одном из предыдущего шагов одного из занятия.

2.1.7 Generics

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public static Object getFirstValue(  List list) {  return list.get(0);  }  public static String getFirstStringValue(  List<String> listOfStrings) {  return listOfStrings.get(0);  }  public static void main(String[] args) {  List<Integer> listOfNumbers =  new ArrayList<Integer>();  ListOfNumbers.add(42);  String name1 = (String) getFirstValue(  listOfNumbers); // Runtime error  String name2 = getFirestStringValue(  listOfNumber); // Compile error  } |

Для закрепления темы проверки типизации и бонусов от проверки типизации на стадии компиляции рассмотрим еще один пример. В этом примере у нас с вами будет 2 функции. Ясно, что все эти функции, которые здесь привел они наводиться внутри какого‑то класса. То есть в принципе вы можете создать некий класс, назвать его MyClass, скопировать туда эти функции, посмотреть. Благо они все статические и даже дистанцирование объектов не понадобиться. В первой функции мы берем список, но без указания типа. Я уже говорил, что это синтаксически допустимый повсеместно порицаемый способ, который уже деприкейтит. Для поддержки того, что было в Java до 5 версии, до того как появились дженерики. Можно взять и передать лист без указания значений. И можно возвращать тип данных из функции типа Object. Что такое Object я чуть позже скажу. Это общий родитель всех типов в Java. Вы можете просто точно так же как в предыдущем примере getFirstValue() взять из листа вернуть нулевой элемент. И тогда вы в общем то Object можно вернуть из любого контейнера. И второй вариант, это то же самое, та же самая функция, но только она использует список со строками, то есть не просто getValue(), а getStringalue(), потому что мы явным образом указали типизированный чем список передать. И в общем здесь мы то же самое делаем, просто берем и возвращаем нулевое значение. Если вы теперь создадите список чисел и заполните его чем‑то (взяли создали список чисел, указали что будут числа, а не что‑то другое), то если вы начнете использовать для получения из этого списка первого элемента первую функцию без проверки, то есть вы скажете хочу получить первый элемент и по каким‑то соображениям между этими 2-мя строками может быть очень много строк кода. Поэтому вы можете забыть уже, потерять эти знания чем же именно, какой же именно с числами создали объект. И компилятор вам ничего не скажет. Скажет ну ладно, хорошо. Ты разработчик, ты знаешь что делаешь. И в этом месте у вас приложение упадет во время работы. И это не сопоставимо хуже, чем если оно упадет во время компиляции, потому что в сложных приложениях далеко не все участки кода сразу работают в работе, сразу обращаются. То есть выражение может какое‑то время работать, а потом внезапно по какому‑то из запросов произойдет runtime error и у вас приложение свалиться или выдаст пользователю какое‑то странное сообщение, странные ошибки, просто потому что вы не проверили типы при работе с контейнерами. Использование второй функции, которая берет на вход строго список со строками, собственно, вы в него не сможете передать список с числами. Вам компилятор просто скажет, что в этом месте я отказываюсь принимать то, что вы делаете и перепишите пожалуйста что вы написали.

2.1.8 Generics

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | Синтаксис  Map<Integer, String> indexToName =  new HahsMap<Integer, String>();  void printCollection(List<Integer>  collection {…};  void printCollection(List<?> collection) {…};  void drawShape(List<Shape> shapes) {…};  void drawShape(List<? extends Shape>  shapes) {…} |

Самое время нам с вами по дженерикам разобрать примеры синтаксиса. То как вообще можно использовать типизированные классы. Чаще всего вам придется создавать типизированные классы, то есть вы сможете, то есть вы должны будете перед тем как начать их использовать указать тип; перед тем как вы напишите имя переменной вы должны указать тип переменной. Тип переменной у нас с вами List. Лист это дженерик класс, и он хочет строку на вход. Собственно, эту строку передаете в качестве параметра при создании переменой. Вы говорите, что этот список будет от строк. После этого вы должны, в 6 Java явно должны это сказать, в 7‑8 можете просто указать угловые скобки. Потому что компилятор сам понимает какой именно ArrayList создавать, потому что он сможет посмотреть в левую часть создания переменой и проверить какой тип здесь будет. Класс может быть, собственно, как и метод, могут быть типизированы сразу несколькими параметрами. В случае с картой, например. При работой с картами вы должны задать тип ключа и тип значения, на который этот ключ указывает. Чтобы это сделать вы точно так же в угловых скобках вначале пишите тип первой переменой и потом пишите тип второй переменной. И после этого то же самое при использование оператора new вы создаете карту. В данном случае от числа на строку. Второй важный блок синтаксиса, синтаксических правил использований дженериков это передача переменных в функцию. Пишите какую‑то функцию свою собственную. Написали и говорите она на вход будет получать список, типизированный числом целым. И передаете эту коллекцию. После этого в теле функции вы можете этот список использовать, можете быть уверены, что компилятор вам подскажет, если вы пытаетесь в эту коллекцию положить строки или не Integer, а переменные другого типа Long или типа Double. Бывает так, что вам нужно передать в функцию коллекцию, но при этом вы хотите. То есть вам не так важно какого именно типа будут в этой коллекции объекты. В принципе можно было бы, наверное, взять и написать просто (List collection), без указания вообще типа, но как я уже сказал это порицаемая практика. Кроме того если вы передаете коллекцию, вам не важно какое именно содержимое внутри, то нужно каким‑то образом еще поддержать запрет на вставку объекта. И вот чтобы это сделать можно использовать вот такой синтаксический прием — написать знак вопроса внутри угловых скобок. Тогда это будет означать, что в эту функцию можно преносить список от любого типа, но будут некие особенности при работе с ним. Я о них прям на следующих слайдах расскажу. Вы можете передать в функцию список от некого абстрактного типа. В данном случае Shape, некая абстрактная фигура. У нее могут быть наследники. Вы можете создать список от абстрактного типа и подкладывать в него наследников. Но в эту функцию вы, например, не сможете передать List<Treugolnics> несмотря на то, что Treugolnics будут наследовать Shape, передать его туда нельзя, потому что явным образом здесь сказали что здесь должны быть именно фигуры. Если вы хотите передавать в функцию списки или какие‑то другие классы с параметром, которые в качестве параметра используют и базовый класс, и наследников, то вы можете снова использовать знак вопроса. Только сказать что это будет не только любой знак вопроса, который как в предыдущем примере, а что это знак, что это тип, который унаследован от некого базового типа и тогда сюда можно будет в эту функцию drawShape передавать как список от абстрактного типа данных, так и список от каких‑то, список конкретных типов данных.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | Сужение области параметра  public class GenericExample<T extends  Number> {  private T value;  public T getT(){  return value;  }  public static void main(String[] args) {  // Integer extends Number – ok  GenericExample<Integer> intObject =  new GenericExample<  Integer>(1);  Integer valueInteger =  intObject.getT();  // String does not extends Number –  // error  GenericExample<String> stringObject =  new GenericExammple<  String>('word');  String valueString =  stringObject.getT();  }  } |

Если вернуться, если продолжить разбор последнего примера, то мы можем захотеть не просто передавать в функцию переменные, в которых шаблонный тип указан как базовые классы наследники, мы можем захотеть еще создавать собственные классы и собственные методы, типизированные неким суженным набором параметров. Не всеми типами, а только неким нужным нам. Например, как в этом примере только теми типами, которые унаследуются от класса Number, то есть в Java есть библиотеки класса Number, от него унаследован Integer, Long, Float, Double. И вы можете сказать, что мой GenericExample будет работать с числами. А с каким конкретным числом мне не так важно. Но чтобы сказать, что именно с числом, вы должны будете при создании класса сказать, что вот ваше T то самое, которое задает тип, оно будет унаследовать Number. В данном случае я забегу еще вперед. Не важно является ли Number классом или интерфейсом. Просто запомните себе, когда мы дойдем до интерфейсов, попытайтесь вспомнить. Слово extends в данном случае означает просто наследование. И не важно от чего именно. Если мы посмотрим на то как использовать, то мы можем использовать наш GenericExample для Integer, и не можем использовать для строк, например. Вместо Integer можно написать, GenericExample может быть от Long или от Double, или от Float. И с числами будет работать хорошо, а с другими какими‑то типами работать уже не будет, и вы на стадии компиляции получите здесь ошибку.

2.1.9 Generics

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | Специфическое использование  Использование без типизации — deprecated  List  Можно передавать List<Object>, ArrayList<Integer>, LinkedList<String> ...  Использование Object в качестве параметра  List<Object>  Можно передавать только коллекции от Object  Произвольный тип в качестве параметра  List<?>  Можно передавать любые коллекции, но вставка невозможна |

Давайте еще разберем варианты использования типов, шаблонных типов, но случай если нам нужно от них, ну если нам нужно не просто указать какой‑то конкретный тип, а нужно от них чуть больше. Самый базовый что ли подход здесь и в общем то опять же, я как уже говорил, самый неудачный — это использовать список или какой‑то другой шаблонны тип без параметров, как в первом примере. То есть компилятор вас ругать не будет, разве что скажет ворнинг что это диприкейтит метод, что так делать не рекомендую, но вы можете взять и создавать такие листы и передавать их, писать вот этот тип без указания шаблонного типа в качестве параметра переменной метода. И тогда вы в этот метод можете передавать или этой переменной переменой присваивать лист от чего угодно. И вставлять в него что угодно. Получается некий такой контейнер всего, но неудачный контейнер. Можно взять и написать лист от Object, второй вариант. И окажется тогда, что может быть в функцию передавать любой список и присваивать, допустим, я беру List<Object> list и могу присвоить List от чего угодно. На самом деле это не так. Присваивать можно только лист от Object. И передавать в функция можно тоже только List от Object. Если вы хотите некий универсальный List, то есть функцию, которая принимает лист от всего чего угодно и еще может вставлять все что угодно, то вы должны написать функцию, в которую вы должны передать List<? extends Object>. Вот если вы такую функцию сделаете, то в нее можно будет передать любой лист, из него можно будет получать элементы, вставлять элементы. Единственное что при получении элементов вы получите ссылку на тип Object, то есть в общем то вы потеряете всю информацию о конкретных типах, которые вы в нее передали. Если у вас задача передать все что угодно, но при этом все что угодно может быть вполне конкретное, то есть допустим представим, что это у нас функция, которая берет List<?> list. И мы хотим в нее передать, например, List от Integer и List от строк, List от Object. Но мы в случае вот такого синтаксиса запрещаем внутри функции вставку чего бы то ни было в этот список, потому что внутри функции мы потеряли информацию о том, что содержит список. И в отличии от предыдущего примера, когда у нас был лист от Object. Список, который нам может прийти в эту функцию, внутрь, он может быть действительно не от Object, а от чего‑то более высокоуровневого. И так как мы информацию вставки потеряли, о типах потеряли, то и вставка теперь у нас не возможна. Мы таким образом запрещаем вставку, потому что мы не контролируем больше что именно в этом объекте лежит, что лежит в списке, поэтому не можем проверить, компилятор не сможет проверить правильно ли вы делаете вставку. И поэтому в этом месте компиляция у вас не пройдет. В том месте, где вы попытаетесь вставить в этот List, то есть напишите list.add(""), то компилятор скажет, что он работать с этим не будет.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Wildcards  List<? extends Number>  Можно передавать List<Integer>, ArrayList<Long>, LinkedList<Number> ...  List<? super Integer>  Можно передавать List<Integer>, ArrayList<Number>, LinkedList<Object> ... |

И раз уж я стал говорить о этих знаках вопроса и использования слов extends надо просто сказать еще пару слов что это такое. Это так называемые Wildcards. Они позволяют вам либо сказать, что я хочу использовать список от наследников, либо от родителей. В общем виде использование Wildcards, кроме простых способов, когда мы просто говорим List от чего угодно, мы можем сказать, что мы хотим передавать функцию список от наследников Number. Тогда мы сможем передавать Integer, Long, Number, Double, Float, все наследники. Либо мы можем, то есть нам в некоторых случаях при разработке может понадобиться передавать список от всех родителей. В таком случае мы можем использовать слово super и тогда в функцию, которую на вход берет этот лист мы сможем передавать тот же Integer, его родитель Number и общий родитель для всех классов Object. Заодно посмотрите здесь я еще сказал, вернее не случайно вам написал что здесь разные типы самих списков, то есть если я говорю что у меня функцию передается Liat, то я в функцию могу так же передавать объекты, которые наследуют List. Например, ArrayList или LinkedList.

2.1.10 Generics

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | generics vs inheritance  Наследование  принцип «разновидность чего то» (is a)  Шаблон  принцип «специализируется на» (of something)  Пример  class Ветеринар<T extends Животное> extends  Человек  T: Слон, Собака, Мышь  В случае T: Человек, получаем ветеринара  по людям — врача |

В заключении рассказа о дженериках, здесь я хотел бы обратиться к несколько абстрактной теме. Уже, к счастью, философско-мировозренческой. У нас есть 2 механизма указания общего между разными объектами. Смотрите, если у нас объектно-ориентированное программирование, то в основе объектно-ориентированного программирования лежит определение, понятие абстракции. То есть вы говорите вот есть некий набор объектов, который можно собрать, обозначить неким общим словом. Как в случае, у нас есть животные, есть разные типы животных кошки, собаки, мыши. И мы можем сказать животное — это то, что их всех объединяет. То, о чем я вам до этого рассказывал только что дженерики, они в общем то тоже от части об этом. Мы говорим о дженериках, что вот есть некий контейнер, который может содержать какие угодно объекты. То есть получается, что он тоже представляет собой нечто общее, как бы выделенную общую часть, то есть мы можем выделенную общую часть обозначить у себя архитектурно задать как бы наследование, а можем как дженерики. И вот здесь есть некоторая путаница в головах, когда что использовать. При проектировании своих систем вы можете использовать или тот или другой подход. И чтобы понять, что именно вам нужно, нужно использовать некий признаки тех, о которых я хочу сейчас рассказать, те которые есть на слайде. То есть для того, чтобы понять что вам нужно использовать наследование, у вас объекты должны быть разновидностью чего‑то общего, то есть вот например животные является разновидностью абстрактного понятия животного или млекопитающее. Или конкретные марки автомобилей можно сказать что это разновидности автомобиля. То есть в английском это признак, его называют is a, то есть является чем‑то. То есть кошка является животным, Земля является планетой. Если вы видите у себя, что между общей частью и конкретной частью зависимость именно этого вида, то вам нужно использовать наследование. А если у вас общая часть является специалистом по чему‑то, то есть она работает с чем‑то, в английском of something. То есть ящик апельсинов. В данном случае ящик будет типом шаблонным, в котором апельсины — это дженерик. У вас может быть ящик апельсинов, ящик картошки, ящик яблок. Мы можем взять и написать, допустим, понятие абстрактный ящик и наследовать от абстрактного ящика уже конкретный ящик апельсинов задать, прям вот явно задать написать класс, который будет называться ящик апельсинов или ящик картошки. Это будет попытка решить с помощью наследования той задачи, которая может хорошо решена шаблонами. То есть если у вас объект чего‑то, например, список строк, карта ключей строк на http сессии. Как‑то так. Тогда здесь нужны шаблоны. Вот пример, который мы сочинили на лекциях. И он мне очень нравиться. Допустим, у нас есть ветеринар. Вы должны понимать, что у нас что это за синтаксис такой у нас здесь с вами значит шаблонный тип, который extends нечто абстрактное. И сам этот класс он тоже у нас расширяет некое понятие. То есть смотрите в данном случае здесь у нас есть и наследование, и шаблоны. Класс ветеринар, ветеринар — это человек, то есть он является человеком (is a man). И поэтому мы можем использовать extends здесь, то есть это наследование — ветеринар является человеком. Можно даже сказать промежуточное состояние сделать: ветеринар является специалистом, специалист является человеком. И при этом этот наш ветеринар он может специализироваться на каких‑то животных. Так же как ящик может быть ящик апельсинов, ветеринар может быть специалистом по собакам или специалистом по кошкам, то есть мы можем задавать наше T и перечислять от чего специалистом является наш ветеринар. И вот в данном случае интересно будет, если мы в качестве T выберем человека, потому что человек у нас очевидно тоже, я немножко нарушу синтаксис, так нельзя писать, но чтобы понятнее было class Ветеринар extends Человек extends Животное. То мы можем в качестве T подставить сюда человека при создании ветеринара. И тогда мы получим такого особенного ветеринара, специалиста по людям, то есть врача. И в заключении уже про шаблоны, важный совет для тех, кто собирается писать по настоящему большой сервер хочу сделать сразу. То есть видите, у нас занятие только началось и сервер мы особенно кода не писали, а вот уже вам совет готовьтесь сразу. Где еще хорошо использовать дженерики. Я вам рекомендую все сущности у себя в системе нумеровать. Например, у вас есть пользователь, у него есть id, у пользователя есть сумка, у сумки есть id, у вас бегают по игровой карте животные, у них тоже должен быть id. И у самой карты тоже должен быть id. У всего, чего угодно должен быть идентификатор. Обычно это идентификатор записи в базе. И вот когда у вас много сущностей разных типов и у всех у них есть идентификатор, вам в какой‑то момент может быть создать функцию, которая будет получать на вход вот эти самые идентификаторы. То есть она возьмет userId, возьмет serverId, какой‑то address и еще там что‑нибудь. И если в качестве идентификатора использовать просто число, в моем примере это long, то рано или поздно вы ошибетесь в порядке этих перечислений long, то есть компилятор не скажет вам никогда, что мы взяли и поменяли местами два long при вызове функции. Или когда вы взяли и допустим поменяли сигнатуру функции и то же что‑нибудь перепутали. И у вас все будет компилироваться и очень странные ошибки у вас из‑за этого будут. Чтобы этого не было я вам предлагаю сделать у себя класс LongId, сделать его шаблонным и в качестве типа передавать класс id которого вы хотите в нем хранить. То есть например, у вас будет LongId<UserProvile>. Или когда появится сервер id от сервера или допустим от класс сервер, или от адреса. И тогда вы сможете переписать свою функцию вызова с явным указанием того, какие именно id вы ожидаете. И здесь уже компилятор вам не позволит поменять их местами. Это очень полезно, когда у вас будет больше сотни классов. Или когда вы разрабатываете относительно долго. В любом случае это просто полезно, потому что помогает вам в коде написать меньше ошибок.

2.2 Коллекции и карты

2.2.1 Коллекции и карты

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Iterator G Iterable  interface Iterator<T>  boolean hasNext()  T next()  void remove()  interface Iterable<T>  iterator<T> iterator() |

После успешного знакомства с дженериками давайте перейдем к коллекциям, к картам, которые на основе этих дженериков построены. И первое с чего здесь надо в этой теме начать — это наследование. То есть как построена иерархия классов коллекций в самом базовом. На самом базовом уровне у нас лежат 2 понятия. Это Iterator и Iterable. Iterator — это то, что вы можете получить для любой коллекции. То есть все коллекции реализуют интерфейс Iterator. Чуть забегу вперед и скажу, что интерфейсы это у нас такой класс специальный, в котором мы перечисляем список методов, которые обязаны реализовать все, кто этот интерфейс реализует. Это некое подобие, то есть абстрактный класс C++ чисто виртуальными методами. Внутри интерфейса Iterator 3 метода. Я здесь перечислил все. Мы можем спросить если следующий элемент, мы можем брать элемент. У вас есть коллекция, вы берете Iterator и помощью Iterator вы можете проверить есть ли следующий элемент, если он есть — начать по нему двигаться. Вы видите, что Iterator здесь у нас шаблонный. То есть на самом базовом уровне, коллекции уже содержать шаблоны. И когда вы спрашиваете next(), то вы уже спрашиваете элемент того типа, который вы задали при создании этой коллекции. Собственно, это Iterator. Чтобы коллекция имела возможность вернуть итератор, она должна реализовывать интерфейс Iterable, то есть все коллекции в Java реализуют интерфейс Iterable. И он тоже типизирован. И он как раз возвращает Iterator нужного нам типа.

Картинка

Так выглядит иерархия контейнеров. Здесь еще идут у нас зависимость на предыдущий слайд, на Iterable. То есть после Iterable идет интерфейс коллекции. И от коллекции идет 3 интерфейса, 3 базовых интерфейса, представляющих некую абстракцию, абстрактные типы данных. Это Set, List и Queue. Если вы с абстрактными типами данных не знакомы, можете обратиться хотя бы к википедии, прочитать про каждый их них. То есть библиотечная реализация абстрактных типов данных. Про карту мы пока не говорим. Про карту чуть позже отдельно скажу. И дальше каждый из этих интерфейсов реализует уже конкретную реализацию этого абстрактного типа данных, написанная разработчиками библиотеки. Для Set — HashSet, LinkedHashSet и TreeSet. Разница в том, что HashSet хранит хеши объектов; LinkedHashSet кроме того, что хранит хеши, хранит еще порядок добавления элементов; TreeSet — это Set, построенный на дереве. В случае List ArrayList — это обертка над массивом, в котором просто лежат в массиве ссылки на элементы, которые мы храним в списке. В случае с Vector — это уже в каком‑то смысле дипрекейтит объект. Для наших нужд пока что, будем считать что это то же самое что ArrayList. Никто в Java его уже не использует. Все использую ArrayList. О тонкостях я сейчас не буду рассказывать. И есть еще LinkedList. Но это собственно связанный список. В Java он 2-связвнный, то есть каждый элемент содержит ссылку на следующий и на предыдущий. Если и тот и другой есть. И есть еще, кроме того, такой как очередь приоритет. Карта стоит немножко отдельно от всего от этого. То есть она не наследует коллекцию, не наследует Iterable. Сама по себе карта — это абстракция, абстрактный тип данных. И абстракция выражена в Java интерфейсом. У нее есть несколько реализаций. Hashtable синхронная карта. Я не буду сейчас про особенности. Если вам интересны особенности Hashtable. Опять же мы ее использовать не будем, прочитаете. LinkedHashMap и HashMap мы будем с вами использовать. HashMap это карта на основе хешей. LinkedHashMap также как LinkedHashSet содержит порядок добавления ключей. И TreeMap — это собственно карта, которая основана на дереве. Кроме того, что вы должны знать какие бывают типы, хорошо бы если бы вы еще знали про особенности этих типов по отношению к операциям, которые с этими типами можно производить. Например, сложность вставки, сложность извлечений элемента вставка в конец, в начала или в середину. К этой теме будут еще отдельные вопросы, посвященные не просто реализации этих типов данных в Java, а вообще принцип работы с ними.

2.2.2 Коллекции

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Collection  Extends Iterable<T>  Методы  add(T object)  addAll(Collection<T> coll)  clear()  contains(Object o)  remove(Object o)  removeAll(Collection<T> coll)  size()  isEmpty() |

Чуть подробнее о коллекциях. Абстракция, от которой унаследованы все конкретные коллекции. Как я уже говорил, она сама по себе эта коллекция наследует интерфейс Iterable. Здесь по терминологии может быть стоит сказать. Если конкретный класс расширяет другой класс, мы используем слово extends. Если какой‑то конкретный класс реализует интерфейс мы используем слово implements. То есть класс implements какой‑то интерфейс. Если же интерфейс расширяет другой интерфейс, то мы можем снова использовать слово extends как здесь и происходит. То есть у нас и Collection, и Iterable — это интерфейсы. И мы можем сказать, что Collection extends Iterable. То есть один интерфейс расширяет другой интерфейс. В данном случае расширяет означает добавляет еще элементов в него. Важно сейчас рассмотреть Collection, потому что все коллекции какие есть в Java, все листы, все сеты, все содержат один и тот же набор методов. То есть универсальный набор моментов, который они унаследовали от коллекции. В первую очередь это добавить объект и добавить все объекты. Вы можете взять и в список или в Set добавить элемент. Потом вы можете добавить все элементы из другой коллекции. И это очень удобно, в том смысле что например вы можете взять и выдать addAll() у Set и добавить в него все списки из List. Или, наоборот, в List все из коллекции, из какой‑то любой другой коллекции, из чего угодно, из другого Set или из другого списка. Любую коллекцию можно почистить, у любой можно спросить содержит он элемент или нет. Но здесь надо быть осторожным. Если вы у какой‑то абстрактной, если вы не знаете что там у вас списки или Set, и какие списки. Если вы у них спрашиваете содержит ли коллекция какой‑то элемент, но вы не знаете конкретный тип, вы можете оказаться в неудобной ситуации, когда у вас поиск становится линейным от размера коллекции. Например, в случае с поиском элемента в List. То есть в случае с HashSet у вас поиск будет константный. Если мы возьмем HashSet, то время поиска будет константное. Если у нас окажется TreeSet, то время поиска будет логарифмическое. А если мы возьмем в качестве контейнера список, то у нас поиск время — поиску длине, то есть при вызове contains() нужно четко понимать что вы делаете и использовать тот тип данных, который нужен. То же самое относится к remove(). Перед тем как убрать элемент, нужно найти элемент. И здесь важно отличать не только Set от List, а еще знать какой именно List вы используете, потому что в случае убирания элемента из List ArrayList приведет к тому, что вам придется пересоздавать List. То есть вы потратите машинное время на то, чтобы создавать еще один элемент. В случае removeAll() из коллекции, здесь вообще сложно время посчитать, потому что нужно убрать все элементы, которые присутствуют в другой коллекции. Дальше вы можете у любой коллекции спросить размер и спросить пустая она или нет. Здесь есть некое недопонимание зачем нужна пустая или нет, если есть size(). Казалось бы можно написать if (size == 0) и вроде как это должно быть то же самое, что if (isEmpty()). На самом деле это не так. Потому что isEmpty() гораздо более дешевая операция, чем size(). Кроме того size() для коллекций, работающих в многопоточных средах, он еще может быть и не определен сразу. Его придется считать. То есть, например, для списков, основанных на деревьях, предназначены для работы в многопоточных условиях, вычисления size() может занять довольно приличное время. А если вам на самом деле надо спросить Empty или не Empty, то проверить пусто не пусто однозначно всегда и довольно быстро. И раз уж мы говорим с вами про коллекции, нужно еще сказать про внутреннее устройство основных реализаций коллекций. Точнее здесь даже не столько реализации конкретно в Java, сколько тоже абстракциях, которые расширяют коллекции, то есть коллекции от коллекции вообще просто нечто, что содержит некий набор элементов. А чтобы понимать как именно будет происходить работа с этими элементами нужно дополнить понятии коллекции некой дополнительной информацией. И вот собственно дополнение это происходит через указания одного их 3‑х типов разновидностей коллекций, то есть первое это списки. Список отличается от остальных коллекций, например, от Set тем, что у нас в списки есть очередность элементов. То есть если вы добавляете элементы в список, то элементы могут повторяться и вы точно знаете в каком порядке вы добавляли. В случае уже конкретной реализации ArrayList вы не просто знаете в каком порядке добавляли, а еще у каждого элемента есть свой индекс. По этому индексу можно быстро к элементу обратиться. В случае со списком, который основан на указания элементов друг на друга. В Java это называется LinkedList, в этом случае вы можете в этот элемент быстро добавлять и удалять элементы, потому что вам просто нужно переставлять ссылки, поменять ссылки и все. Всю остальную структуру менять не нужно. То есть 2 реализации как я уже сказал LinkedList, ArrayList, самые часто используемые. LinkedList быстро удаляет добавляет элементы, ArrayList быстро получает доступ к индексам. Быстро в данном случае означает за константное время. То есть вы просто взяли и обратились. В Set нет никакой очередности, нет никакого индекса. Это просто корзина, в которой лежат элементы. И основной бонус Set в том, что вы можете искать в нем элемент. Быстро в случае HashSet означает за константное время, в случае TreeSet означает за логарифмическое время. Бонус Set, основанный на деревьях в том, что его удобно использовать в случае когда система меняется. То есть у вас есть поток информации, вам нудно его упорядочить, вы скорее будите использовать TreeSet, чем HashSet, потому что время вставки элемента в TreeSet тоже логарифмическое. Есть еще очередь FIFO. Я думаю за деталями вы можете обратиться к уже специальной литературе. Мы будем использовать очереди, и для нас просто важно, чтобы бы был некий контейнер, в который можно положить элементы и потом извлечь в обратном порядке удобным образом.

2.2.3 Карты

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Map<key, value>  Быстрый поиск по ключу — get(key)  Объект в качестве value. Например другой контейнер (Map<Integer, List<T>>)  Set<keyType> keySet()  Collection<valueType> values() |

Картинка

Кроме коллекций у нас есть еще карты. Либо их также называют еще дишенери, либо … Есть некий холевар о том называть ли map картой. Мне нравится слово карта. Оно мне кажется более естественным чем говорить Map. Поэтому я у себя буду использовать карту. Если вам больше нравиться какое‑то другое название просто транслируйте на привычной себе. Пользователи шарпа называют его дикшенери. От того как мы будем называть сущность не меняется. Эта структура данных, которая представляет нам удобную работу с парой элементов ключ‑значение. Ключ — это то, по чему мы хотим находить значение. То есть карта — это структура данных, в которую мы закладываем сразу пару элементов. Ключи должны быть уникальны. Значение могут быть не уникальны. По ключу мы можем находить значение. Причем желательно как‑то быстро находить значение. Что такое быстро и как это устроено я в рамках этого занятия хочу рассказать как устроена, общий обзор, как устроен HashMap. Первое что нужно понимать это что такое хеш. Хеш — это число, которое вы ставите в соответствии объектам из вашего множества. То есть у вас есть множество ваших объектов. И вы этому множеству ставите в соответствии множество чисел. В Java это множество чисел типа Integer. То есть мы берем объекты, строки, другие числа, какие‑то более сложные объекты, содержащие смесь этих строк и чисел. И ставите ему в соответствии некое число. Число из диапазона Integer. Желательно, чтобы у хеш функций, которые вы используете были, чтобы она как можно больше использовала чисел для разных объектов. Вы взяли некое множество ваших объектов, ему сопоставили некое подмножество в множестве целых чисел типа Integer. Чем функция, хеш функция лучше, чем большее подмножество в Integer оно займет. Кроме того, у хеш функций должно быть еще, хорошая хеш функция разным элементам рядом составит в соответствии разные числа, желательно очень удаленные друг от друга. Чем больше не просто подмножество, а чем больше это подмножество чисел по охвату чисел. Естественно, что хеш функция должна всегда одному и тому же объекту ставить соответствие одно и то же число. Не должна зависеть от каких‑то внешних факторов. У нее есть объект и для него мы вычисляем хеш функцию. Она всегда остается одинаковой. Но при этом вы должны понимать, что для разных объектов хеш функция может дать одно и то же число. Самый простой пример для понимания как это может быть, представьте что у вас есть числа типа long. И числам типа Long нужно прочитать хеш. И хеш в Java это Integer. То есть мы явно сильно сужаем область чисел. У нас множество Long гораздо больше, чем множество Integer. Тем не менее мы должны иметь возможность вычислять. То есть для некоторого довольно существенного числа, чисел типа Long, хеш будет один и тот же. Когда у нас эта хеш функция есть, мы можем для всех наших объектов эту функцию вычислять некое число. В хеш карте по этому числу остатком от деления на число buckets. То есть в каждой карте есть некий набор buckets. Проще всего представить buckets, что это такое, это массив Integer. В данном случае здесь у меня, например, массив из 256 элементов. То есть мы при создании карты создали карту, в которой внутри есть массив, массив 256 элементов. Если я хочу вставить в карту элемент, я вычисляю для него хеш. Получаю некий Integer. После этого я этот хеш остатком от деления на размер массива байтов нормализую к индексу этого buckets. Например, в случае Джона Смита у нас здесь на картинке получился элемент 152. И я в эту ячейку записываю ссылку на value, которую хочу хранить. Вставка объекта в хеш означает вычисления хеша, нормализация на число бакетов и записывания в массив ссылок. Для разных ключей мы выселяем их хеш и складываем ссылки. Так как у нас во‑первых множество Integer обычно меньше чем множество объектов, да и заодно мы еще нормализацию производим из остатка деления на количество buckets, у нас возможны коллизии. То есть коллизии по хешам это означает, что у разных объектов совпали хеши. А еще заодно может быть просто коллизия после нормализации. Есть 2 способа как ее решить. Можно внутри одного элемента buckets хранить список. То есть один объект, а несколько. Либо можно при появлении такой коллизии записывать новые объекты в следующую ячейку, потом в следующую и так далее. И заодно в картах есть еще механизм перестроения себя в том случае, когда коллизий стало слишком много, фактор этого перестроения. Что значит слишком много, вы можете задать вручную, можете использовать стандартный. По‑моему стандартный это 0,7. Вы вставляете элементы. В какой‑то момент понимает, что надо себя перестроить. И она берет и, например, увеличивает размер buckets с 256 до 512, то есть в 2 раза себя увеличила и перестроила соответственно, переноризовала все записи. Собственно, бонус какой от использования этой структуры данных. У вас поиск элемента по ключу в случае, если коллизий не много, занимает константное время. То есть что значит константное. У вас есть ключ, для ключа вы можете за константное время посчитать хеш, нормализовать на количество buckets и после этого так как это все массив, buckets — это массив, вы можете, зная результат, получившегося от остатка деления, это не что иное как индекс элемента в этом массиве. И обращение к индексу в массиве — это тоже константное время. То есть получается, что вы за константное время посчитали хеш, нормализовали и за  время нашли по индексу нужный вам элемент. Размер карты может быть миллион элементов. Но при этом вы всегда за константное время, не зависящее от размера карты, можете найти value, которое этому ключу соответствует.

2.2.4 Коллекции

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Comparable  interface Comparable<T>  int compareTo(T o)  Возвращаемое значение типа int может быть >,  =, < нуля  Integer a = 1;  Integer b = 2;  a.compareTo(b) == -1; // true  a.compareTo(a) == 0; // true  b.compareTo(a) == 1; // true |

Вернемся к коллекциям и рассмотрим утилиты, которые позволяют нам с коллекциями удобно работать. Первое с чего хочется начать здесь, это интерфейс Comparable. Я уже говорил, что по сравнению с C++ в Java нельзя переопределять операторы. То есть вы не можете взять и переопределить у себя поведение операторов больше, меньше или равно. Однако же сравнение для собственных типов данных нам нужно, то есть у нас есть задачи, в которых нам нужно сравнить элементы друг с другом. Причем не в стандартной заданной библиотеке, а наши собственные элементы. И вот для того, чтобы решить эту задачу в стандартной библиотеке есть интерфейс Comparable. Он тоже типизирован. Вы наследуете от него свой тип данных и этот же тип данных передаете в качестве параметра ему. И в этом интерфейсе у нас есть единственная одна только функция — это compareTo(). То есть все типы данных, которые реализует интерфейс Comparable обязаны у себя задать функцию compareTo(), которая на вход берет то, с чем мы будем сравнивать и на выход возвращает Integer. Возвращаемое значение это Integer оно может быть у нас больше, меньше, равно нуля. Естественно для любого числа. Собственно, когда вы, что значить все, кто реализует интерфейс Comparable, должны у себя задать. То есть вы пишите у себя тип данных собственный, например MyNumbers, и хотите их или там вектора вы хотите сравнивать, например, по длине. То вы говорите, что они у вас реализуют интерфейс Comparable и задаете у себя своем типе данных функцию comareTo(), задаете в ней тело функции. И в этом теле функции вы должны вернуть int. Вы должны вернуть int. Если вы хотите сказать, что тот элемент, который вы сравниваете меньше, чем тот элемент, с которым вы сравниваете, то возвращаете отрицательное число. В самом простом случае -1. Естественно, что если они у вас равны, то вы должны вернуть 0. И если вы хотите сказать, что то число, которое вы сравниваете с тем которое вы сравниваете больше первое чем второе, то вы должны вернуть положительное число. В самом простом случае единицу. То есть вот здесь вот вы должны написать, в моем случае здесь будет просто return a < b ? -1: 1, пишите проверку на равно до return и вот похожую запись. И тогда вы сможете свои типы данных сравнивать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Class Collections  Collections — набор статических методов для работы с контейнерами  Основные методы  void copy(List dest, List src)  Object max(Collection coll)  Object min(Collection coll)  void reverse(List list)  void shuffle(List list)  void sort(List list)  void swap(List list, int i, int j) |

Кроме возможности сравнивать собственные типы данных друг с другом есть еще очень много возможности, работая коллекциями. В том числе для чего мы этот Comparable смотрели, это, например, если вам нужно просортировать элементы вашей коллекции. То есть если у вас есть коллекция, список, коллекция с порядком. Внутри вы хотите по этой коллекции взять и просоритровать элементы в ней, то вам нужно эти элементы сравнивать. Чтобы эти элементы сравнить, вам нужно скачать, что они сравнимы друг с другом. Кроме сортировки есть в стандартной библиотеке еще некий набор возможностей. И весь этот набор возможностей собран в специальный класс, класс с названым Collections. На мой взгляд название не самое удачное. Я бы его назвал CollectionHelper. Ну потому что, собственно, это класс, содержащий статические методы, не требующие дистанцирования этого класса, который позволяет вам что‑то сделать с коллекциями. Скопировать одну коллекцию в другую или найти максимальную, найти максимальную и минимальную вам тоже понадобиться, чтобы коллекция была  просто от типов, а от типов, сравнимых друг с другом. Можно взять и поменять порядок местами начало с концом, то есть перевернуть вашу коллекцию. Можно перемешать коллекцию. Просортировать. Кроме того вида сортировки, есть еще сортировка, когда вы передаете как сортировать, если у вас объекты не реализуют этот Comparable и если вам нужна кастмная какая‑то сортировка, неестественная, то есть вы можете например захотеть просортировать сроки, но в афлафитном порядке по началу строки, а по длине строки, то вам нужно будет еще, есть возможность здесь же в классе Collections задать кастомную сортировку. Ну и кроме того здесь есть важный еще момент. Мы еще встретим его. Можно в коллекции взять и поменять местами две позиции. То есть задать первую, вторую и поменять их местами.

2.2.5 HttpSession

Давайте от технических деталей реализации коллекций, карт вернемся к нашей задаче. Теперь, когда вы знаете что такое коллекции, карты и что такое дженерики, вам будет проще понять как; вы можете сейчас вернуться к нашей задаче и еще раз посмотреть видео с задачей и уже лучше понимать какие именно, как именно нам нужно организовать аккаунт сервис наш, который будет у нас отвечать за хранение информации о регистрации, авторизации пользователей. И в нем, в этом аккаунт сервисе у нас с вами будет 2 карты. Первая карта будет логин пользователя на UserPrifile. Вы уже знаете, что такое карта, что такое ключ и как по ключу происходит поиск элемента. И вторая карта, которая нам с вами понадобиться, это карта http сессия на UserPrifile. Это карта, по которой мы будем мы узнавать авторизовался ли пользователь. Если первая нужна для того, чтобы понять зарегистрирован ли пользователь является нашей некой заглушкой для базы данных, то вторая уже вполне себе боевая сущность, которая у нас вполне останется прямо до конца работы, она хранит информацию о том, пользователь, который запрашивает у нас что­‑то, авторизован или нет. Запрашивает что‑то это означает, что обращается к одному из наших сервлетов и в этот сервлет передает свой запрос, то есть к нам сервлет придет request. И в request заодно есть еще такая сущность как http сессия. Это картинка к этому слайду она иллюстрирует работу с куками, то есть Jetty, когда работа с пользователем, она о себе информацию у пользователя в браузере записывает в куки. Когда пользователь запрашивает что‑то у веб сервера, в нашем случае запрос попадает в Jetty вместе с запросом пользователь заодно еще отправляет значение из куки и в куки, то есть по‑русски печенки. И в этой куки, в этой печенке записана информация http сессии, то есть Jetty сама по значению, которое присылает пользователь находит http сессию и нам в сервлет эту http сессию передает.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | HttpSession session = request.getSession();  Long userId = (Long) session.getAttribute(  'userId');  if (userId == null) {  userId = userIdGenerator  .getAndIncrement();  session.setAttribute('userId', userId);  }  String key = session.toString(); |

Вот собственно пример выглядит следующим образом. Вот этот код может быть может находиться внутри сервлета. Внутрь сервлета у нас попадает request. Если вы посмотрите на сигнатуру функцию doGet или doPost() внутри сервелета, то вы увидите, что попадает request и у этого request можно взять http сессию. Вот эта сессия. В сессию можно взять и сохранить что‑то свое, то есть вы можете спросить, а если в сессии, то есть в сессии внутри есть аналог карты, ассоциативный массив ключ‑значение. Вы можете спросит у нее атрибут. Если его нет, то вы можете этот атрибут задать. А если он есть, то вы можете взять и в качестве ключа в карте аккаунт сервиса, в которой мы пишем залогинен пользователь или нет, использовать, например, вот этот userId (Long). Либо если вы не хотите писать значения в сессию, вы можете взять у сессии строку, то есть превратить сессию toString(), вызвать у нее toString(), получить ключ сессии. Я не проверял, но скорей всего он будет совпадать со значением в куки. И у себя в карте хранить уже запись строка на userProfile. То есть в первом варианте у нас получится в аккаунт сервисе Map<Long, userPrifile>, а во втором случае у нас получится Map<String, userProfile>. Вариант какой выберете — на ваше усмотрение. Главное просто чтобы по запросу пользователя, по его request, можно было однозначно понять если для него профиль у нас в карте или нет для него профиля. Если нет, надо попросить его залогиниться.

2.3 «Углубленные» основы языка

2.3.1 Наследование

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public abstract class Animal {  private int weight;  public Animal(int weight) {  this.weight = weight;  }  public boolean isDangerous() {  return isPredator() || getWeight()  > 15;  }  protected int getWeight() {  return weight;  }  abstract boolean isPredator();  }  public interface Jumpable {  boolean canJamp();  } |

2 UML‑диаграммы

Давайте продолжим нашу подготовку к выполнению первой серьезной задачи и разберем некий блок базовых вопросов. Я постараюсь их рассказывать не просто как базовые, а чуть глубже чем обычно принято. И начнем мы с вами с наследования. В общих словах наследование важный элемент объектно-ориентированного программирования и естественно, что в Java не могли обойти этот вопрос и добавили в язык возможность классов наследовать друг друга. На этом слайде я привел пример класса и интерфейса. Класс у меня абстрактный, то есть я не могу создавать объект этого класса. И интерфейс он, он как бы всегда сам по себе абстрактный. И здесь же попутно я привел еще UML‑диаграмму, которая этому классу и этому интерфейсу соответствует. Я думаю, что серьезных объяснений здесь делать не нужно. Если вы совсем не знакомы с Java посмотрите на важные ключевые слова, которые здесь есть. Это public. public говорит о том, что класс можно использовать в любом пакете, где хотите в вашем приложении если на модуль, в котором он лежит есть зависимость. Этот класс абстрактный. Как я уже сказал это означает, что его непосредственно нельзя создать, объекты этого класса нельзя создать. Можно спокойно создавать объекты от классов, которые будут наследовать этот класс. Само ключевое слово class. Всего таких ключевых слов, выполняющих похожую роль в Java три. Это класс, интерфейс и enum. Ну и название. В нашем классе есть одно поле. И это поле приватное. В данном случае я решил создать абстрактный класс для некоторого животного абстрактного и задать в нем вес животного. Потом в этом классе есть конструктор, то есть функция, которая ничего не возвращает, то есть у нее нет возвращаемого значения. И я в этот конструктор передаю параметр. Здесь тоже все понятно. Дальше у функции есть 3 метода. И начну я снизу, с абстрактного метода. Абстрактный метод означает, что этот метод должны обязательно задать все наследники. Класс может быть абстрактным и не иметь ни одного абстрактного метода, но если вы задаете абстрактный метод, то класс вы тоже должны задать как абстрактный. И есть 2 функции, отличающиеся областью видимости. public функции, protected функции. public доступна всем тем, у кого есть ссылка на объект. protected доступна только наследникам и еще тем, кто лежит вместе с этим классом в том же пакетом. По содержимому я думаю здесь все более‑менее понятно. И здесь же на слайде есть еще интерфейс. Это тоже public интерфейс, то есть его можно создавать, ссылки этого типа можно создавать. И в этом интерфейсе у меня всего одна функция. То есть я буду явно, понятно, что я буду использовать животных и Jumplable одновременно, буду проверять будут у меня животные прыгать или нет. Посмотрите на UML‑диаграмму. Как принято обозначать приватные поля я уже показал. Как принято обозначать public поля и protected. Абстрактные методы принято писать курсивом. И вот кстати для интерфейса тоже у нас будет запись курсивом. Полей никаких у нас в интерфейсе нет.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class Elefant extends Animal  implements Jumpable {  private boolean canJump = false;  public Elefant() {  super(200);  }  public boolean isPerdator() {  return false;  }  public boolean canJump() {  return canJump;  }  } |

UML‑диаграмма с 2 связями

И посмотрим, как выглядит класс, который использует рассмотренные ранние интерфейсы и абстрактные классы. Мы можем создать вполне конкретный класс для конкретного животного. Я выбрал слоника. И говорю, что этот слоник он у меня расширяет животное, то есть расширяет в том смысле, что добавляет особенности. Здесь есть в головах некоторая путаница что такое расширение и что такое сужение. То есть, с одной стороны, перейти от слона к абстракции это вроде как расширение в области применения, а с другой стороны. На мой взгляд, как я понимаю эти термины. Слон расширяет возможность, расширяет признаки животного. Животное — это нечто безличное, сбор неких признаки всех животных, которые вообще есть. А слоник наш он уже вполне конкретное животное. И он добавляет свои характеристики к описанию. То есть описание абстрактного животного, то есть если вы зададитесь целью выписать признаки абстрактного животного. Вы какое количество их напишите. Если вы захотите написать признаки слоника, то вы напишите гораздо больше признаков. В том числе признаки животного тоже войдут в него. Вот это и есть у нас наследование, когда вы наследуете признаки. Кроме того, здесь я уже сказал, что он у меня будет имплементить интерфейс Jumpable. В принципе, любое животное может имплементить в той или иной степени. И это означает в моем случае, что этот класс обязан еще задать метод, который задан в Jumpable. То есть смотрите у меня абстрактный класс с абстрактным методом. Абстрактный метод я обязан задать. И здесь же у нас интерфейс. И интерфейс я тоже обязан, в интерфейсе есть метод, который я тоже обязан определить. Это я и делаю. Я задаю абстрактную функцию и обязан задать реализацию метода интерфейса. И смотрите еще внимательно, здесь есть еще конструктор. И в этом конструкторе я написал ключевое слово super(). super это отсылка к конструктору базового класса. Отсылка к базовому классу обязательно должна идти первой в строке конструктора, то есть я не могу определить сначала какие‑то методы. Здесь вызвать или задать переменные, а потом позвать super(). Нет, не могу. Я должен super() обязательно 1-ым вызвать. Я создал такого небольшого слоника, весом 200, сказал что он не хищник. И сказал, что он прыгать не умеет. Потому что я возвращаю runJump, а runJump я задал при создании объекта. Посмотрите, есть такая запись — инициализация переменной при написании переменной внутри класса. Это означает, что переменная будет заполнена в момент создания объекта. То есть, когда вы объект создаете, вы создадите и эту переменную тоже. Ну и UNL‑диграммы, я не стал здесь перечислять методы и переменные, то есть вот сюда надо по хорошему дописать, сюда переменные и … Можете посмотреть как они выглядят и сами написать, если вам интересно. Посмотрите здесь интересный вид ссылок. Это стрелка зависимости, обозначающая наследование, соответствует от слову extends. А пунктирная — это стрелка, обозначающая наследования типа implements. Мы таким образом можем создавать любые наследники от животного, можем создать кроме слоника тут мышку, кошку, кого угодно. И у них будет примерно похожая диаграмма.

2.3.2 Наследование

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | public static void main(String[] args) {  // Animal animal = new Animal();  Elefant elefant = new Elefant();  boolean isDangerous =  elefant.isDangerous();  if(elefant.canJump())  jump(elefant);  Jumpable mouse;  mouse = new Mouse;  // boolean isMouseDangerous =  mouse.isDangerous();  jump(mouse);  }  public static void jump(Jumpable object) {  // Animal animal = object;  // boolean isDangerous =  // object.isDagerous(); - ошибка  if(obgect.canJump()) {...}  } |

Теперь, когда мы разобрались с вами с абстрактным классом, с интерфейсом, с наследником от класса от интерфейса и с UML‑диаграмам, которые их представляют, можем посмотреть пример. На этом слайде я вывесил пример как может выглядеть использование тех классов, которые я до этого вам показал. И действия все здесь будут происходить в функции main(). Про функцию main() мы с вам уже говорили. Я думаю, что вы довольно быстро запомните сигнатуру вообще, как эта функция выглядит. Многократно будите еще встречать. В этой функции я привожу примеры того, как можно, и как нельзя делать. Как можно у меня код раскомментирован, а как нельзя я его закомментировал и сейчас объясню почему я его закомментировал. Если его раскомментировать, он компилироваться не будет. Первым делом здесь первой же строкой закомментировал создание объекта типа Animal. Потому что это попытка создать объект абстрактного класса. Создавать его нельзя. Поэтому если мы раскомментируем, то нам компилятор скажет, что он не будет его создавать. А как правильно создавать. Правильно сказать new Elefant(). Здесь при желании вы можете написать ссылку не типа Elefant, как здесь, а например типа Animal. Либо же ссылку типа Jumpable. Попробуйте у себя сами этот вариант подставьте сюда. Другие типы, типы родительские или интерфейсы. И посмотрите, эта строка компилироваться будет. Следующие может уже и не будут компилироваться. А это будет работать отлично. То есть вы можете спокойно присвоить новосозданный объект в ссылку записать типа родительского класса или типа интерфейса, любого из интерфейсов, который он реализует. Дальше на строке я показываю как можно с этим объектом работать. У нашего слоника есть функция isDangerous(), причем эта функция она public и задана в Animal. Так как она public и задана у всех животных, мы у слоника тоже можем вызвать. И мы можем у нее спросить может он прыгать или нет. Потому что canJump() это функция из интерфейса, которая она реализует. Она все функции из интерфейсов, они тоже public, то есть мы спокойно их можно вызвать. Мы можем взять и нашего слоника передать в другую функцию. Причем посмотрите, мы передали его в функцию, которая на вход принимает Jumpable. То есть в тот момент, когда мы исполняем код. Мы его исполняем сверху вниз. В какой‑то момент мы оказываемся в 18 строке, потому что вызываем функцию Jumpable. Правда у слоника мы не позовем, потому что прыгать он не умеет, но если бы умел, позвали бы. Компилятору на самом деле все равно, он же логику не проверяет. Поэтому так или иначе, для какого‑то другого животного мы можем сюда его передать и мы здесь окажемся. И когда мы здесь оказались с вами внутри функции Jump, мы уже забыли о том, что это слоник. Забыли в том смысле, что объект остался точно такой же. В памяти где‑то лежит объект, который мы создали. Мы его создали, он в памяти лежит. Мы на него поставили ссылку типа Elefant. То есть смотрите, здесь как бы с 2‑х сторон надо смотреть. У вас есть объект и есть ссылка на него. И, вообще говоря, тип объекта и тип ссылки они могут не совпадать. Тип объекта у вас будет тот, который вы задали при конструировании, то есть вы его пишите new что‑то там, то вы создаете собственно тип объекта. А тип ссылки может быть, он может совпадать, то есть вы можете взять буквально тот тип, который создаете, а можно взять в качестве типа ссылки любого родителя или любой интерфейс, который этот класс создаваемого объекта наследует или реализует. И если вы передаете в функцию объект по ссылке, то в Java вы создаете в стеке новую ссылку, то есть вот в этот момент при вызове jump() мы перейдем в функцию. Чтобы перейти в функцию, мы создадим на наш объект нашего слоника еще одну ссылку. Это название переменной новой ссылки. Вы вызываете функцию, создаете новую ссылку. И создаете ссылку опять же так же, как при создании любого из типов родителей или любого из типа интерфейсов. В данном случае Jumpable. И как я уже сказал мы потеряли знание о том, что там слоник. Мы знаем только здесь в этой функции только то, что он реализует интерфейс Jumpable и все. Если мы с вами создадим класс Робот абстрактный и будем от него наследовать конкретные виды Роботов и решим, что они тоже умеют прыгать, то мы в эту функцию jump() сможем передать не только наших животных, а любого робота тоже. И сюда же можно передавать мячик, потому что он умеет прыгать. При этом что он слабое отношение имеет к животным и к роботам. То есть нам здесь не важно что еще умеет объект. Что он вообще собой представляет. Нам совершенно все равно. Нам важно, что он реализует интерфейс Jumpable. Мы по этому интерфейсу и передаем. Кроме этого интерфейса мы о нем больше ничего не знаем. То есть каст к Animal он не возможен. Я закомментировал эту строку. Потому что мы не знаем Animal это или нет. Это может быть что‑то другое. Это может быть тот же робот. И здесь мы не можем вызвать у этого объекта isDangerous(), потому что isDangerous() это функция Animal. В Jumpable есть только одна единственная функция canJump() и все. Больше ничего нет. То есть мы потеряли данные о том, что мы передали. Объект остался точно таким же. Объект не менялся. Вот какой он был слоник, как в памяти лежал, так и лежит. Просто мы на него поставили другую ссылку. Здесь пример с мышкой. Мышка прыгать умеет. В принципе здесь все то же самое. Единственное что посмотрите я мышку сразу задал Jumpable. Поэтому опять же ызвать у нее isDangerous() я не смогу, потому что создал ссылку интерфейса и потерял. Тому, кто использует эту ссылку он уже не знает что там конкретно за объект. Он знает только то, что ему сказали, что он прыгать умеет.

Код и диаграмма с 4 элементами

Ну и в заключении я еще раз хочу подчеркнуть, что в Java можно наследовать один класс и можно реализовывать сколько угодно интерфейсов. Мы можем кроме Jumpable добавить еще Movable и таким образом показать все это на UML‑диаграмме.

2.3.3 Object, Class

Продолжим разбор наших углубленных основ языка. И когда мы с вами поговорили про классы. Про классы как вы их видите в момент написания кода. Приведенная там UNL‑диаграмма подтверждение тому, то есть это представление о классах программиста в момент кодирования и в момент проектирования системы. Кроме того, классы и объекты в Java это не просто абстракция, это сущности, которые существуют в runtime тоже. То есть допустим, если в C++ вы работали с кодом, написали классы, зависимости проставили, потом собрали и в собранном виде потеряли все что у вас было в ваших текстах, классах и наследованиях, то в Java это не так. И не так благодаря сущности с названием ClassLoader.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Classloader — часть JVM, которая загружает данные о классах  Все классы должны быть загружены при старте JVM  При старте JVM работают следующие загрузчики:  - Bootstrap class loader (<JAVA\_HOME>/jre/lib);  - Extensions class loader (<JAVA\_HOME>/jre/lib/ext);  - System class loader (CLASSPATH) |

Забегу вперед, скажу что она нам нужна для такого понятия как рефлекшн. И мы с ним будем встречаться в течении всех занятий, поэтому так как по тихонько я буду вас подводить к этой мысли и сейчас пришел момент разбора ClassLoader, понятия класса и объекта. Значит, смотрите, когда вы запускаете свое приложение вы байт‑код, который и является этим приложением, передаете в Java‑машину. И Java‑машина начинает этот ваш код исполнят. Перед тем как она приступит к исполнению, Java‑машина проверяет что все там байт‑кодом консистентно. Кроме того, Java получает доступ к списку классов, которые вы создали, то есть вашим классом. В добавок у Java‑машины есть стандартная библиотека, которой ваше приложение будет пользоваться. Это тоже байт‑код и тоже со своими классами. И Java‑машине перед тем, как начать работать с вашим приложением нужно в память загрузить данные о всех этих классах. То есть смотрите есть стандартная библиотека, есть в ней какие‑то классы. Есть ваш собственный код. В нем есть ваши классы. Есть еще фетпати библиотеки, которые вы используете. Там тоже есть какие‑то классы. Так устроена Java‑машина, что она перед тем как начать исполнять ваше приложение должна в память загрузить данные обо всех этих классах. На лекциях я здесь делаю паузу и спрашиваю студентов: если бы вы проектировали систему, проектировали Java‑машину и Java как бы вы сделали загрузку классов. Что такое загрузка классов. Что такое классы. Вы можете у себя поставить на пауза и подумать над вопросом. Я вам просто сейчас сразу и отвечу. То, в виде чего мы загружаем в память класса этот объект, то есть смотрите в Java‑машине и Java есть понятие класса и есть понятие объекта. И вот когда мы загружаем в память информацию о классах, мы эту информацию храним в виде неких объектов. На следующем слайде я постараюсь объяснить это место подробнее. Пока что про ClassLoader слайд было не сложно прочитать. Он состоит из нескольких загрузчиков, они работают друг за другом, по очереди загружают все эти данные, которые нам нужны.

Картинка

Данные, которые нам нужны, они у нас лежат в виде объектов в памяти. Если у нас есть объекты, которые представляют классы, то естественно предположить, что это не просто какие‑то непонятные объекты, что‑то в памяти, что это объекты для которых тоже есть класс. То есть вы когда написали свой класс, вы его зачем написали. Вы написали затем, чтобы в runtime создавать объекты этого класса, то есть у нас не может быть объекта, если этому объекту не соответствует никакой класс. И точно так же объектам, который представляют классы, должен соответствовать некий класс. Это довольно сложная часть лекции. Сейчас давайте я постараюсь привести пример из реальной жизни. Потом сделаем еще отсылку к древним грекам. Может быть получится объяснить понятнее. Первое как мне нравиться объяснять класс и объект это вот загрузка классов на старте, загружаемую информацию мы организуем в объекты, объекты у нас в памяти лежат и если есть объекты, должен быть класс. Это первое. Второе с какой стороны можно зайти к пониманию того, что такое java.lang.Class. Я сюда его выписал. java.lang.Class — это класс, объекты которого описывают классы из библиотеки ваши собственные и фитпати. Здесь я привел некую попытку пояснительной диаграммы. Смотрите, нам сейчас с вами интересно правые 2 верхняя и нижняя ячейки. Когда вы пишите свой код. Представьте, что у вас задание написать код про самолеты. То есть в вашем приложение должны создаваться объекты, в процессе работы вы должны из создавать эти объекты самолетов. И как‑то они должны летать в приложении. Перове что вы делаете естественно — это пишите классы обоим, взяли и создали класс. В нем перечислили переменные, которые могут быть у самолетов, может быть сделали какое‑нибудь наследование, не о нем сейчас речь. Представьте, что есть один класс. И в классе есть поля, методы, работающие с самолетом и окружением. Потом в runtime вы создаете непосредственно самолет. То есть это стадия написания кода, то есть кодирование, это runtime. В runtime вы создаете уже вполне конкретный самолет. То есть у вас был абстрактный самолет, а потом вы создали конкретный самолет. Тем из вас, кто знаком с классической философией это прямая отсылка к Платону и его идеям. Airplane — это идея самолета, а все конкретные объекты, которые вы инстанцировали через new — это вполне конкретные объекты. Мы не будем там как греки вдаваться в подробности что красиво, что божественно. Нет, просто конкретный самолет. Теперь перейдем к левой части схемы и здесь у нас есть класс с названием java.lang.Class. Это такой же класс, как классы plane. То есть это некая абстракция. Ее написал кто‑то из разработчиков. То есть это текст, код, написанный в виде текста. Разработчиков Java я имел в виду. И она тоже существует во время кодинга, когда вы пишите код. Когда вы перезапускаете приложение, ClassLoder загружает в память объекты. Объекты класса Class. Вот оставшиеся ячейка показывает нам это место. Это уже реальные объекты в runtime. В них записана информация о том, какой класс они представляют. Если все еще не понятно. Я думаю, мы еще когда дойдем до Reflection, к этому месту еще вернемся.

2.3.4 Object, Class

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | java.lang.Class — объект, который представляет в runtime данные о классе объекта  Основные методы  Static Class<T> forName(String className)  String getCanonocalName()  Fields[] getField(String name)  Class[] getInterface()  Method[] getMethods()  Constructor[] getConstructors() |

Теперь, когда мы разобрались с вами с классом, загрузкой классов на старте, несколько слов о том, зачем нам вообще все это было делать. В runtime Java‑приложениях есть возможность у объектов спросить класс. То есть у вас есть объект, вы говорите ему getClass и получаете возможность посмотреть структуру того класса, объект которого вы имеете. То есть вы получаете объект, то есть вы говорите ваш объект object.getClass() и получаете некий объект на выход. Объект класс То, о чем я говорил до этого. И когда вы его получили, вы можете с ним некие действия произвести. И здесь я выписал основные методы. На самом деле здесь их может быть пятая часть, может быть десятая часть от общего количества методов. Там их очень много. Если вам интересно просто наберите в поисковике java class и вы получите одну из ссылок, ссылку на туториал от оракла или даже прям на исходник самого класса, или на документацию по исходнику. Посмотрите весь список. Здесь я выписал те, которые мне кажутся самыми интересными. И первая из них — это статическая функция, которая позволяет получить класс по имени. То есть у вас есть строка, строка в которой вы пишите полное имя класса. Полное имя — это то, которое содержит еще и путь к классу. Например, здесь у меня сидит сверху в названии слайда написанное полное имя классу, которое называется Class. Если у вас есть строка с полным именем класса, то вы можете статическим методом класса Class получить объект этого класса. Давайте вернемся чуть в перед, к предыдущей теме, ClassLoader. Представьте, что у вас приложение уже загружено и уже начинает работать. И вы в это приложение еще взяли и передали config. config — это набор строк. И среди этих строк есть название класса. Полный путь классу. Вы можете в runtime по этому названию поискать среди загруженных Java‑машиной классов, те самые объекты, которые были загружены на стадии загрузки классов, среди них можно найти по имени объект, который этому классу соответствует и получить его. Пока что остановимся на том, что мы можем так сделать. У самого этого объекта, когда мы уже получили, мы можем спросить имя. Он нам вернет то самое имя, по которому мы его искали, если искали. Можете не искать по имени класса. Можете просто если у вас уже есть объект спросить у него getClass(). Об этом чуть позже. Мы можем получить полный список полей этого класса. То есть в runtime получить список полей класса, объект которого вы имеете, или полное имя, которое вы знаете. Полный список интерфейсов, методов и много чего интересного и самое главное интересно здесь получить список его конструкторов. И по этому списку конструкторов вы можете вы брать какой‑то из конструкторов, который вам по каким‑то причинам нравится и создать объект. То есть я говорил, что объекты в Java можно создавать только через вызов оператора new, и я там сделал еще оговорку что не всегда, что есть еще другая возможность. Вот это вот как раз та самая другая возможность. То есть получив каким‑либо образом объект класса. Например, статических вызовом через передачу полного имени класса или через вызов по gerClass() объекта, имея объект класса вы можете создавать объект этого класса дальше у себя в runtime. Зачем все это надо? Пока что ограничусь тем, что скажу что я не зря упомянул config. Вы можете в config написать пути к классам. Например, списком пути класса. А в runtime вы можете взять и объекты этого класса создать. То есть вы, меняя config, можете поменять объекты, которые вы будете создавать в приложении. Вот это одно из больших применений класса Class.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | class Object — класс от которого унаследованы все остальные классы  class MyClass{...} == class MyClass extends Object{...}  void myFunction(Object varName) — может «обработать» любой объект  Основные методы класса Object  public Class<?> getClass()  public String toString()  public boolean equals(Object obj)  public int hashCode()  protected Object clone() |

Из вот этой темы нам с вами осталось еще разобрать класс Object. С классом Object это пороше, потому что класс Object звучит понятнее, чем класс Class. Хотя Object Objec тоже звучит не очень хорошо. Я не представляю, как это читать на английском. К счастью, на русском есть еще возможность просклонять это. Ну вот, класс Object. Второй важный класс. Ну их 2 самых важных класса: класс Class и класс Object вообще в библиотеке Java. Класс Object — это класс, от которого унаследовано вообще все. То есть полностью писать порядок наследования, то в конце иерархии любого класса будет Object. То есть мы extends Object пропускаем, не пишем. Мы можем любой объект, просто сделать ссылку типа Object и эту ссылку поставить на любой объект вообще, на какой угодно. Единственное, на что нельзя ставить ссылку класс типа Object это примитивные типы. Их там не так много, всего 8 штук, мы до них дойдем, поэтому считаем, что практически на все можно поставить. То есть если вы хотите в функцию передать что угодно, то вы можете сказать функция принимает на вход Object. Бонус от этого решения в том, что у класса Object есть некий набор методов, и так как это родитель всех классов, то эти методы есть вообще у всех объектов. Какой бы вы объект не взяли у себя в приложении в нем обязательно будет getClass(), тот самый, о котором я говорил. Его можно привести к строке, его можно сравнить с другим объектом, у него можно взять hashCode(). Вот это очень интересное место. То есть Object, самый базовый класс в языке имеет еще возможность посчитать по нему хешкод. Кроме того есть еще protected clone(), позволяющий вам, если вы хотите, чтобы объекты вашего класса можно было клонировать вы можете раскрыть этот клон на public и клонировать свои объекты. Методов у Object больше, чем я написал, не сильно больше, примерно еще столько же, но они нужны для работы с потоками, и мы пока с вами разбирать не будем.

2.3.5 Примитивные типы

Давайте подробно рассмотрим еще одну важную тему, тему примитивных типов. Обычно эту тему поднимают, когда начинают изучать язык, какие‑то базовые знания по языку передают. Мне кажется, что к ней, кроме того что в базовой части тоже надо рассказывать, к ней нужно вернуться и в более углубленной части. Потому что, мне кажется, сложно переоценить важность понимания базовых типов. Потому что какое приложение вы не взяли, у вас в приложении, если разобраться в том, что лежит в основе любого приложения, то у вас будут ссылки и примитивные типы, больше ничего. … отбросить то, как мы пакуем данные, то есть в общем‑то основы объектно-ориентированного программирования вы собираете все в классы и создавать объекты, это некая запаковка примитивных типов и ссылок внутрь структур, удобных человеку для понимания. Когда вы начинаете изучать ООП, обычно начинают с того, что человек мыслит классами объектами и вы при проектировании своей системы должны выделить в ней классы и потом создать объекты этих классов и посмотреть как эти объекты будут взаимодействовать. Это все придумано просто для того, чтобы удобно запаковать, структурировать данные, представленные примитивными типами.

|  |  |  |
| --- | --- | --- |
| bits | type | type |
| 8  16  32  64 | byte  short  int  long | boolean  char  float  double |

В Java примитивных типов 8. Запомнить из очень удобно по этой таблице, потому что они отличаются количеством бит, отведенным на тип. И количество бит увеличивается вдвое при переходе от одной группы типов к другой. Мне кажется, что запомнить это нужно обязательно. Всех своих студентов я на сдаче этой части задания обязательно эту таблицу спрашиваю. Потому что нужно, мое мнение, нужно знать типы данных, с которыми вы работаете. Кроме того, как дополнительное задание я вам могу задать разобраться с пределами с максимально допустимыми значениями, которое есть у каждого из этих типов. Очевидно, что оно зависит от количество памяти, отведенное. Что в long можно записать больше данных чем в int, но вот на сколько больше и можно ли в int записать миллиард, а 10 миллиардов. И в long то же самое. И с типами с плавающей точкой, с ними вообще отдельная тема. Там нужно разбираться сколько у вас значащих чисел может быть в типах float и double. И нужно хорошо себе представлять какие арифметические операции вы можете без потери точности с этим типом данных проводить. Самый простой пример это 0.1 + 0.2 не даст 0.3 в случае, если вы даете с числами с плавающей точкой. Здесь в этой таблице есть еще 16‑ти битные short и char. short используют не так часто, хотя в тех случаях, когда очень важно размер передаваемых данных, например, если вы по сети что‑то передаете, а у вас есть некий id передаваемых данных, можно для него использовать short. chat лежит в основе любых строк. И заметьте в Java 16‑ти битный, то есть Unicode 16.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | Обертки простых типов  boolean, byte, char, short, int, float, long, double  Примитивные типы  мало памяти  простая структура  Stack  Boolean, Byte, Character, Short, Integer, Long, Double  Обертки простых типов  Наследники от Object  Сложные типы  Heap |

Если дальше разбирать тему примитивных типов, то кроме самих этих 8 примитивных типов у нас на каждый из примитивных типов есть еще соответствующий ему класс, то есть я их называю обертками. Честно говоря, не совсем устоявшийся термин, мне он нравится. То есть у нас есть примитивные типы 8 штук и на каждом из них соответствие в виде некоторого класса. В том виде, как я их здесь выписал на этом слайде. Нужно это за тем, чтобы если вы, то есть иногда нужно использовать одни, иногда другие. Бонусы от примитивных типов. Первый основной бонус то, что памяти выделяется на стеки под него, памяти выделяется мало, гораздо меньше, чем под объект. И примитивный тип — просто ячейка в памяти и все, то есть, например, вы решили создать int, это означает что вы выделили 32-бита памяти и дали ссылку на первый, точнее даже у вас переменная, которую вы завели, она соответствует этой ячейке памяти. Здесь даже о ссылках не стоит говорить. Важно здесь отметить еще для тех из вас, кто работал с C++ в Java разрядность примитивных типов задана жесткой спецификацией, то есть не зависимо от платформы реальной, на которой вы будете работать, у вас Integer будет всегда 32 бита и long всегда 64. И в все остальное все точно так же останется, потому что это разрядность виртуальной машины. А как потом вот эту разрядность виртуальной машины перекладывать в настоящую разрядность реальной машины это уже проблема писателей виртуальной машины. Но они нам должны гарантировать, что вот эта вот разрядность будет сохранена. Если вернуться к оберткам, то иногда бывает полезно использовать не примитивный тип, а то, что над ним, класс, который для него создан. Во‑первых все эти обертки наследники от Object, то есть вы создаете объект. Естественно, что каждый из этих оберток внутри лежит примитивный тип. Но вы можете пользоваться теми бонусами, которые есть у всех объектов. Например, просто позвать метод объекта, о которых я до этого говорил. И собственно эти объекты вы будете выделять еще в хиппи, то есть память под этот объект будет выделена в хиппе, вы должны держать на него ссылку, если вы ссылку потеряли, этот объект будет не сразу удален, а будет удален по пришествию некоторого времени, когда его сборщик мусора удалит. И кроме того, если вернуться к теме с контейнерами, то примитивные типы не могут лежать в контейнерах, то есть чтобы создать список или карту, или Set, вы должны в качестве типа шаблона в эту структуру данных, которую вы хотите использовать, передавать именно обертку. Примитивные типы нельзя там использовать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Boxing  Character a = 'a';  Unboxing  int a = 1 + new Integer(2); |

Часто вот в этот момент при разборе Boxing и Unboxing меня студенты спрашивают про перегрузку операторов. Я еще раз скажу перегрузки операторов в Java нет. Вы не можете перегрузить оператор плюс, равно, меньше, больше. Есть некая локальная перегрузка оператора равно для примитивных типов и для оберток над ними. Каст одного в другое происходит автоматически, то есть если вы напишите ссылку типа char и присвоите ей char от примитивного типа, то компилятор спокойно совершенно превратит одно в другое. Вам никаких специальных действий производить не нужно. Точно то же самое произойдет наоборот, когда вы создадите объект и вот в этот момент, когда будет произведен этот плюс, произойдет каст одного в другое. Прямое и обратное действие называется boxing и unboxing.

2.3.6 Массивы и строки

Теперь, когда мы разобрали примитивные типы, между всем тем множеством прекрасных классов и объектов этих классов и примитивными типами есть еще один промежуточный слой. Некая группировка примитивных типов, у которой обычно принято использовать тоже повсеместно. Фактически это тоже в каком‑то смысле примитивные типы, но чуть более сложные, чем примитивные типы. Какая‑то некая Собственно, это массивы и строки.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Массив — это объект (наследник от Object)  Массив хранит свой размер: (new int[100]).length;  Переменная может быть размером массива  int[] arrayInts = new int[100];  String[] arrayStrings = new String[100];  int[] arrauInts = {1, 1, 2, 3, 5};  java.lang.Arrays — класс для работы с массивами |

Массивы в Java — это объекты. Чтобы создать массив. Я здесь привел примеры. Вы пишите тип, который должен быть у массива. Это может быть примитивный тип, это может быть класс. Потом вы указываете скобки квадратные, пишите название и через new создаете новый массив, то есть это именно создание нового объекта, то есть выделение памяти в хиппе. Так как это объект, вы у массива можете позвать любую из функций объекта. Можете взять toString() сделать у него. Он вам у себя пропишет, все элементы свои пропишет. Либо можете у него позвать getClass. Он вам класс, представляющий описатель для класса массива. В Java массив хранить свой размер. И это очень удобно, потому что мы не должны полагаться на нулевой последний элемент или еще на что‑нибудь, как в C++ передавать вместе с массивом еще заодно и размер массива. И при создании элемента через new вы можете динамически сказать какой размер он будет. То есть вот new и сказать переменную внутри квадратных скобок. То есть в квадратные скобки после new можно при желании поставить переменную, вычисленную раннее. И что такое массив в Java? Это некий заголовок, потому что это объект. Чуть позже про заголовок. И потом упакованный один за другим аккуратненько либо примитивные типы, либо ссылки. То есть вы говорите. Когда я говорю создать объект примитивного типа, я просто делаю заголовок и потом в память закладываю одну за другим элементы, которые хочу. То есть в случае с Integer по 32 бита на каждый будет аккуратненько уложено. Если Long, там 64 будет. Если я создаю массив не примитивных типов, в ссылочных типов, то есть каких‑то классов, то у меня будет так же заголовок в памяти, а потом аккуратненько одна за другой лежать ссылки. Размер ссылок у нас будет зависеть от разрядности системы, то есть там будет либо 32, либо 64 бита на ссылку. Раз я уже сказал про разрядность системы. Когда вы себе выкачиваете JDK или JRE, вы явным образом выкачиваете под определенную разрядность, то есть это не какой‑то процесс, когда Java‑машина принимает себе решение. Это именно ваше решение в тот момент под какую платформу вы пишите, то есть под сколько разрядную машину выкачиваете JDK или JRE. В этот момент вы принимаете решение о разрядности ваших ссылок внутри машины. Кроме возможности создания массива через оператор new есть еще упрощенная запись. Пример на слайде. Взяли и в фигурных скобках перечислили, что должно лежать в массиве. Это то, что касается массивов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | char — 16 bit (UTF 16)  String — обертка над char[]  Immutable: "abc" + "bcb" — создание новой строки  System.out.println("abc");  String cde = "cde";  System.out.println("abc" + cde);  String c = "abc".substring(2, 3);  String d = cde.substring(1, 2); |

Строки — это некая обертка над массивом char. char у нас 16‑ти битный, то есть в Java у нам все строки 16‑ти битные. Особенности появляются в тот момент, когда вы пытаетесь строки стерилизовать. То есть, например, записать в файл или передать по сети. Там запись может произойти не в UTF‑16, а в UTF‑8. Это чуть более сложная информация. Если вам интересно, вами найдите описатель строк. Обертка над массивом chart внутри каждой строки, какой‑то массив char лежит. Даже если вы создаете пустую строку, вы все равно создаете объект, в нем будет heder, в этом объекте будет ссылка на массив char, только вот он будет сам объект char лежать, у него тоже будет heder, но он будет пустой. Сстроки в Java имьютбл. Это означает, что если вы один раз строки, и пытаетесь с ней какие‑то манипуляции действия, например, соединить ее с другой строкой, то изменение самих первых строк не произойдет. У вас при манипуляциях над строкой будет создан новая строка. Есть особенность при создании новой строки на основе старой строки, поиск подстроки. У них может оказаться общий массив char. То есть строка новая, а массив char тот же. Но тем не менее сам этот массив char меняться не будет, то есть когда вы создаете строку, создаете объект для строки, вы создаете массив char, записываете в него, собственно, char, который стоят в строке и потом менять его уже не можете. Здесь на этом слайде я еще привел примеры того, как можно работать со строками, что вообще с нами можно делать. И по строкам и все. Я уже начал некий такой вводный про следующую тему, про размеры. В следующей части мы ответим на вопрос, когда я создаю пустую строку, какой же размер строки я создаю, сколько памяти я потрачу.

2.3.7 Size of Object

Заключительная тема, которую я хочу в этом уроке затронуть, это размер объектов, который мы создаем. Сразу скажу, что в Java с контролем выделения памяти плохо. Собственно, как и с контролем чистки памяти. То есть по задумке разработчик вообще не должен думать о том, что там в памяти происходит. По задумке. Тем не менее желание у разработчиков знать, когда мы создаем новый объект, выделяем для него память, сколько же памяти там на самом деле будет потрачено, потому что мы в реальном мире живем, не в мире виртуальной машины и у нас количество памяти ограничено. И вот этот вопрос он волнует многих. И когда я задался этим вопросом, стал искать статьи на эту тему, понял на сколько все плохо, когда нашел статью, в которой разбирали стенд, то есть приложение, которое делало следующее. Ты ему давал тип данных, мы с дефолтовым конструктором. Приложение это создавало миллион объектов, записывала их в некоторый контейнер и измеряло сколько памяти получилось после того, как мы выделили под них память. Потом удаляли ссылку на контейнер. И вызывало фугц. И после этого мерило количество оставшейся памяти и делило на миллион. И говорила вам сколько памяти занимает ваш объект. То есть никакого удобно способа получить размер в Java нет. И все что вы можете — это сделать некие оценки. Сейчас я буду говорить вам про эти оценки. И перед тем, как приступить к этому. Мне очень нравиться этот слайд. Он отношение к Java в общем то не имеет, он про размеры. То есть это размер Годзиллы, как размер менялся в зависимости от времени. Мне кажется, тенденция на лицо и имеет некое отношение к Java объектам, с которыми мы с вами тоже работаем и со временем и с разрядностью.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Точный размер объекта зависит от:  - версии Java  - издателя (Oracle JDK, open JDK...)  - разрядности ОС  - параметров запуска JVM  Для измерения размера используют «инсрументы» |

Так вот, значит, первое что вы встретите, пытаясь разобраться с количеством памяти под объекты это то, что вам никто точно не скажет, в спецификациях нигде ни что не прописано. То есть размер зависит от версии Java. Вот я привел вам Годзиллу уже. Зависит от того, кто писал JDK. Вы можете написать свое собственное JDK и там сделать что хотите. Спецификация не говорит какого размера должен быть объект. От разрядности оси, в которой вы работаете. Это в общем то очевидно, потому что размер ссылок вас меняется и еще от параметров запусков JVM. Параметры запуска поменяли и количество памяти у вас тоже поменялось. И для измерения используются некие инструменты. И про один из этих инструментов я вам уже только что расскажу.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | Размер объекта состоит из:  - Object header (8 byte)  - размера примитивных типов  - размера ссылок (32 или 64 бита на ссылку)  Гранулярность 8 байт |

Однако же, есть некие признаки, по которым вы можете попытаться разобраться с размером объекта, с которым вы работаете. Так вот значит. У каждого объекта есть хедер, 8-ми байтный. В этом хедере мы храним ссылку на класс. То есть вам же нужно как‑то класс, знать где он лежит, объект класса. И мы храним еще поле под хеш. Вот это очень интересное место, я вам когда говорил про объекты, говорил что у каждого объекта можно взять хеш. Так вот, по правилам хеш меняться не должен. То есть вот есть объект и между запусками в процессе работы хеш меняться не должен. Хеш при запросе у объекта принято Java вычислять на основе текущего адреса в памяти. Но проблема в том, что адрес объекта может меняться, потому что сборщик мусора может начать его двигать. То есть в Java адрес объекта величина изменяемая. Он может в разных местах находиться в процессе, менять свое положение в процессе работы приложения. И вот чтобы не потерять этот хеш, его хранят в хедере. Если у вас объект массив, то в массиве хедер будет уже не 8 байт, а 12 и еще 4 он потратит запоминание своего размера. Из этого примера можете сделать вывод, что в Java невозможно массивы, размером более чем Integer.MAX\_VALUE, то есть записать. Если учесть, что опять же, что они у нас только положительные, в значении можно использовать для размера, то это 2 с лишним миллиарда элементов, а потом все, больше создать нельзя. Размер объекта, кроме хедера, зависит размер еще примитивных типов, который в нем. Как я уже говорил, примитивные типы — это базовая вещь для всего. Поэтому от них тоже размер зависит. В Java он не зависит от порядка в каком вы пишите свои переменные как в C++, например. То есть нет смысла задавать сначала double, потом к ним снизу присваивать boolean. От этого, к счастью, мы спасены, но просто помните всегда, что между вами, когда вы пишите приложение и вашим пользователем еще стоит 2 компилятора. Первый, который байт‑код делает, второй just‑in‑time компилятор, они весь код переделывают как им нравится. То есть такого уровня оптимизация нам, к счастью, делать не нужно. Но тем не менее размер примитивных типов кончено влияет на размер объекта. И размер ссылок. Размер ссылок у вас зависит от разрядности. Кроме того, у нас есть некая гранулярность 8 байт, то есть если мы получили допустим 35 байт, то увеличиваем размер до 40. То есть всегда должна быть кратна 8 в Java. И то есть если все эти знания применить к строкам, то совершенно внезапно мы обнаружим, что когда мы создаем пустую строку, просто сказали String a = "", то мы сразу в памяти выделяем 40 байт под эту пустую строку.

2 видеоразбора примера

3 Работа с базами данных

3.1 JDBC

3.1.1 Ознакомление

Добрый день. Давайте начнем очередное занятие. Если на предыдущих уроках мы с вами разбирали язык, на котором пишем, и формулировали задачу, которую мы будем решать, и даже начали решать ее на сервере со стороны фронтенда, то сегодняшнее занятие будет полностью посвящено серверной части. Нам с вами нужно добавить в нашу систему персистентность. Сделать так, чтобы данные, которые оставляет пользователь в нашем приложении, не исчезали при рестарте системы. Еще лучше так, чтобы данные, которые ввел пользователь, сразу же оказывались у нас в базе, чтобы мы могли не беспокоиться о том, что, если вдруг наше приложение перестанет работать, пользовательские данные будут потеряны.

Итак, разговор будет про базы данных и про работу с базами данных из Java. Начнем мы с вами со стандартной библиотеки, с JDBC. Это то, что создатели языка приготовили для нас, чтобы мы могли этим пользоваться при работе с базами. Потом мы с вами разберем лямбды. Это нововведения, добавленное в Java в 8 версии. Я покажу как мы жили без лямбд и что нам лямбды добавили. Это будет обзор лямбд. Я не буду особенно вдаваться в подробности. Если функциональное программирование вам интересно, то найдите описание лямбд в Java и вообще этого подхода сами. А мы с вами разберем далее один из стандартных шаблонов проектирования работы с базой на низком уровне. Сейчас уже редко кто пишет эти вещи сам, потому что есть хороший библиотеки, которые от вас это скрывают, но важно будет разобрать внутренности, как это устроено. Если вдруг вам придется писать какие‑то особенности для баз, то хорошо будет, если вы будете понимать хотя бы где искать. В заключении первой части разберем транзакции, то есть в общем‑то базы нам дают отличные возможность совершать действия транзакционно и будет странно, если мы не будем этим пользоваться.

Во второй части мы разберем аннотации. Аннотации — это некий дополнительный к языку функционал, позволяющий подписывать все, что в классах для того, чтобы другие библиотеки или вы сами могли по этим подписям как‑то реагировать особенно на объекты, либо ваши классы. Подробности в теме про аннотации. И аннотации нам будут нужны потому, что мы с вами разберем что такой общий крешл маппинг. И посмотрим одно из самых популярных решений в реализацию ORM — это Hibernate. Разберем Hibernate воржнлич, некий промежуточный вариант языка запросов между SQL и объектами. То есть это попытка сделать запрос с помощью конфигурации объектов. В общем довольно удачная попытка. В заключении я вам сформирую задание, чтобы я хотел, чтобы вы у себя в приложении сделал и как это все должно работать.

3.1.2 Термины и сокращения; инструменты

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | JDBC — Java Database Connectivity  ORM — Object Relational Mapping  JPA — Java Persistency API  Hibernate — популярная библиотека для ORM, implements JPA  DAO — Data Access Object |

Начнем мы с вами с определений. Как я уже говорил на первом своем занятии очень много в Java сокращений 3‑х или 4‑х буквенных, первая буква из которых J. Никто вам ни в каких статьях не будет расшифровывать, поэтому лучше сразу эти сокращения понимать и выучить. Я уже упомянул здесь JDBC. Это название для библиотеки, который входит в состав стандартной библиотеки, то есть поставляет вместе с JRE или JDK. Там собраны все нужные для работы с базами классы и интерфейсы. На библиотеку можно смотреть с 2‑х сторон. Одна сторона — это сторона, которой мы будем пользоваться. Это набор классов и интерфейсов, которые мы у себя в приложении будем использовать для того, чтобы работать с базой. С другой стороны, этот интерфейс видят разработчики баз данных. То есть для того, чтобы использовать JDBC, чтобы мы могли использовать JDBC у себя в приложении, разработчики баз данных должны со своей стороны реализовать некоторую функциональность, которая заявлена в JDBC, то есть получается, что мы от JDBC берем классы интерфейсы, которые нам нужны в работе, а разработчики баз данных, если они хотя чтобы к их базам можно было обращаться из Java, должны написать свою собственную реализацию неких интерфейсов, заложенных в JDBC. И тогда через эти интерфейсы мы сможем с их функционалом, написанный разработчиками баз данных, работать. То есть получается, что как бы JDBC — это прослойка между нами, с одной стороны, и базой, с другой стороны. Получается у нас: мы, наш код, потом JDBC, потом драйвера, которые написали разработчики баз данных, потом база данных. JDBC здесь как раз по середине, то есть это некоторая API для нас и для разработчиков баз данных.

Потом очень важный термин. Он не только про Java, он вообще про работу с базами. Это ORM. Разработчики последние 15 лет пишут, используя объекты. То есть мы пишем приложения, используя понятия о классах и объектах. Мы думаем в терминах объектов. У нас есть собственно взаимодействие частей нашего приложения. Оно происходит через работу объектов друг с другом. Все данные у нас реализованы в объекты. С другой стороны, база данных там все реляционное, то есть там таблицы. Я не имею в виду базу данных в смысле менеджмент, я имею в виду саму структуру данных. И там таблица. У нас данные объекты, а у них там данные на таблицы. И вот чтобы мы могли работать с базами удобным для нас способом есть такое понятие как ORM, которое, с одной стороны, объекты, с другой стороны, реляционные схемы. А маппинг — это то, что перекладывают одно в другое.

Далее мы с вами встретим понятие JPA. Так же как и JDBC — это часть стандартной библиотеки. Это набор интерфейсов, классов и аннотаций. Главное здесь еще аннотации, которые разработчики языка дали тем, кто разрабатывает OPM решение, то есть для тех, кто пишет ORM. В составе библиотеки есть JPA. Они у себя могут это использовать, и мы у себя можем это использовать для того, чтобы их решение у себя применять. JDBC — это как бы посерединке между нами и разработчиками сторонних библиотек для работы с базами. И JPA — это тоже API, которое посерединке между нами и разработчиками ORM решений.

Одно из самых популярных на данный момент ORM решений — это Hibernate. Мы ее тоже с вами разберем. Она использует JPA во всю у себя в работе.

Пока мы будем разбирать Hibernate, либо классическое решение, описанное на чистом JDBC, нам и там и там встретиться понятие DAO. Это то, во что мы с вами будем заворачивать непосредственную работу с базой. То есть это объект, позволяющий нам получить доступ к данным, либо записать наши данные в базу. На самом базовом уровне можно считать, что DAO — это уже некоторое маленькое ORM, то есть вы в него передаете объекты, а оно вам их в реляционные схемы раскладывает. Либо это обертка над Hibernate, если вы используете Hibernate для этих решений.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | https://dev.mysql.com/downloads/  MySQL Community Server  MySQL Workbench  MySQL Connector Connector/J  <dependency>  <groupId>mysql</groupId>  <artefactId>mysql-connector-java  </artefactId>  <version>5.1.35</version>  <dependency> |

Перед тем, как идти дальше, нам с вами, разобрав сокращения, нужно еще некие действия произвести с нашим кодом и с окружением, в котором будем работать. На вашей рабочей машине вам нужно будет установить database сервер. Я предлагаю просто поставить MySQL. Я нигде специально не буду именно под MySQL затачивать свои предложения или решения, вы можете здесь использовать любую реляционную базу. Из известных и популярных в общем доступе MySQL и PostgreSQL. Вот насчет не реляционных — здесь будет чуть сложнее. Почему сложнее, я объясню. Если вы идете по стандартной схеме, используете MySQL, то вам нужно поставить сервер, поставить Workbench и еще скачать Connector/J. Connector/J — это как раз та самая реализация JDBC от разработчиков баз данных. И вам нужно будет подключить по зависимостям к себе этот самый Connector/J, либо той базой, которой вы будете пользоваться.

3.1.3 JDBC

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Java Database Connectivity — API для работы с базами из приложения на Java  Предназначена для работы с реляционными базами данных  Предоставляет методы для получения и обновления данных  Не зависит от конкретного типа базы  Java Application ‑> JDBC API ‑> JDBC Driver Manager ‑>(1)  (1) ‑> JDBC Driver ‑> Oracle  (1) ‑> JDBC Driver ‑> SQL Server  (1) ‑> JDBC Driver ‑> ODBC Data Source |

Когда мы с вами разобрали определения, поняли что нам надо выкачать, как вообще все это в общем виде выглядит, давайте погружаться в детали. И как я обещал, первое будет JDBC, API для работы с базами. Эта самая JDBC придумано для работы с реляционными базами, то есть в принципе использовать JDBC по задумке можно относительно легко переключаться между MySQL, PostgreSQL, MySQL Oracle базой. Потому что между этими базами гораздо больше сходства, чем разницы. То есть они все реляционные, у них у всех больше меньше степени одинаковой SQL для того, чтобы с ними работать. С не реляционными базами сложнее, потому что там между этими базами гораздо больше разницы чем общего, поэтому работа через JDBC с ними сложна. Потому что разработчикам этих баз будет трудно поддержать все то, что JDBC представляет разработчику приложений. Поэтому мы с вами сейчас будем разбирать работу с реляционными базами. Мне кажется, это правильно, хотя здесь может некий конечно холеварство. JDBC не зависит от типа конкретной базы, с которой работать. Опять же если это реляционная база. То есть миграция с одной базы на другую должна происходить относительно просто. Мы на Аллодах использовали MySQL, а в Skyforge мы использовали PostgreSQL. И вот этот переход был не сказать, что совсем прост, но более‑менее прост. Особенности были в том, что мы еще на Skyforge стали использовать Hibernate, до этого использовали шаблоны Executor, которые я сегодня буду рассказывать. В принципе, можно взять и переключиться на другую базу. Бонус от использования JDBC будет именно то, что вам при написании кода, который будет работать с базой, не так важно какая именно конкретная база там у вас лежит. То есть происходит некое такое разделение на две половины. С одной стороны, вы живете, а с другой стороны разработчики драйверов для баз данных.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | JDBC API  Connection  Объект отвечает за соединение с базой и режим работы с ней  Statement  Объект представляет выражение обращения к базе  ResaultSet  Объект с результатом запроса, который вернула база |

Для нас, для разработчиков JDBC API, представляет три базовых сущности, которой мы с вами будем пользоваться. То есть в первую очередь это Connection. Это класс, объект которого будет держать connect к базе. То есть вы перед тем, как начать общаться с базой, должны установить с ней соединение. И JDBC API вам представляет для этого Connection. То есть вы создаете Connection, говорите ему куда и как подсоединяться, и он подсоединяется. Как происходит вообще работа с базами? Неплохо бы на самом деле, чтобы в начале этого занятия у вас было некоторое представление о работе с базами, может быть из скриптовых языков. Если нет, попробуем разобрать с нуля. Когда вы установили соединение, вам нужно к этой базе обращаться. Здесь нам помогает Statement. И это тоже класс. И объект этого класса, там их в иерархии несколько разных типов, они позволяют нам писать обращение к базе, то есть установить соединение, написали запрос, свернули запрос в Statement, выполнили Statement. Исполнение Statement означает для нас исполнение запроса в базе. И финальная. Мы же не просто для удовольствия к базе обращаемся. Мы что‑то хотим от базы. Либо поменять, либо запросить некий набор данных. Вот если мы хотим запросить некий набор данных, то он должен в виде чего‑то вернуться. И вот это в виде чего‑то — это ResultSet. После работы Statement, если Statement был на запрос данных, нам вернет ResultSet. И в этом ResultSet будут неким образом упакованы результаты, по которым мы сможем пройтись. Как именно я позже расскажу.

3.1.4 JDBC. Connection

|  |  |
| --- | --- |
| 1.  2.  3.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | JDBC Driver Manager  Название класса драйвера com.mysql.jdbc.Driver  Создание объекта драйвера при помощи reflection:  (Driver) Class.forName("com.mysql.jdbc  .Driver").newInstance()  java.sql.DriverManager — класс хелпер для работы с драйверами  Регистрируем драйвер:  DriverManager.registerDriver(driver); |

Мы уже с вами знаем, что такое JDBC. Знаем, что с 2‑х сторон ее: с одной стороны пишем мы, с другой стороны разработчики. В целом уже представляем содержимое JDBC. И теперь самое время нам просто погрузиться уже в детали. После того, как мы установили базу куда нам нужно, скачали к себе драйвер для базы по зависимости, мы запускаем наше приложение. Не важно с JDBC или с Hibernate мы будем работать. И наше приложение должно установить соединение. То есть еще как бы до установки соединения, надо разобраться с тем, к какой именно базе нам соединяться. То есть наше приложение должно понять, что оно будет работать с какой‑то определенной базой. Не в смысле базой в смысле с инстансом, а в смысле какого типа. То есть она должна понять вторую часть JDBC. Точнее ту самую часть, которую пишут для вас разработчики баз данных,. Ваше приложение должно разобрать, чтобы понять что вот есть такая база, к ней есть определенный драйвер, есть даже такой тип баз, и не просто конкретной базы где‑то стоящей, а вообще просто есть допустим база MySQL, оно должен откуда‑то это узнать. И вот чтобы оно могло это узнать, есть понятие JDBC драйвера. И драйвер менеджера. То есть вы когда выкачали вот этот коннектор J вы на самом деле выкачали драйвер. Внутри этого коннектора, выкачанного по зависимости, в нем лежат Java‑классы. Самый главный из них — это Java класс, который надо будет нужен, это вот (4). Он лежит по этому пути. Мне о нем сказали разработчики вот этого самого драйвера. Можно на сайте MySQL прочитать, что если вы хотите обращаться к MySQL базе из Java, то вам нужно использовать вот этот класс. Мы должны объекты этого класса создать. Здесь начинается некоторый забег вперед, потому что для создания объекта драйвера удобно использовать рефлекшн. Когда я вам рассказал про классы, я говорил, что объект класса можно получить по имени. Здесь именно это и происходит. Я говорю Class (7), это название класса. Вызов статической функции forName() (7). Потом я говорю что же я хочу найти (7, аргумент функции). Вот этот драйвер я хочу найти. Он уже должен быть у меня в runtime. Этот драйвер, он уже должен быть загружен в виде класса в runtime, classLoader‑ом. И поэтому я по имени могу его найти. И после этого я могу вызвать его дефолтовый конструктор newInstance() (7). В строке 7 я создаю объект драйвера. Так же как если бы через new я его создал. Я его создаю через имя. Это удобнее, потому что если вы хотите поменять драйвер, то вы должны будут просто вот эту строку с названием увести в config, а потом при желании менять на то, что хотите. И таким образом у вас прямой зависимости на конкретное решение баз данных нет. Есть только вот такое решение. После того, как вы этот драйвер сделали, вам нужно его зарегистрировать, потому что при желании вы на самом деле можете таких драйверов зарегистрировать несколько, то есть вы можете сказать мое приложение будет работать и с базой MySQL, и с базой PostgreSQL. Вы должны будете выкачать зависимости оба драйвера, оба драйвера создать. А потом положить его в менеджер. Есть специальный java.sql.DriverManager. Это уже в стандартной библиотеке класс, входит в JDBC. И вы туда должны драйвер зарегистрировать. Просто так же как вы статическим forName() вызывали статическую функцию, так же у драйвера менеджера есть статическая функция зарегистрировать 13. После того, как вы это проделали, ваше приложение может начинать соединяться с базой.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | Driver driver = (Driver) Class.forName(  'com.mysql.jdbc.Driver')  .newInstance();  DriverManager.registerDriver(driver);  StringBuilder url = new StringBuilder();  url  .append('jdbc:mysql://') // db type  .append('localhost:') // host name  .append('3306/') // port  .append('bd\_example?') // db name  .append('user-tully&') // login  .append('password-tully'); // password  // URL: "jdbc:mysql://localhost:3306/  // lecture\_db?user-tully&password-tully"  Connection connection = DriverManager  .getConnection(url.toString()); |

И вот здесь нам понадобиться класс Connection. То есть опять же я здесь повторил работу с драйверами (1-2). После того, как я эту работу проделал, я должен установить Connection. Уже непосредственно конкретному интсансу базы, то есть не просто к чему‑то. Просто говорю у меня будет база определенных типов. Нет я уже должен иметь на машине куда я хочу соединиться конкретной датабазой, система управления базами. И кроме того, там уже должна быть создана база. То есть в принципе есть возможность из Java, как из любого языка программирования, создать базу, если база не создана. Но это чуть выходит за рамки нашего курса. Я подразумеваю, что вы базу уже руками создали. То есть база у вас там уже есть. Зашли на сервис и создали базу. Если вы хотите разобрать как программно создавать базы, то вам просто придется подсоединяться, но только к системным базовым. Во всех базах есть системные базы, и вот вам нужно подсоединяться к системным базам, чтобы создать свою собственную базу. Мы будем считать, что у нас база уже есть, что у нас на localhost (10) на стандартном для MySQL порту (11) есть база с именем db\_example (12). И у нее есть пользователь (13). И у пользователя есть пароль (14). Что вы здесь видите? Вы здесь видите StringBuilder (6). То есть я заодно еще решил показать StringBuilder. StringBuilder — это способ работы со строками, когда вам нужно строки соединять. Если вы будете просто производить конкатенацию строк, создавая новые строки, то это очень неэффективно по памяти, потому что каждый раз вам придется создавать новый объект. По производительности и по памяти плохо. Потому что, соединяя строки, вы создаете новую строку, то есть новый объект. Чтобы так не делать, в Java есть StringBuilder. И он позволяет просто через вызов функции append() подсоединять строки друг с другом. Вначале их набить сколько вы хотите соединить, а потом через вызов toString() собрать все вместе. То есть на самом деле в URL у меня будет лежать (16‑17). Вот полная строка коннекта к базе. Здесь я указываю протокол, по которому соединяюсь, и тип соединения (9), куда соединяюсь (10), на какой порт соединяюсь (11), имя базы (12), пользователь (13). После того, как я весь этот url сформировал, я могу по нему получить Connection. То есть Connection конкретно к этой базе, который считает что она уже создана. Если она не будет создана, то здесь в момент создания Connection будет исключение. Драйвер менеджера, статической опять же функции, я могу взять и получить Connection по этой самой строке. Connection менеджер поймет какой именно драйвер нужно использовать, потому что мы здесь ему скажем, что у нас ни что иное, как MySQL (9). Таким образом мы получаем Connection. И дальше мы уже с вами готовы с этой базой работать.

3.1.5 Statements и ResaulrSet

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | Statements  JDBC позволяет создавать и выполнять запросы к базе  Update statements: CREATE, DELETE, INSERT...  Query statements: SELECT  Интерфейс  Statement  PreparedStatement  CallableStatement  Query statements возвращают ResultSet  Update statements возвращают число измененных строк |

Ну вот, Connection мы с вами создали. Так как у нас появился Connection в базе, мы можем через этот Connect с базой что‑нибудь делать. Вот это что‑нибудь делать оно означает в большинстве случаев два типа запросов. Мы можем что‑нибудь в базе поменять или можем что‑нибудь у базы запросить. То есть вот две большие группы запросов Update statements и Query statements. Update statements вроде как они ничего не должны возвращать, но потому что их задача что‑нибудь поменять просто. Но если вы писали запросы через консоль в базу, то вам от базы в консоль приходил ответ. Он по задумке должен приходить, а сколько именно записей, сколько строк было изменено по результатам этого вашего запроса. Update statements, те которые направлены на изменение чего‑нибудь в базе, они по хорошему должны вернуть результат своей работы в виде количества измененных строк в таблице. При работе с базой в Java происходит то же самое, то есть Update statements вам вернут количество измененных строк. А Query statements вам должны будут вернуть набор данных, которые вы запросили. Statements предоставлены 3‑мя интерфейсами, то есть работа с ними может быть сделана не то что 3‑мя способами, а просто для разных задач есть 3 разных типа statements в JDBC. Самый обычный Statement он называется Statement, просто Statement. Вы его создаете, предаете в него запрос и подсоединяете к базе, исполняете Statement. Запрос опять же может быть любого из 2‑х типов. Может быть Update, может быть Query. В этом смысле разделения нет. То есть и тот и другой Statement может у вас работать и с запросом на изменение, и с запросом на извлечение. По результатам опять же можно у него спросить либо количество измененных строк, либо ResultSet, который получается по результатам. Кроме обычного запроса есть еще PreparedStatement. Это возможность сделать некую заготовку, шаблон запроса и, подставляя в него конкретные значения, исполнять его, то есть не как в случае с предыдущим Statement передать туда просто запрос и исполнить, а сделать некий шаблон запроса, в нем специальными символами пометить куда подставлять значения, а потом в цикле подставлять значения и записывать результат в базу, то есть некий подготовительный, то есть получается что работа разделена на 2 части. С одной стороны приготовились, а потом исполнили. В случае с обычным Statement просто вам каждый раз придется новый Statement создавать. И CallableStatement он нужен для работы с хранимыми процедурами. Вы с вами разбирать его не будем. Просто чуть‑чуть выходит за рамки курса. Если вам интересно, сами посмотрите. Про формат возврата я уже сказал. Все вот эти Statement и работу с ними я покажу на примере. То есть я покажу пример кода со Statement и PreparedStatement. Пока просто слова о том как вообще с этим работать из чего все это состоит.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | ResaulrSet  Содержимое результата запроса  Перемещение по строкам  next()  previous()  isLast()  Доступ к полям текущей строки  По имени колонки: getBoolean(String name), getLong(String name) ...  По индексу колонки: getBoolean(int index), getLong(int index) ... |

Картинка

Если мы с вами в случае изменения получим просто число измененных строк, то в случае с запросом мы с вами получим ResaulrSet. ResaulrSet — это объект, который будет содержать результат запроса. И внутри него будет некая таблица. Логично, потому что вы запрашиваете реляционную базу в ней внутри таблицы. Запрос ваш SQL, он очевидно содержит на выход некую таблицу. И ResaultSet — это просто некая обертка над этой таблицей, которая позволяет вам по этой таблицы интерироваться. Когда вы, чтобы понять как именно вы можете использовать этот самый результат, представьте что вот вам вернулась таблица как та, что здесь на слайде показана и изначально, то есть у вас есть как бы 2 поинтера: горизонтальный и вертикальный. Двигая горизонтальный и указывая вертикальным что именно вам нужно вы можете извлекать конкретные значения конкретных полей. Когда вам только ResaultSet вернулся в нем вот тот самый указатель на строку он показывает на некоторую строку перед первой. Я себе это представил как просто указатель на имена, на заголовок. То есть у таблицы есть заголовок, в нем имена колонок и как будто указатель на него показывает. Это сделано затем, что если ваш результат запросов пустой, то есть ничего нет, запросили базу, а ответ пуст, то кроме списка заголовков вам ничего больше не придет. Если вы для того, чтобы начать работать с этой таблицей с ResaulSet вам нужно начать итерироваться по строкам, то есть вначале вы перемещаетесь по строкам, а потом вы в этой строке можете получать запрашивать значения, то есть у вас есть вот этот глобальный указатель внутри ResaulSet. Вы можете сказать ему переместиться. Задать ему задачу перемещение наследующую строку. И перед этим проверить является ли указатель на последнюю запись. Если он последний, то, очевидно, вам двигаться дальше не стоит. Если он не последний, то вы можете next() просто взять и переместиться вниз. Или при желании переместиться вверх. То есть вы можете двигаться по объекту и каждый раз у вас будет выбрана какая‑то определенная строка. В этой строке вы можете обращаться к колонкам, то есть понятно что наша задача перенести эту самую колонку из ResaultSet в некий удобный для нас формат данных, то есть тот, который нам нужен. Допустим у нас в этой колонке списки пользователей. Из списка пользователей имена пользователей и id пользователей. Мы его перемещаясь по списку пользователей можем извлекать id‑имя и себе куда‑то в нужный нам формат складывать. Если мы перемещаемся по строке, просто вызывая next() и вызывая isLast(), то при получении непосредственно значения мы должны уже знать что же именно мы хотим спросить. Мы должны знать либо индекс колонки, либо мы должны имя колонки. С именем колонки, мне кажется, надежнее, но потому что в индекс вы должны понимать, где у вас что находиться. Кроме того, индекс при обращении к ResaultSet еще начинается с единицы. То есть вот это вот не ноль, а первая колонка. И так как мы все привыкли, что у нас все всегда индексируется с нуля это тоже можно вызвать некоторые ошибки у вас. Поэтому просто обращайтесь по имени к колонке. Просто вы знаете, что у вас есть колонка с именем Name и вы можете взять и получить не Name, а Age. И Age, если допустим это по какой‑то причине long, вы можете взять и сказать getLong(String name), здесь в качестве имени написать Age и получить его значение.

3.1.6 close()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | close()  Все обращения надо закрывать вручную  resultSet.close()  statement.close()  connection.close()  Или использовать AutoCloseable |

Мы с вами умеем устанавливать драйвер, умеем подсоединяться к базе, создавать запросы к этой базе, получать либо число измененных строк, либо ResualtSet и итерироваться по ResualtSet, получать данные которые в нем есть. После этого вы уже на самом базовом уровне с базой работать умеете. Остается важная тема. Я специально вынес ее в отдельный шаг, специально приготовил для нее вот такой кричащий слайд. Вам ресурсы, которые вы используете, нужно закрывать. Это касается не только работы с базой, это потом я вспомню еще про работу с файлами. Но это очень важно. Смотрите, для вашего приложения, есть некое окружение. Окружение ближнее — это библиотека классов, объекты, которые вы из соседей в этих библиотеках создаете. И есть окружение, которое внешнее, то есть то, с чем вы общаетесь снаружи. Допустим, с файловой системой, с операционной системой, с базами данных. И вот когда вы создаете к ним соединение, поработав с ним вы должны эти соединения закрыть. Вот этот самый close() очень важно, иначе у вас будут утекать connection, у вас будут утекать ресурсы по statement, и в конечном итоге у вас эти ресурсы закончатся. Поэтому после работы вы должны все закрыть. Начну в этом списке с конца. Вам нужно закрывать соединение при работе с базой, но вот это наверно самое редкое действие, потому что по‑хорошему вы должны его закрывать, когда вы закрываете свое приложение. То есть по задумке ваше приложение если оно работает с базой, оно поднимается, соединяется с базой и после этого соединение держит. Либо более сложный вариант, когда вот это создание соединения получаете конекшн в специальной библиотеке Конекшн пул который называется, которая просто содержит пул коннектов. И оно уже тогда управляет соединениями. Но правильно, если ваше приложение перед закрытием все коннекты закроет. В принципе оно и так само по себе закроет, когда будет выключаться. Если вам нужно, допустим гасить какие‑то модули, отдельно работающий с базой, то вам при этом нужно закрывать соединение. Об этом часто все забывают. То есть не так фатально, как вот то, о чем я сейчас дальше буду рассказывать, но тоже плохо. Когда вы создаете свой statement, то при работе со statement, вам этот statement тоже нужно закрыть. То есть он тоже держит некие ресурсы. И поработав со statement, вы должны statement закрыть, явно вызвав у него close(). И еще close() есть у ResaultSet. То есть смотрите, ResaultSet тоже держит некие ресурсы, как минимум таблицу объектов, которую он с результатами запроса. И ее тоже нужно закрывать. Закрытие statement подразумевает закрытие ResaultSet, то есть если вы statement, результатом которого стал ResautSet, закроете, то ResaultSet у вас тоже закроется. Естественно, если вы connection закроете, у вас тоже все каскадом закроется. То есть они каскадом друг друга закрывают. Это очень важный момент, особенно при закрытии statement, как он влияет на закрытие. После закрытия ResaultSet вы к нему обратиться уже не сможете. То есть данные он вам не скажет. И это очень важно как раз чуть дальше, когда мы дойдем до Executor, я подчеркну это мест, что нам нужно сделать так, чтобы открытие закрытие происходило в некой нами написанной библиотеки, но при этом чтобы еще и данные ResaulSet не потерять. Плохо в этой ситуации то, что при закрытии может произойти исключение, и тогда как бы вам нужно еще раз убедиться в при отлове исключения о том, что вы действительно все закрыли. К счастью, для нас в 7 уже Java появилось понятие AutoCloseable. Все эти наши ресурсы, которые нужно закрывать, они AutoCloseable. И есть специальная структура модификация try‑catch, которая позволяет в try показать что нужно в try закрыть и после выхода из try она автоматически будет закрыта, Java‑машина за вами проследит. И пример этого я покажу чуть позже, особенно когда мы с вами будем исключения разбирать.

3.1.7 Лямбды

Продолжим занятие. И сейчас некоторое отступление. Перед тем, как нас с вами перейти к Executor. Для того, чтобы организовать все те знания работы с базой, которые вы уже получили, в некий паттерн. Можно пытаться сочинить свои велосипеды с тем сырым набором возможностей, о котором я уже вам рассказал, какие‑то пытаться комбинации делать. Но лучше уже использовать готовое. То есть на то они паттерны и придуманы, что вы их друг у друга подсматриваете и у себя копировать. И вот об одном из них, об Executor, я вам расскажу. И перед тем, как к нему перейти, нам нужно смотреть возможность передачи функции в другую функцию. То есть λ — это некий способ написать коды своему приложению, что нужно в функцию нужно передать другую функцию.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Задача  Функция в качестве переменной  «Библиотечный» код подготавливает данные для вызова метода  И вызывает его  Какой именно метод будет вызван решает «пользователь» библиотеки |

В принципе, задача эта стояла уже давно у разработчиков. Задача какая? Нам нужно, чтобы некий библиотечный код, то есть мы пишем какой‑то код и этот код в процессе работы библиотечный должен вызвать участок кода того, кто его использует. Самый близкий вам пример сейчас будет Jetty, который мы с вами используем для работы с вебом. И в Jetty мы пишем сервлеты. В сервлетах есть методы doGet(), doPost(). То есть, смотрите, Jetty содержит веб сервер, Jetty и есть веб сервер. И он содержит работу с запросами, но в некоторый момент Jetty нужно при обработке конкретного сервлета запросить наш собственный код, то есть библиотека, о которой я здесь пишу, библиотечный код. Представим, что это Jetty. А вот метод, который должна вызвать эта библиотека, это наш собственный метод, например doGet() или doPost(). И мы, как пользователь библиотеки, решаем что же именно, какой именно код здесь будет у нас использован. То есть мы должны в общем то в каком‑то смысле передать нашу функцию в качестве аргумента.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Передача интерфейса  Решение в Java 7  List<String> names = Arrays.asList('peter', 'anna', 'mike', 'xenia');  // передать в метод анонимный класс  Collections.sort(names,  new Compareator<String>() {  @Ovveride  Public int compare(String a, String b) {  return b.compareTo(a);  }  }); |

Как это сделано при работе с Jetty. Мы с вами наследуем некий класс от сервлета и в нем перечисляем функции. То есть мы явным образом говорим, что у нас есть библиотека, библиотека хочет определенный класс. И мы от этого класса наследуемся и в нем функцию определяем. Есть в предыдущих версиях по сравнению с 8 Java версиях возможность передавать функцию в другую функцию без создания дополнительного класса, так называемый анонимный класс. На этом слайде пример передачи анонимного, ну то есть решение задачи передать функцию в другую функцию через создания анонимного класса. Я для примера взял Collections. Это хелпер для работы с коллекциями и в нем есть функция sort(). sort() в добавок к тому, что именно она будет сортировать еще на вход хочет компаратор. То есть она должна понять, как сравнивать элементы. Чтобы их просотрировать, нужно их сравнивать. Как их сравнивать мы можем ей объяснить, передав. Мы должны передать некую функцию. Компаратор — это тоже из стандартной библиотеки интерфейс. Если мы хотим вот этот сортинг алгоритм, который в Collections будет по дефолту поменять на что‑то наше, мы как в случае, также как в случае с Jetty классический способ создать наследника с компаратором. То есть мы должны были создать где‑то у себя класс MyComparator implements Comparator {}. Внутри нашего класса, так как компаратор требует, чтобы все те, кто его реализует реализовали у себя функцию compare(), мы должны были эту функцию compare() здесь у себя записать, то есть внутри было бы public int compare(). Этот способ означал бы, что нам нужно на каждый компаратор придумывать отдельный класс. Можно этого не делать и использовать анонимный класс. Анонимные классы — это возможность прямо в теле функции передать 2 поля. Первое поле names, а второе поле — какой‑то компаратор. То есть если зайдете внутрь сорт и увидите там, что там на вход вначале коллекция, которую сортируем, а второе поле — это то, чем сортируем. Синтаксис выглядит так, что мы пишем new и пишем интерфейс или абстрактный класс, который мы как будто хотим создать. Но создать мы с вами его не сможем. Потому что он интерфейс или абстрактный класс. И нам компилятор скажет, что создать его нельзя, и хорошие среды разработки заодно предложат сделать здесь анонимный класс. И вы внутри скобок фигурных будете писать тело класса, как будто вы его создаете отдельно, но не отдельно, а прям в этом месте. И здесь вы должны будете перечислить список методов, который требует интерфейс. В данном случае это просто compare(). Одна единственная функция. Это способ создания анонимного класса. То есть совсем классическое решение — создать отдельный класс, в нем все определить и передавать объект классам в функцию. То есть если нам нужно передать куда‑то в какую‑то функцию другую функцию, мы передаем не функцию, а объект который класса, который эту функцию содержит. Это классическое решение. Чуть более сложное решение создания анонимного класса.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Лямбда выражение  Решение в Java 8  List<String> name = Arrays.asList('peter', 'anna', 'mike', 'xenia');  // анонимный класс  Collections.sort(names,  (String a, String b) -> { return  b.compareTo(a); }  ); |

И в 8 Java появилось возможность меньше писать текста. И они сказали зачем мы будем вообще создавать этот анонимный класс, когда все и так понятно, мы просто таким синтаксисом скажем, что у нас будет функция от 2‑х переменных с вот таким телом. И все. То есть они сказали не будем писать анонимный класс как на предыдущем слайде, а синтаксисом запишем переменные указывают на тело, которое эти переменные используют. Это и есть лямбда.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Лямбда выражение  Решение в Java 8  List<String> name = Arrays.asList('peter', 'anna', 'mike', 'xenia');  // анонимный класс  Collections.sort(names,  (a, b) -> return b.compareTo(a);  ); |

И потом они пошли дальше и сказали нам в общем‑то не обязательно даже писать типы, потому что типы мы и так знаем. Потому что функция sort() уже знает какого типа ей нужно на вход лямбду. Поэтому типы тоже можно убрать, оставить a и b, то есть a и b — это переменные той функции, которую мы должны передать в функцию sort(). А правая часть (8) — это тело функции, которое мы должны передать. Получилось лаконичнее, но нужно просто некое время потренироваться, чтобы понимать по коду что здесь происходит.

3.1.8 Executor. execUpdate()

Ну вот мы с вам уже знаем как работать с базой и уже приготовились для того, чтобы организовать эту работу таким образом, чтобы не просто все работало хорошо, а еще чтобы можно было поддерживать код, на котором мы пишем, относительно просто расширять их, и как говорил Страустров, чтобы не отстрелить себе ногу. И Executor — это паттерн для работы с базой, которая позволяет нам остаться при своих ногах.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Executor  Объект, который содержит методы для работы с запросами  Обработка запроса на создание, вставку, обновление и удаление  execUpdate(Connection connection, String update)  Обработка запроса на получение данных  execQuery(Connection connection, String query, ResultHandler handler)  public interface ResultHandler {  void handler(ResultSet result)  `throws SQLException;  } |

Итак, мы умеем посылать запросы к базе, у нас уже есть соединение, но нам при этом с вами нужно при работе с базой создавать соединение и закрывать соединение. Поэтому нам хорошо будет их всех тех действий, из того кода, который у нас будет с вами будет работать с непосредственно с вставкой, удалением, изменением данных вынести ту общую часть, которая будет заниматься непосредственно работой с базой. Можно просто создать класс, в него положить статические методы, в каждом из этих статических методов в него передавать Connection, можно даже statement не передавать. Можно в нем создавать запрос, открывать statement, заполнять statement, забирать результат, куда‑то передавать. Можно так сделать. Так будет работать. Но лучше будет, меньше будет ошибок и проще будет читать код и расширять его потом, если мы отделим изменяемую часть от неизменяемой. В каждом нашем действии работы с базой есть некая неизменяемая часть — это создание statement, стандартная работа с ним, исполнение statement и закрытие statement. Не зависимо от того, что вы хотите выполнить. Вы хотите извлечь какие‑то определенные типы данных, определенные результаты из таблицы, или вы хотите поменять их — все равно эти действия нам нужно делать. И нам эти действия нужно куда‑то спрятать. Чтобы у нас были люди, ответственные за написание той части, которая работает с базой, и люди, которые допустим готовят непосредственно сами строки запросов и получают результат. И Executor — это та самая часть, которая общая для всех запросов какие бы они ни были. Когда вы начнете работать с базой, вы довольно быстро поймете, что вам приходится делать одно и те же очень часто. В каждой функции работой с базой вы производите одни и те же действия. Когда вы попытаетесь эти действия выделить вы столкнетесь с неким набором трудностей. Сейчас я попытаюсь эти трудности разрешить с помощью объяснения Executor. Во‑первых, это класс. В этом классе у нас будет, мы можем создавать объекты этого класса. В нем можем хранить connection. Connection, как я уже говорил, не нужно создавать каждый раз при каждом запросе. Сonnection вы создали и держите. Если вы в момент выполнения запроса поняли что Сonnection нет, разорвалось почему‑то, вы можете пересоздать Сonnection. Но создавать Сonnection в каждом обращении плохо, это очень сильно ресурсоемко задача. Поэтому создаем либо один Сonnection, если у нас простое приложение, либо несколько и организуем их в пул, если более сложное. Так вот, в Executor у нас уже есть Сonnection, мы его туда при создании Executor передали, мы его там храним. Либо мы можем Executor сделать статическим, тогда нам надо будет Сonnection нам каждый раз туда передавать. Так вот, как у меня здесь написано. Я здесь подразумеваю, что это статические функции, то есть в этом примере у меня Executor это некий хелпер. Значит в Executor у нас будет 2 метода execUpdate() и execQuery(). Разница у них в том, что от execUpdate() нам не нужно возвещаемое значение. То есть мы можем в execUpdate() возвращать количество измененных строк. То есть мы заранее знаем что именно нам вернет execUpdate(), он вернет Integer, количество измененных строк. Мы в него будем передавать Сonnection и будем передавать строку, содержащую запрос. С точки зрения программиста, использующего Executor, в Executor 2 метода. Один метод довольно прост, это execUpdate(). В него мы должны передавить Сonnection, в него мы должны передать SQL‑запрос и на выходе получить количество измененных строк, либо исключение, если что‑то пошло не так. С получением данных чуть сложнее, потому что в query, который уходит в получение данных в зависимости от того, какую query мы передаем, мы получим разные результат. То есть в случае с update там все просто, просто Integer вернулся всегда и все. В случае с query там может вернуться что‑то разное. Может вернуться одно значение, может вернуться строка, может вернуться лист строк. И нам нужно каким‑то образом понимать как именно, какое именно значение из этих query должно вернуть, то есть смотрите, это Executor, в нем execQuery(), она одна функция. Не по execQuery() на каждый запрос, а она всего одна. Она на вход получает Сonnection, query и еще она получает ResaultHendler. Вот ResaultHendler — это передача функции в функцию, о которой я до этого рассказывал, то есть ResaultHendler — это объект класса, содержащего функцию, которая поработает с ResaultSet. Дальше на примерах я поясню как именно мы будем это делать. Заранее я сразу скажу, что все это придумано за тем, чтобы не возвращать из execQuery() ResaultSet. Если мы в принципе ну может показаться что это нормальное решение, взять из execQuery() вернуть ResaultSet. Но в этом случае мы поручаем тому, кто вызывает Executor, задачу закрытие ResaultSet и закрытие statement. Потому что если мы statement закрыли, RecaultSet у нас будет тоже закрыт. То есть получается execQuery() может нам открыть запрос, запросить данные, но не может закрыть его. И это очень плохо, потому что это раскрытие деталей работы с базой и надежда на то, что вызывающая сторона выполнить все закрывающие действия, кроме того у нас в общем то, если мы возвращаем ResaultSet, а statement не возвращает, то statement окажется не закрытым. То есть плохая идея возвращать ResaultSet из execQuery(). Те трудности, о которых я говорил, с которыми вы встретитесь. Вот одна из них. Что возражать из execQuery(). Сейчас мы попытаемся эту проблему решить. С execUpdate() с ним это сильно просто. С execQuery() сложнее. Тут появляется ResaultHendler и еще потом я подключу сюда шаблоны, чтобы показать как лучше всего это место организовать. Итак, здесь на этом слайде я привел интерфейса ResaultHendler. У него будет одна единственная функция hendle(). Эта функция hendle() будет брать на вход ResaultSet. То есть мы таким образом в execQuery() передаем наш код, тот код, который на вход получит ResaultSet.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | Update Statement  public static int execUpdate(Connection  connection, String update) {  Statement stmt = connection  .createStatement();  stmt.execute(update);  int updated = stmt.getUpdateCount();  stmt.close();  return updated;  }  Примеры  execUpdate(connection, 'create table users  (id bigint auto\_increment, name  varchar(256), primary key (id))');  execUpdate(connection, 'insert into users  (name) values ('tully')'); |

Пока вернемся к execUpdate(), чтобы на примере посмотреть как execUpdate() работает. Дальше execQuery() тоже очень подробно разберем. Значит execUpdate(). С ним все относительно просто. Мы в нем должны создать statement, вызвав createStatement() у connection. Мы должны statement попросить выполнить update. update это так самая строка с update, которая к нас пришла. Когда строка Execut прошла, то вы можете считать, что задача в базе выполнена. То есть мы до этого места дошли, эту строку запустили. И если мы перешли на следующую строку без исключений, то есть исключение никакое не вылетело, то все, update прошел. Здесь сразу хочу сказать и еще вернусь к этой теме чуть позже. У нас обращение к базе синхронный. То есть мы действительно ждем, пока обращение будет сделано. После того, как мы это обращение сделали, мы у statement можем спросить количество строк, которые были обновлены. Мы можем закрыть statement. На самом деле код тут будет чуть более сложным, потому что тут еще всевозможные исключения могут быть выкинуты. Я именно в реальных примерах, когда мы до разбора примеров в среде разработки разберемся, я еще раз обращу внимание, что просто. Это иллюстративный пример, просто закрыли и все хорошо. Но закрыть мы обязаны здесь. (9) строка должна быть либо выполнена явно, либо мы должны AutoCloseable здесь сделать решение. И после этого мы знаем, что нам нужно вернуть Integer, мы этот Integer возвращаем. После того, как вы эту часть сделали, у вас уже есть Executor, в нем есть этот метод. Вы можете в своем приложении где угодно, где вы хотите работать с базой, взять его и позвать и сказать execUpdate, передать connection и передать строку с запросом. В строке (15) будет создана таблица с двумя полями. И потом при желании вы можете другой запрос (19) отправить на insert. Какой хотите запрос: на удаление строк, на удаление таблиц. execUpdate с этим справится. Вам вернет результат. И все в этом месте у вас будет хорошо.

3.1.9 Типизация. Executor. execQuery()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | public interface TResultHandler<T> {  T handle(ResulteSet resultSet)  throws SQLException;  }  public class TExecutor {  public <T> T execQuery(Connection  connection, String query,  TResultHandler<T> handler)  throws SQLException {  Statement stmt =  connection.createStatement();  stmt.execute(query);  ResultSet result =  stmt.getResultSet();  T value = handler.handle(result);  result.close();  stmt.close();  return value;  } |

Ну вот теперь самая сложна часть этой части урока. Нам с вами нужно не просто возвращать void из execQuery(). Нам нужно из execQuery() возвращать значение. Мы же не просто просили написать в лог результат. Нам же на самом деле нужно спросить Executor извлечь некие данные, передать в Executor строку с запросом и еще на выход от него получить значение, которое мы хотим. То есть мы хотим, чтобы execQuery() сам там себе внутри что‑то сделал при работе с базой. Работал непосредственно с statement, открыл его, выполнил, закрыл. А нам на выход вернул нужное нам значение. Причем это должен быть не ResaultSet. И вот, чтобы эту задачу решить, нам с вами понадобиться дженерики. На слайде я привел пример того, как это можно организовать. Нам для этого с вами понадобиться типизация интерфейса. Первый пример (1) — это модифицированный ResultHandler, типизированный неким значением. Я переписал ResultHandler, добавил в него типизацию. И теперь handle() возвращает теперь не void, а тот самый тип, который типизирован сам handler. Создавая ResultHandler для разных типов, я захочу получить от handler разные значения. То есть я знаю, что я хочу получить на выход. И это, что я хочу получить на выход, я как раз и передаю в качестве параметра типизации ResultHandler. Executor мне тоже нужно будет поменять. execQuery() у меня будет типизированная функция (7). Я здесь говорю эта будет работать с параметром. И эта функция будет возвращать тип, который я захочу задать при ее использовании. И этот же самый тип в точности этот же тип я буду передавать в handler (9), то есть я буду говорить handler у меня будет именно того типа, который будет возвращать execQuery(). Дальше по коду самого execQuery() практически все то же самое, что было в предыдущем пример. Мы создаем statement (11). Мы исполняем этот statement (13). Мы получаем ResaultSet (14), и мы передаем ResaultSet внутрь handler (15). Пока ничего не изменилось. Но внутри handler мы производим действия, которые приводят к сборке нужного нам результата, этого самого T value. То есть мы в handler описываем как именно собрать тип, который handler вернет в execQuery() и который в результате execQuery() вернет нам после вызова (19). То есть смотрите что у нас получилось. Мы подготовили statement. Этот statement должен собрать нам строку, то есть результат исполнения statement будет строка. Мы запрашиваем базу, говорим данное имя … и в результате строка. Мы знаем, что это должна быть строка. Мы создаем execQuery(), типизированной строкой, создает ResultHandler, типизированной строкой. В handler мы собираем строку на основе данных из ResaultSet и эту строку возвращаем. Если нам с вами понадобиться от execQuery() не собрать строку, а собрать List строк. Допустим, мы говорим дай нам имена всех пользователей, которые вчера заходили днем на наш сервис. И это будет список. Список имен, список строк. Поэтому мы в execQuery() говорим: execQuery() ты будешь возвращать список, вот тебе handler, который хочет список. Типизированный списком. И в этом handler мы на основе ResaultSet список собираем и возвращаем.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | TExecutor execT = new TExecutor();  String query = "selest user\_name from users  where id 1";  String name execT.execQuery(  connection,  query,  new TResultHandler<String>() {  public String handle(ResultSet  result) throws SQLExeption {  result.next();  return result.getString(  'user\_name');  }  });  System.out.append('User: ' + name + '\n'); |

Пример, который иллюстрирует то, что я сейчас говорю, именно со строками. То есть смотрите. Мы создаем Executor (1). Логично было сюда connection положить. Либо не создавать его, использовать статический. Не важно сейчас. У нас есть Executor. У нас есть строка (2). Строка, которая запрашивает по id из пользователей имя. Так как мы знаем, что нам должны вернуть строку, мы говорим, что будет строка на выходе (4), Executor типизирован строкой. Handler (7), который мы передаем сюда, тоже типизирован строкой. Все эти типы данных должны совпадать. Если мы захотим поменять запрос так, чтобы запрос возвращал не строку, а допустим long, мы должны будем во всех этих местах, где сейчас стоит String, написать long. Таким образом мы переключим Executor со строк на другой тип данных. И в handler (9) мы, как и раньше, получаем ResaultSet. Мы уже знаем, что именно с ним нужно делать. Нам надо с него строку получить. То есть мы в (12) должны получить строку. Мы эту строку получаем и возвращаем. Все. То есть мы с вами по результатам этого подхода организовали работу с базой таким образом что у нас появилось две функции. Одна execUpdate(), другая execQuery(). И больше нам функция для работы с базой не нужно. Мы меняем запросы. Создаем разные запросы по извлечению или по модификации данных и внутри execQuery происходит вся та черная работа по созданию statement, по обращению к базе, по закрытию statement. А мы можем просто execQuery вызывать и не беспокоиться о том, что внутри там написано. Просто мы должны ему передать нужный handler. Чтобы он нужные нам данные организовал в нужный нам тип данных. Если вернуться к теме про лямбды, которые мы с вами разбирали до этого особенности 8 Java и вы хотите уже у себя начинать использовать, то весь вот этот пример можно переписать на упрощенную форму записи.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | TExecutor execT = new TExecutor();  String query = "selest user\_name from users  where id 1";  String name execT.execQuery(  connection,  query,  result -> {  result.next();  return result.getString('user\_name');  }  );  System.out.append('User: ' + name + '\n'); |

Здесь компилятор понимает тип значения handler сам. То есть я задаю строку (4) и больше эту строку нигде не упоминаю. О том, что конструкция (7‑10) должна обработать ResaultSet и вернуть строку ясно только, если мы зайдем внутрь execQuery (4) и посмотрим, как именно она устроена. Давайте сейчас вернемся к предыдущему слайду, и у нас здесь есть то же самое место (7‑14). (7‑10) — это сокращенная форма записи, в которой я говорю, что result должен быть передан в качестве параметра в функцию. И сама эта функция (7‑10) является параметром и содержит тело следующего вида, которое я здесь написал. Ничего страшного не будет, если вы будете использовать у себя в работе предыдущий пример, если он вам покажется проще. Если вы легко переключитесь на новый вариант или уже готовы с функциональными подходами, то используйте вот этот вариант.

3.1.10 Транзакции

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Transactions  По умолчанию autocommit после выполнения каждого statement  void setAutoCommit(boolean autoCommit) — вкл/выкл автокоммита  void commit()  void rollback() |

Картинка

Рассказ про работу с базами и с Java будет не полным, если не упомянуть еще транзакции и Prepared Statements. Транзакции, как следует из названия, — это действие, которое могут быть выполнены либо целиком, атомарно, либо не выполнены совсем. То есть если мы какие‑то изменения должны сделать транзакционно, то это означает, что мы либо все эти действия применяем, либо у нас, если мы отменяем эти действия, то так же отменяем все полностью. Например, это может быть обмен предметами ваших пользователей друг с другом. Или продажа предметов одного пользователя другому. В этом случае ваши пользователи будут не рады, если по результатам обвала системы, например, у одного пользователя исчезнет предмет, а у другого он не появится. Или у кого‑то появится 2 предмета, а у кого‑то не появится. В любом случае, особенно там списаны деньги, а предмета не будет. При этом работа на изменение должна затрагивать сразу несколько таблиц в базе. То есть вам нужно не просто в одной базе чтоб переставить, а сразу в нескольких таблицах что‑то поменять при этом нужно сделать так, чтобы это было надежно. Для этого и нужны транзакции. Сразу скажу, что транзакции должны быть поддержаны в базе. Что если база транзакцию не поддерживает, то никакими ухищрениями JCBC вы этого не добьетесь. Поэтому я и призывал вас работать, по крайней мере в таких критически важных участках вашего приложения, именно с реляционными базами, потому что они транзакции поддерживают. Если смотреть на транзакционность с точки зрения нас, разработчиков программного обеспечения, использующих JDBC, то для нас важно понятие auto‑commit. auto‑commit — это режим, когда после каждого вашего statement на изменение база получает приказ на сохранение этого изменения на запись к себе. Либо залогировать как минимум у себя. По умолчанию auto‑commit включен. То есть когда вы по умолчанию используете JDBC, пишите statement на изменение, после выполнение каждого statement результата будет сохранен в базу. Если нам нужна транзакционность, то нам такой вариант не подходит, потому что нам нужно сделать несколько обращений к базе, возможно, к нескольким разным таблицам. И только после того, как мы выполнили последнее, мы должны сказать а теперь все вместе сохрани. Поэтому мы должны у себя setAuto‑commit выключить, то есть явным образом сказать auto‑commit больше не использовать. Если мы это сделали, то нам явным образом нужно будем по результатам всех наших действий вызвать явным образом commit(), то есть применить все изменения. Либо, если произошли какие‑то проблемы в процессе исполнения запросов, то мы должны cделать явным образом rollback(). Если мы сами его явным образом не сделаем, то база сама его через некоторое время сделает. Кроме того, есть такое понятие как Savepoint, когда мы сделали некое количество …, сохранили, если нужно откатили к SavePoint, если нужно откатились в самое начало.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | Transactions  public void execUpdate(Connection connection,  String[] updates) {  try {  connection.setAutoCommit(false);  for(String update: updates) {  Statement stmt = connection  .createStatement();  stmt.execute(update);  stmt.close();  }  connection.commit();  } catch (SQLException e)  try {  connection.rollback(true);  connection.setAutoCommit(true);  } catch (SQLException ignore) {}  } |

В виде примера это выглядит следующим образом. Представьте, что у вас есть некий набор строк с update (4). Вы должны первым делом вы выключить у connection setAutoCommit() (6). После этого вы можете в цикле создавать statement (8), выполнять statement (10) и закрывать их (11). Каждый раз на каждую строку. И по результатам после выхода из цикла взять и все вместе применить (13). Если по каким‑то причинам у вас что‑то пошло не так, вы можете у connection опять же вызвать rollback() (16), то есть все откатить. Вот и все. Выглядит оно странно в том виде, что нам надо каждый раз создавать новый statement на этот запрос. И чтобы этого не делать как раз следующий пример про Prepared Statements.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | Prepared Statements  public void execUpdate(Connection connection,  Map<Integer, String> idToName) {  try {  String update = 'insert into  users(id, user\_name)  values(?, ?)';  PreparedStatement stmt = connection  .prepareStatement(update);  for(Integer id: idToName.keySet()) {  stmt.setInt(1, id);  stmt.setString(2, idToName  .get(id));  stmt.executeUpdate();  }  stmt.close();  } catch (SQLException e)  e.printStackTrace();  } |

То есть с транзакциями разобрались, транзакции нужны для того, чтобы сохранять блог изменений атомарно, а Prepared Statements нам могу в этом случае помочь. Например, если нам нужно сохранить однотипные запросы целиком. Объединив этот пример и предыдущий, мы с вами получим следующее. Мы можем создать уже не statements, а Prepared Statements (9). Создается точно так же через обращение к connection и создание Prepared statements передачей в него строки с запросом. Но строка с запросом отличается от предыдущий тем, что в ней есть еще поля, отмеченные знаком вопроса (8). Те поля, которые будут заданы потом перед вызовом. В цикле мы можем проставлять поля (13‑14). Опять же обратите внимание, как в базах, у нас здесь единичка и двойка. То есть поля, нумерация полей начинается с единицы. Потом мы можем вызвать execteUpdate() (16). В текущем примере после execteUpdate() сразу пойдет запись в базу. Если мы объединим этот пример с предыдущим, выключим комит, накидаем изменения, потом сделаем их целиком, то мы много что сэкономим. Мы сэкономим на подготовке statement. У нас statements будет один. Мы не нужно каждый раз его закрывать. И на запись сразу пачку изменений в базу без необходимости сохранять каждый раз после каждого обращения.

Видеоинструкция

3.2 ORM и Hibernate

3.2.1 Аннотации

Картинка

Добрый день. Давайте начнем вторую часть занятия по базам данных. И если в первой мы с вами разбирали базовые возможности, которые вообще нам предоставлены в языке библиотеки для того, чтобы работать с базами, то в этой половине мы с вами разберем более высокоуровневые подходы, то есть разберем то, что делает нашу работу с базами удобнее и скрывает от нас детали внутренней реализации, то есть если прошлый раз мы с вами разобрали Executor, то сейчас этот Executor от нас будет спрятан внутри библиотеки, которую мы будем использовать. Начну я не совсем с баз данных, начну с аннотаций, потому что работа очень многих сторонних библиотек, работы которых основана на анализе вашего кода. Эта работа основана на восприятии аннотаций.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Аннотации — метаданные в коде  Содержат данные о программе, не являясь частью программы  Не влияют непосредственно на работу приложения  Влияют только на ту функциональность, которая их обрабатывает  Могут влиять на работу компилятора, инструментов и библиотек  "Decorating" or "wrapping" классы в runtime |

Для начало разберем что такое аннотации. Я думаю, все из вас знают, что код можно подписать комментарием, то есть вы можете в любом месте своего кода поставить два слеша и написать что‑нибудь поясняющее. При желании можно написать приложение, которое проанализирует ваш код и найдет все те места, которые вы пометили как комментарий. Если вы, как человек, который пишет комментарии, и человек, который пишет анализатор, договоритесь, что какие‑то виды ваших комментариев должны быть каким‑то образом интерпретированы, то программа, которая будет разбирать ваш код и искать комментарии, сможет этот ваш код поменять на основе ваших комментариев. То есть, например, вы можете создать функцию, у функции перечислить параметры, которые в функцию нужно передавать, и поставить перед переменной внутри функции комментарий, означающий что это переменная не должна быть null. А та часть, которая будет ваш код обрабатывать, другое приложение, она сможет найти этот ваш комментарий, понять что здесь должен быть не нулевая переменная и удалить комментарий, а вместо него в начало функции вставить проверку, что действительно переменная не null, а если null — выкинуть это все. Или какое‑то исключение, какое договоритесь. То есть вы таким образом получили возможность подписать свой код, расширить его возможности перед компиляцией или если у вас есть возможность перенести эти комментарии в runtime, то уже как‑то отреагировать на них в runtime. Это и есть аннотации. То есть аннотации — это такой удобный способ подписать некий функционал дополнительно, то есть подписать ваш код, сказать что тому, кому интересно, может увидеть здесь еще дополнительные возможности. Аннотации также как и комментарии, которые я упоминал, в чем‑то дополнительны к коду. За редким исключением наличие или отсутствие аннотации на компиляцию не влияет. Только есть некий набор аннотаций, который компилятору известны, специально для него написаны. Все остальные, тем более самописные аннотации, наличие или их отсутствие в коде также как наличие и отсутствия комментариев на компиляцию не влияют. И в общем‑то по большой части непосредственно на работу тоже не влияют, то есть вы можете подписать аннотациями, такими метакомментариями, свой код, запустить его, убрать, собрать запустить в общем‑то можно работать и так. Таким образом, добавляя аннотации, вы добавляете возможность другим библиотекам, либо вашим собственным, либо написанными кем‑то еще, разобрать ваш код в тот момент, когда вы его пишите, в тот момент, когда идет компиляция, и уже в runtime, если аннотации остаются в runtime. То есть вы таким образом передаете некую информацию, дополнительную к коду, тому, кому эта информация может быть интересна. Если она ему не интересна, он ее пропускает, если интересна, он каким‑то образом может на нее реагировать. И аннотации, в отличии от комментариев, могут повлиять на работу компилятора, то есть компилятор может обратить внимание на те или иные аннотации. Влияет на инструменты, которыми вы пользуетесь. Например, на среду разработки, то есть вы пишите код, а среда разработки следит за тем, есть ли аннотации на том, что вы написали, и что вы стороннее используете. Например, среда разработки может сказать вам, что эту функцию лучше не использовать, потому что разработчики библиотеки, из которой вы ее используете, считают, что эта функция уже устарела. То есть в процессе написания и в процессе компиляции, и еще в runtime, когда вы уже собрали приложение, передали его в Java‑машину, она его запустила, вы в своем приложении обратились к некой библиотеке, библиотека обратила внимание на ваш код и проанализировала его. Например, на ваши объекты. То есть вы ей говорите вот мои объекты или классы моих объектов, а библиотека уже начинает смотреть что же там за классы и видит в них аннотации. Получается такая некая дополнительная обертка для того, кому это нужно.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | Примеры аннотаций  Аннотация на класс  @Deprecated  class oldClass{...}  Аннотация на метод  @Test  public void unitTestSomething(){...}  Аннотация на поле  @Nullable  private Object object;  Аннотация на переменную  pubic int getUserId(@NotNull User user){...} |

Если перейти уже к практической части, то так могут выглядеть аннотации. Аннотации можно прицепить к классу. То есть вид аннотации. У вас есть нечто, на что вы хотите повесить аннотацию, вы пишите собаку, название аннотации, может в нее какие‑то параметры. Здесь у меня в примерах параметров нет. oldClass, который здесь представлен в самом начале, я на него повесил аннотацию @Deprecated. И это означает, что среда разработки если увидит, что я пытаюсь использовать этот класс, она мне скажет, что лучше его не использовать, потому что он @Deprecated. Кроме как аннотации на класс, мы можем повесить аннотацию на метод. Опять же вспоминайте пример с комментарием. Представьте, что я просто взял и написал комментарий Deprecated oldClass. И среда разработки точно так же, как она увидела аннотацию, мы с ней можем договориться, что она будет видеть мои комментарии. Получается нам не удобно, если делать все комментариями, то все в куче получается и комментарии для человека и комментарии для среды, и для компилятора. Поэтому решили, что давайте отдельно удобнее сделаем возможность подписать. И эта удобная возможность подписать — это и есть как раз аннотации. Кроме класса можно повесить на функцию. В данном случае функция тестовая. И эту аннотацию @Test увидит библиотека, которую вы запустите, чтобы протестировать свое приложение. Она просто посмотрит, что у класса есть методы, у этих методов есть аннотация @Test, а это значит, что можно эти функции запустить. В них произойдут какие‑то изменения, посмотреть на результаты, если результаты хорошие, то тест прошел. Можно повесить аннотацию на поле, то есть представьте, что у меня класс, в классе есть поле с именем object класса Object. И я могу сказать, что в этом классе этот object будет @Nullable. То есть будьте все осторожны, когда вы пытаетесь к нему обратиться, проверяйте на null, потому что я могу в конструкторе его не создать, а надеяться на то, что кто‑нибудь когда‑нибудь вызовет сеттер и проставит его. И можно повесить аннотацию на переменную. То есть если у вас есть функция, у этой функции есть список параметров, вы список параметров можете вставить в аннотацию. Пример, с которого я начал, @NotNull — это аннотация означает что пользователь не должен быть null.

3.2.2 Аннотации

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | @NotNull  Ссылка не может быть null  Для программиста — указание на то, что на null можно не проверять  Для среды разработки — подсветка присваивания null  В runtime — исключение в момент присваивания null |

Теперь давайте разберем как этот самый @NotNull у нас вообще работает. По задумке @NotNull нужен нам для того, чтобы показать, с одной стороны, разработчику, операционной системе, среде разработки и компилятору, что поле не null. То есть у вас есть класс, в классе есть поле. Вы прицепляете к нему аннотацию @NotNull. Таким образом вы всем, кто смотрит на код и человеку, и машине подсказываете, что это поле по задумке null не может быть. Точно так же может быть @NotNull тип возвращаемого значения. То есть функция обязана поработать и по результатам своей работы вернуть не null поле. Или это может быть переменная, в функцию которой вы перекладываете. И тот человек, который пишет функцию, он всем, кто эту функцию будет вызывать, подсказывает, что при вызове этой функции в качестве параметра null передавать нельзя. Задумка понятна. Вернусь к самому первому примеру. В данном случае использование аннотации похоже на использование комментария. В том только отличие, что у нас есть встроенный в компилятор и среду разработки проверка того, что то, что мы написали, оно действительно @NotNull. Некий такой получился оговоренный комментарий. Мы договорились со средой разработкой, договорились с компилятором, что если @NotNull, значит @NotNull. Как это может работать? То есть я везде расставил @NotNull. И я уверен, что у меня в функцию null не попадет. Но как это подкреплено в языке? В среде, в принципе, понятно. Среда разработки может статически проанализировать ваш код и увидеть, что я передаю в @NotNull функцию, и мне подсветить это место и сказать, что тут возможны ошибки. Компилятор в принципе тоже может понять, то есть он собирает также статически проанализирует. Но нам надо то на самом деле не только при разработке, и при компиляции. Нам еще хорошо бы в runtime все это дело анализировать. Вдруг если я допустил ошибки при сборке, то есть пропустил эти сообщения от среды разработки, от компилятора, оно соберется, кстати. Плюсы еще если я использую какую‑то библиотеку, а в библиотеки у меня уже собранное. То мне нужно бы в runtime тоже проверять, а в runtime проверять дорого.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Как работает @NotNull  Annotation preprocessors  Плагины для компилятора  Разбирают аннотации  Могут создать дополнительный код (в том числе новые классы)  Можно написать свои плагины |

И поэтому есть на стадии компиляции такая стадия, в которую мы запускаем Annotation preprocessors, то есть это плагины для компилятора. Их работа пробежать по коду, который вы собираете, проанализировать аннотации и для известных им аннотаций выполнить некие действия. То есть они могут неким образом модифицировать ваш код, то есть получается в момент сборки ваш код может быть поправлен preprocessors неким образом, который вы указали в аннотациях. В нашем случае, в случае @NotNull, annotation preprocessors пробегут по коду, увидят везде, где @NotNull. @NotNull можно будет при желании будет убрать, можно будет и оставить. Будет ли он в runtime в байт‑коде потом я имею в виду. В том месте везде, где @NotNull взять и вставить проверки, что допустим везде, где идет присваивание переменной, присваивание идет не @NotNull если переменный класс. Функция перед тем, как вернуть значение, может проверить null или не null возвращаемое значение. С параметром все вообще просто, просто в самое начало тело функции preprocessors может вставит проверку а null ли переменная, если переменная null, то взять и пропарсить в этом месте. Вы можете писать свои собственные плагины. Точно так же, как вы можете писать свои собственные аннотации. Об этом я буквально сейчас расскажу. Например, вы можете придумать аннотацию, которая будет проверять, что строка, которую вы передаете в функцию, не пустая. То есть стандартной аннотации на это нет. Но никто не мешает вам ее сделать. Вы пишите аннотацию, называете ее @NotEmpty, прицепляете эту аннотацию к строке, может быть в классе, может быть при передаваемых параметрах, и пишите плагин, который позволяет при компиляции проверить везде, где такие аннотации есть, поменять место, где эти аннотации, на некий код, который будет встроен во все те места, где вы работаете с этой строкой. И тогда получается вместо того, чтобы вам везде дублировать свой один и тот код проверки на пустую не пустую строку, вы получаете возможность просто расставлять удобную аннотацию везде короткую @NotEmpty. И вам удобно будет, и разработчикам другим. И можете среду каким‑то образом тоже через плагин настроить. И компилятору будет понятно, что с этим кодом делать. Пока что бонусы от аннотации на стадии разработки и компиляции очевидны. Какие есть еще бонусы — в следующих занятиях.

3.2.3 Аннотации

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Синтаксис аннотаций  Не может наследовать или быть базовым классом  Не содержит конструкторов и полей  Может содержать методы без переменных, которые работают как поля  @interface CreateBy {  String author();  String date();  }  @CreateBy (author = "tully", date =  "01.01.2014")  public class MyClass {...}  @CreateBy (author = "tully", date =  "01.19.2014")  public static void main(String args[]) |

Я уже говорил, что вы можете создать свои собственные аннотации. И самое время посмотреть, как это делать. На этом слайде внизу пример кода самой аннотации и использовании аннотации. Для того, чтобы создать свою собственную аннотацию, вы в отдельном файле задаете интерфейс с названием аннотации. Единственная разница по сравнению с обычным интерфейсом в том, что перед интерфейсом вы еще должны поставить собаку (8). То есть разработчики аннотации решили ничего нового не придумывать, сказали пусть будет собака, потом слово interface и название интерфейса. Выглядит все как обычный класс или обычный интерфейс, только вот еще собака. Никакого наследования в аннотации быть не может, ни сама не может наследовать, ни какого‑то другого, не нужно ему никакие конструкторы, никакие поля, в общем так же как у интерфейса. Но может содержать свои методы. Здесь начинаются странности. Всего я таких странностей знаю в Java два места: аннотации и Input/Output. До Input/Output еще дойдем, а в случае с аннотациями смотрите. В аннотации решили добавить поля. Но аннотация — это некая надстройка над интерфейсом. В интерфейсе полей нет, кроме статических. И чтобы добавить в интерфейс поля, решили что а давайте мы их напишем как будто они методы, но работать они будут как поля. То есть синтаксис (9) означает, что у меня есть аннотация, внутри аннотации есть поле. И поле это с именем author. Написали, давайте использовать. И мы можем куда хотим на классы, на методы, на поля ставить свою аннотацию, написать @CreateBy (12) и перечислять поля, то есть в скобочках после аннотации я могу перечислить поля через запятую и значения, заполняющие эти поля. В данном примере я создал аннотацию, которую можно повесить на некий участок кода. Аннотация подписывает того, кто и когда создал этот участок кода, то есть, например я написал некий класс, пометил его аннотаций, передал туда имя и дату. Дальше я хотел бы, чтобы эту информацию можно было получить в runtime, то есть, например, я запускаю код и хочу, чтобы запущенное приложение мне выдало список того, кто когда что сделал. Такая задача. И сделать это можно следующим образом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Обработка аннотации в runtime  Class myClass = MyClass.class;  Method method = myClass.getMethod("main");  CreateBy annotationC = myClass.getAnnotation(  CreateBy.class);  CteateBy annotetionM = method.getAnnotation(  CreateBy.class);  log.info("Autor of the class: " +  annotationC.author());  log.info("Date of Writing the class: " +  annotationC.data());  log.info("Author of the method: " +  annotationM.author());  log.info("Date of Writing the method: " +  annotationM.data()); |

У нас в runtime есть возможность получить доступ к аннотации. Предыдущий мой рассказ об аннотации был о том, что можно статическим анализом увидеть аннотации и повлиять как‑то на код. Кроме этого, есть еще возможность к аннотациям обратиться в runtime. Вы помните, что есть в Java понятие Class (3), то есть это класс, представляющий объект класса. И я в предыдущем примере уже создал класс с названием MyClass. Присваивание в этой строке (3) означает следующее. Я в runtime говорю дай мне пожалуйста в этом месте в переменную myClass запишем ссылку на объект, в объекте в этом будет информация о классе, то есть эта запись означает я хочу получить ссылку на объект с информацией о классе. И кроме того, когда я эту ссылку на объект получил, я могу у этого класса попросить а дай мне метод. Я знаю, что у тебя есть метод main(), дай мне пожалуйста ссылку на метод, то есть Method method (4) — это тоже библиотечный класс такой же как, простите, класс. И я могу получить ссылку на объект, представляющий метод. То есть объекты, приставляющие классы, в них хранится информация о классах. Что значит хранится информация? Значит она каким‑то образом представлена. Если у меня у касса есть несколько методов, то логично эту информацию представить в виде массива методов. Здесь именно это и происходит. У меня есть MyClass, в нем есть метод main() и я получаю ссылку на объект, который в runtime представляет этот самый main(). Я думаю вы уже догадались зачем мне это нужно. И у класса, и у метода я могу спросить аннотации. В runtime данные об этом есть. Эта информация записана в классе, записана в объекте метода. Поэтому я могу просто сказать дай мне аннотацию (6, 8). Либо я могу сказать дай мне все аннотации. Такой вызов тоже есть. И после этого я, получив аннотации, могу взять и в логе прописать все то, что получил. Я могу проанализировать все свои классы, со всеми своими методами в runtime и узнать нужную мне информацию. Мы с вами встретимся уже в следующей части занятий с тестированием, с Unit‑тесты. Unit‑тесты тесты мы будем запускать, используя JUnit. И одна из аннотаций, которая нам поможет в этом будет аннотация @Test. Я ее сегодня даже показывал. И эта аннотация подскажет библиотеки Junit, что есть некий набор классов, в классах есть методы, эти методы, помеченные аннотацией @Test нужно запустить как тесты. То есть произойдет в точности то, что я здесь показал, но только не по отношению к моим собственным аннотациям, а по отношению к стандартным, библиотечным аннотациям @Test. Что такое будет @Test? @Test — это будет обычный класс, в классе будет некий набор методов, часть из них будет помечена аннотацией @Test. И когда я захочу запустить тестирование, я ничего дополнительного в класс добавлять не буду. Просто библиотека, которая захочет протестировать, посмотрит список методов, в точности так же, как на слайде. И увидит где какие методы с аннотацией @Test и запустит их.

3.2.4 Аннотации

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | Виды аннотаций  RetentionPolice.SOURSE — аннотации присутствуют только в коде  Информация для компилятора  Информация для инструментов IDE  RetеntionPolice.CLASS — сохранены в .class, но не доступны в runtime  Доступны при анализе byte кода  Не доступны через reflection  RetеntionPolice.RUNTIME — сохранены в .class и доступны в runtime  Аннотации можно получать из getClass() через reflection |

Я уже несколько раз говорил вам, что аннотации можно проанализировать во время написания кода, среда разработки может проанализировать. В принципе, вы сами, когда видите какую‑то аннотацию, уже тоже можете понимать в чем здесь дело, что происходит именно в этом участке кода. Анализ человека, анализ машины, анализ среды разработки, анализ компилятора и еще возможность получить доступ в runtime. Чтобы каким‑то образом это все упорядочить придумали виды аннотаций, то есть три стандартных вида аннотаций. По тому, откуда они видны, то есть их решили так разделить в три группы. И назвали все это RetentionPolice. Это enum, у него три значения SOURCE, CLASS и RUNTIME. (3) означает, что аннотация будет доступна только при анализе исходного кода. В байт коде ее не будет совсем. То есть она просто туда не попадет. Если она вам там не нужна, если вы не собираетесь аннотацию получать в процессе работы приложения, то делайте (3) и все.

Есть (8). Это стандартный по умолчанию RetentionPolice. Это означает, что запись об аннотации попадет в байт‑код, но в runtime доступна не будет опять же. То есть запись о ней будет, но получить ее через рефлекшн у класса будет невозможно. Дискуссия у нас была о том, зачем эта вообще аннотация CLASS нужна, сошлись на том, что в общем полезно будет для безопасности. Вы подписываете свой код, в байт‑коде он есть. Но на производительность, на работу он не влияет.

И есть еще (12). Аннотация этого типа означает, что вы где угодно можете к этой аннотации получить доступ. Вы можете естественно ее увидеть в исходниках. Она есть в байт‑коде, и она будет еще в run time. Через reflection можно получить класс или метод и спросить дай аннотации и там все (12) у вас списки будут. Теперь вопрос внимательным слушателям разделам про аннотации. На лекциях я в этом месте даже делаю некую паузу. Вам предлагаю перед тем, как я перейду на следующий слайд, тоже остановить видео и предложить свое решение. Есть у нас аннотации, мы можем написать новые аннотации, у нас уже есть библиотечная аннотации, и они разбиты в три группы. И нам нужно при создании своей аннотации, либо обратившись к исходнику уже существующей аннотации, понять какого она типа, то есть нам нужно указать какого типа аннотация. Вы пишите аннотацию, вам нужно указать какого она типа. Как бы вы указали тип аннотации. Здесь такая пауза. И если вы этой паузой воспользовались, подумали, у вас есть свое решение, решение в действительности следующее.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Аннотации аннотаций  @Retention — аннотация для указания типа аннотации  @Retention(RetentionPolice.RUNTIME)  @interface CreateBy {  String author();  String date();  } |

RetentionPolice мы передаем через аннотацию, то есть аннотация нужна для того, чтобы подписать функциональность. У нас есть аннотация, мы хотим ее подписать, мы для нее используем аннотацию. Пример того, как мы ее используем. То есть у нас есть аннотация @Retention. Она на вход берет параметр RetentionPolice. Если переменная всего одна в аннотации, то не нужно указывать ее имя, просто пишите ее значение. Это как здесь, то есть @Retention, у нее есть одно поле, видимо retention. И мы передаем RetentionPolice.RUNTIME. Сама RetentionPolice.RUNTIME она RetentionPolice.RUNTIME. Так что все нормально. Там бесконечный цикл не появляется. CreateBy в моем случае будет видна во время исполнения приложения тоже, потому что я ей указал RetentionPolice.RUNTIME.

3.2.5 ORM

Вернемся к разбору особенностей работы с базой данных из Java. И перед тем, как перейти к библиотеке, про которую я и хочу вам рассказать, остановимся на том что такое ORM. Некоторые слова я уже говорил в начале занятия об этом. Но мне хотелось бы подробнее это место еще обговорить, чтобы было понятно какие бонусы мы получаем с вами.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Object-Relation Mapping  Связь между объектной моделью программы и реляционной моделью базы  Примеры  Метод DAO принимают объекты и сохраняют их в базе  Методы DAO возвращают объекты собирая их на основе запросов |

Картинка

У нас есть приложение, написанное с использование объектов с одной стороны и базы данных с другой стороны. У базы данных нет объектов. Там таблицы. У нас в приложении нет таблиц, у нас объекты. ORM — это то, что позволяет нам видеть из приложения базу как будто там объекты, а для базы приложение поставляет запросы к реляционным схемам. То есть ORM перекладывает объекты в записи в таблицах и на основе записей из таблиц может собрать объекты. И для этого мы будем элементы ORM, о которых я сейчас рассказываю, это DAO и DataSet. DAO — это Data Access Object. DataSet — это набор данных. DAO — это центральный элемент ORM, потому что задача DAO взять от вас объекты и превратить его в запрос к базе, либо предоставить разработчику функцию, при вызове которой внутри себя DAO создаст запрос к базе, а разработчику отдаст объекты, то есть это некий дополнительный слой абстракции, позволяющий всем тем, кто работает с базой вместо того, чтобы писать запросы производить вызов методов у объектов. То есть вы хотите получить некую запись от базы, вы результат этого запроса оформляете как DataSet, запрашиваете DAO дай мне данные уже не просто в виде таблицы, не просто в виде ResaltSet, о котором мы раньше говорил, а дай мне данные в виде списка DataSet и получаете на выход объекты, с которыми потом каким‑то образом работать. Либо у вас есть уже объекты, и вы передаете их в DAO и DAO их превращает в записи.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Java Persistence API  Package javax.persistence  Аннотации  @Entity: объект класса можно «переложить» в таблицу  @Table: связывает класс и таблицу  @Id: поле является первичным ключом в таблице  @Column: связывает поле и колонку в таблице |

Почему мы до этого разбирали аннотации. Потому что, чтобы все это работало, в Java есть раздел javax.persistence. Это часть стандартной библиотеки, то есть разработчики стандартной библиотеки позаботились о том, чтобы предоставить возможность разработчикам ORM решений и пользователям ORM решений использовать единую систему аннотаций. На этом слайде я привел основные аннотации из этого списка. Если вы хотите, чтобы какой‑то объект был понятен ORM системе как объект, который можно переложить в таблицу, вы обозначаете его как @Entity, потом аннотации @Table можете передать его конкретной таблице и разметить поля, которые в этом классе есть, пометить их как колонкой или как специальная колонка с аннотацией id, если вы хотите, чтобы эта колонка была первичным ключом, то есть вы не работаете с базой запросами, вы работаете с базой через создание классов со специальной разметкой. Сообщаете библиотеке о том, что у вас есть эти классы и при запросе дай мне данные из таблицы вы на выход получаете объекты этих классов.

3.2.6 DataSet

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | DataSet  DataSet — объект, содержащий данные одной строки таблицы  На каждую таблицу свой DataSet  Извлечение и вставку данных удобно проводить через DataSet  В терминах JPA DataSet это Entity |

Подробнее разберем что такое DataSet и что такое DAO. ORM подход к работе с базой он кроме бонусов, которые сразу вам предоставляет, что вы видите результаты запросов как объекты и отправляете на запрос тоже объекты, он дает некие ограничения. Ограничения в том смысле, что без ORM вы можете спросить какой‑то участок таблицы, ограничив его набором нужных вам колонок вплоть до того, чтобы извлечь одну колонку. Получить на выход некую абстрактную структуру данных типа таблицы или как ResaltSet сделан и кастомным образом с ней поработать. В случае ORM не принято писать на каждый особенный тип запроса свой класс, то есть обычно делают таблицу в базе с некоторым количеством колонок, 10 колонок. И для каждой таблицы создают один класс, представляющий одну строку в таблице. В этом классе будут перечислены все поля, если у нас там 10 колонок, то в DataSet будет 10 полей с именами, совпадающими с теми, что в таблице. И с типами, совпадающими теми, что в таблице. То есть получается что когда вы запрашиваете дай мне записи. Вам нужно только дни рождения пользователей. В этой таблице у вас и дни рождения, и последнее время входа в систему, и может быть очки какие‑нибудь еще набранные, имена. Вы все равно запрашиваете целиком все строки, то есть вы говорите дай мне запись пользователей, вам на самом деле нужно только имя пользователя или только день его рождения, но тем не менее вы запрашиваете целиком всю строку. То есть это некая избыточность, но при проектировании идут на эту жертву, потому что в целом работать становится удобнее. И запрашивают у базы из таблиц целиком строки. То есть мне нужен пользователь, зная его id запрошу целиком строку и эту строку получу в виде DataSet, то есть почему так получилось. Нам нужно при запросе из приложения получить на выход объект. Объект — это некая структура данных с заранее заданным количеством полей. В Java по крайней мере так. Если вы в каждом запросе будете менять формат выдачи данных, то вам нужно будет каждый раз новый этот Set придумывать. Так никто не делает. Поэтому давайте просто такое ограничение сразу при использовании ORM. DataSet — это объект, в классе которого перечислены поля и эти поля соответствуют данным одной строки таблицы. Соответствуют точнее одной строке таблицы, каждое поле соответствует своей какой‑то колонке. На каждую таблицу вы заводите DataSet. Если вам нужно что‑то у таблицы спросить, вы у объекта DAO, который сейчас будем обсуждать, спрашиваете дай мне DataSet, соответствующий пользователю, которого мы ищем. И тогда, если нам нужно новую вставку сделать в таблицу, то мы создаем DataSet, заполняем все поля, которые нам нужны, кроме генерируемых таблицей, если такие есть, и передаем в DAO объект DataSet и говорим сохранить. И она ее сохраняет. То есть мы при работе с базой из приложения вызовем объекты DAO, либо получаем от них DataSet, либо передаем в них DataSet. Это удобно, потому что мы работаем с базой через объекты, но минус в таком подходе в том, что нам приходиться как‑то ограничивать себя. Мы не можем в DAO прописать какой‑то любой абстрактный запрос, который не пойми что нам вернет. Он обязан нам вернуть либо DataSet, либо набор DataSet, либо ничего, если ничего не нашел.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | DataSet  public class UsersDataSet {  private long id;  private String name;  public UsersDataSet(long id,  String name) {  this.id = id;  this.name = name;  }  public UsersDataSet(String name) {  this.id = -1;  this.name = name;  }  public String getName() {  return name;  }  public long getId() {  return id;  }  } |

Как может выглядеть DataSet. Это обычный класс, в нем список полей. И в нем еще должен быть конструктор, который позволит нам этот DataSet создать и гетеры у этих полей. Ничего особенно сложного. Чтобы ORM системы понимали DataSet, его нужно будет разметить аннотациями, но об этом чуть позже в разделе про hibernate я покажу просто как этот DataSet нужно разметить, чтобы hibernate его понял. Пока что мы просто считаем, что у нас есть просто объект, и каждому объекту соответствует запись в таблице.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | DAO  Объект доступа к данным  Шаблон проектирования, скрывающий детали работы с базой  Обычно один DAO на одну таблицу  Высокоуровневый доступ к данным через DataSet  Варианты операций над базой:  Вставки строки — добавление DataSet  Поиск строки по ключу — возврат DataSet  Поиск строки по признаку — возврат List<DataSet>  Удаление строки |

DAO в свою очередь — это тоже объект. Тот, который скрывает от нас работу с базой. Вообще говоря, DAO не подразумевает именно работу с базой. Это сокрытие работы с данными. Объект, позволяющий получить доступ к данным. А что там внутри таблицы на самом деле для DAO не так важно. Обычно, когда говорят DAO, подразумевают работу именно с базами. У класса этого объекта есть набор методов, который позволяет нам обращать к этим методам передать туда DataSet, либо получить от туда DataSet. То есть мы можем написать методы. Например, метод, который на вход получает id пользователя, а на выход дает DataSet пользователя. Или метод, который на вход получает временной интервал с какого по какой момент времени, а на выход возвращает List<DataSet> всех пользователей, который в этот момент заходили в приложение. Обычно DAO делает отдельные таблицы. На самом деле это не обязательно. В том приложении, о котором я вам рассказываю, я одному DAO завожу на таблицу. Часто под DAO подразумевают еще более высокого уровня систему, которая вообще ко всем таблицам закрывает доступ. Например, что можно сделать с базой — можно вставлять строки, то есть добавлять DataSet, можно искать по ключу, возвращать, искать по признакам и можно удалять ее. И выглядеть это может следующим образом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | interface UsersDAO  public interface UsersDAO {  UsersDataSet get(long id)  throws SQLException;  UsersDataSet getByName(String name)  throws SQLExeption;  void add(UsersDataSet dataSet)  throws SQLExeption;  void dalete(long id) throws SQLExeption;  } |

Я здесь привел интерфейс. Не саму реализацию, а просто список функций. Представим, что у нас есть таблица с названием Users. Мы для этой таблицы заводим UsersDataSet и UsersDAO. И в UsersDAO перечисляем методы, которые позволят нам работать с таблицей через создание или добавку DataSet. У меня приложение некое работает. Пользователь хочет авторизоваться, присылает свой логин и пароль. Я должен запросить базу, чтобы этот логин пароль извлечь. Что я делаю. Я обращаюсь к UsersDAO и говорю getByName() и передаю туда имя, то есть логин, который мне пользователь передал. На выход я получаю UsersDataSet. В UsersDataSet у меня описано имя, время регистрации, пароль, емайл и еще какие‑то поля. Я проверяю, что пароли совпадают. И говорю, что пользователь авторизовался. После того, как этот блок работы с базой написан, дальнейшая работа с базой сводится не к написанию запросов, а к обращению к обертке, к объекту, который скрывает от нас непосредственно уже работу с базой через запрос.

3.2.7 Hibernate

Перейдем к содержательной части занятия. Мы всячески готовились до этого к тому, чтобы понять как работает и из чего состоит библиотека Hibernate. Библиотека Hibernate — это реализация принципов ORM. Плюс еще некие дополнительные особенности, о которых я сейчас вам и расскажу.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | ORM библиотека для Java  <dependencies>  <dependency>  <groupId>mysql</groupId>  <artefactId>mysql connector java  </artefactId>  <version>5.1.35</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artefactId>hibernate core  </artefactId>  <version>4.3.8.Final</version>  </dependency>  </dependencies> |

Перед тем как начать Hibernate использовать нам надо с вами его выкачать. То есть у нас, во‑первых, уже должна быть зависимость в pom файле на драйвер к базе, с которой вы собираетесь работать. И заодно еще туда надо будет добавить зависимость на библиотеку. Еще раз обращаю ваше внимание, что проверяйте версии. Приложения развиваются, библиотеки тоже, поэтому версии у вас могут быть новые, другие. Единственное, что я со своей стороны хочу сказать именно про эти версии, что с этими версиями я проверил, с ними работает. С теми, которые у вас будут, у вас может и не заработать. Если что вдруг читайте почему или возвращайтесь к моим версиям. Первое, что надо сделать перед тем как с Hibernate работать — это выкачать сам Hibernate себе через постановку зависимостей <dependencies> …

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | Configuration  Configuration configuration = new org  .hibernae.cfn.Configuration();  configuration.setProperty(propertyName,  propertyValue); | |
| propertyName  hibernate.dialect  hibernate.commection  .driver\_class  hibernate.connection  .url  hibernate.connection  .username  hibernate.connection  .password  hibernate.show\_sql  hibernate.hbm2ddl.auto | propertyValue  org/hibernate.dialect  .MySQLDialect  com.mysql.jdbc.Driver  jdbc:mysql://  localhost:3306/  db\_example  tully  tully  true  update |

После того как вы его выкачаете вам нужно его настроить. Настройка похожа на настройку работы базы через jdbc. То есть вам тоже нужно будет включить некий набор параметров. Если вы вернетесь к тому месту, где мы конфигурировали работу с JDBC, увидите, что многие из них совпадают. В случае с hibernate нам нужно создать объект конфигурации. Здесь я выписал специально полный путь конфигурации (3), потому что в библиотеке Java и вообще в тех библиотеках, которые вы скачиваете, очень много классов с названием конфигурация. Чтобы на всякий случай не запутаться, я вам написал полное название класса конфигурации. И после того, как вы его у себя его создадите, в него нужно прописать property, чтобы дать понять hibernate с чем и как именно мы будем работать. И вы можете посмотреть на список параметров. Часть из этих параметров вам должно быть уже знакома, то есть понятно что нет смысла особенно пояснять что такое URL, потому что URL мы уже разобрали, username, userparol и драйвер тоже уже более менее должно быть понятно. (10‑17) — с ними все понятно, они точно такие же как у JDBC, никакой разницы нет. Но есть еще здесь три поля дополнительных, которые являются особенностью hibernate. Первый из них — диалект. Мы с вами уже указали драйвер. То есть, казалось бы, драйвер мы уже передали, hibernate знает, что мы работаем с MySQL, зачем еще нам диалект? Диалект — это подсказка hibernate как именно формировать SQL запросы. Несмотря на то, что SQL стандартный, должен быть всеми базами одинаково понимаем, между разными базами все таки есть разница в том, как именно SQL‑запросы писать. Диалект — это подсказка hibernate как именно работать с той базой, которую передаем. Я не проверял, но по всей видимости диалектов работы с MySQL может быть несколько, иначе было бы бессмысленно передавать драйвер и еще заодно диалект. Тем не менее MySQL диалект как в том виде, в котором он здесь представлен точно есть, вы можете просто зайти в список диалектов и посмотреть что там происходит. Если вы его не укажете, то hibernate будет писать некий сайв тип запросов, стараясь никак не использовать особенности базы, потому что он не знает какая там, то есть никак не оптимизирует свои запросы, никак не использовать особенности, если он не знает как это делать. Дальше у hibernate есть параметр show\_sql. Это очень удобный параметр для отладки. В примере у меня проставлен true, вы можете у себя тоже проставить true в процессе написания. Во время работы лучше убрать false, потому что если у вас выставлен true у вас в логах будут записи о том, какие именно запросы к базе делает hibernate. Посмотрите, это очень интересно на самом деле, то есть hibernate скрывает от вас запрос. Вы ему поставляете объекты, он эти объекты превращает в запросы, отправляет в базу. И наоборот вы у него спрашиваете объекты, он понимает по тому, что вы спрашиваете какой запрос сделать, делает запросы, собирает данные, получает тот же самый ResaltSet. Собирает на основе ResaltSet объекты и вам возвращает, собственно, ORM. Задача такая. Кроме того, он может вам в лог при записях еще писать SQL‑запросы, который он для этих запросов вам создал, то есть вы увидите какие запросы создают машина для того, чтобы сделать те запросы объектные, которые вы к ней делаете. И последний параметр здесь hbm2ddl.auto — это очень интересный параметр. Про него отдельный целый слайд.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | Hibernate.hbm2ddl.auto  Автоматически создает или проверяет схему базы при создании SessionFactory  validate: проверяет схему не внося изменений  update: обновляет схему, если находит различия  create: пересоздает схему  create drop: уничтожает схему при закрытии SessionFactory |

Значит hibernate при старте вы сообщаете о своих DataSet, то есть вы написали приложение, создали DataSet, разметили DataSet, покажу пример ниже, и сообщаете hibernate вот у меня есть некий набор DataSet. На основе этих DataSet Hibernate понимает какие именно таблицы этим DataSet соответствует. Каждому DataSet соответствует своя таблица. Каждому полю в DataSet соответствует колонка. И Hibernate всю эту информацию от вас получает, то есть вы ему сообщаете классы, в классы помечены аннотации, мы вспоминаем что такое аннотации, понимаем что hibernate анализирует ваши классы, извлекает аннотации, по аннотациям понимает как работать с этими классами. И более того он на основе этих классов понимает какая структура таблиц должна быть в базе. Так вот перед тем как начать работать у hibernate есть возможность проверить а существующая в базе схема соответствует ли тем DataSet, которые вы в hibernate передали. Потому что если она не соответствует, то он запросы сделает, но база их не выполнит. Потому что там структура может быть другая. Либо в базе может быть вообще пусто. То есть базу вы создали, таблиц в ней нет. Вы сообщаете в hibernate вот у меня есть некий набор DataSet, каждому набору DataSet таблица, hibernate смотрит в базу, таблиц нет. Так вот, у hibernate есть некий набор реакций на разные ситуации соответствия или несоответствия DataSet, которые вы ему дали, и таблиц, которые он нашел в базе. На этом слайде я выписал возможные варианты поведения hibernate. И именно их мы задаем параметром hbm2ddl.auto. Первый — это самый простой способ validate. Если hibernate работает в режиме validate, то при старте, при создании коннекта к базе hibernate проверит есть ли в базе все нужные таблицы и вообще правильная ли вообще зависимость между этими таблицами, все ли там соответствует тем DataSet, которые hibernate получил на вход перед установки коннекта, от вас он получил на вход. Если схема соответствует, то с ней дальше работает. Если не соответствует, то выкидывает вам исключения, говорит не будет с ней работать. Следующий вариант его поведения — это update. Все то же самое, что в случае с validate, только в том случае, если он находит отличие, например, нет таблицы, или видит, что в DataSet, который ему пришел, полей стало больше. Разработчик добавил еще несколько полей, а в таблице полей нет, то hibernate попытается эту схему улучшить, модифицировать так, чтобы схема в базе соответствовала классам объектам в коде. То есть он добавит необходимые колонки в таблицу или создаст их, если их там нет. Вот это режим update и есть еще два режима create и create‑drop. Create означает, что не важно что там было в базе, при присоединении мы полностью все пересоздаем. Там все дропаем, ничего там не остается и создаем заново. То есть первым делом при установке соединения hibernate смотрит, если там что‑нибудь, если там что‑нибудь, он все удаляет. Создает новую схему, полностью соответствующей тем DataSet, которым передали в него. И create‑drop — это не просто пересоздать, а еще и почистить за собой, если при нормальном завершении приложения никаких исключительных ситуаций не случилось. При нормальном завершении приложения почистить там все за собой, просто взять и все убрать.

3.2.8 Session Factory

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | Session Factory  Фабрика, которая создает сессии  Одна фабрика на поток  Одна сессия на запрос |

Картинка

После того, как вы настроили hibernate, разобрались с тем, как он будет создавать вам схему, установили соединение, вам нужно начать работать с hibernate, точнее даже вы еще не установили соединение, вы пока все настроили и проверили, что все может работать. И дальше вам нужно создать Session Factory. Как следует из названия — это фабрика, задача которой будет производить вам сессии. А сессия — это юнит, позволяющий вам делать запросы к базе. Создание Session Factory оно тяжелое, то есть процесс, требующий неких затрат. И не нужно пересоздавать сессию каждый раз. То есть по‑хорошему сессию у нас должна быть одна фабрика на поток вашего приложения. В сессии, в которой фабрика будет создавать, они легкие, создание их потокобезопасное. И вы можете их создавать на каждый запрос, который вам нужен, то есть Session Factory такая большая фабрика по производству сессии, сессии относительно легкие. Задачи к базе, которые из Session Factory можно производить, и к сессии обращаться непосредственно уже к базе.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | Session Factory  StandartServiceRegistryBuilder builder =  new StandartServiceRegistryBuilder();  builder.applySetting(configuration  .getProperties());  ServiceRegistry serviceRegistry =  builder.buildServiceRegistry();  SessionFactory sessionFactory = configuration  .buildSessionFactory(  serviceRegistry);  Session session = sessionFactory  .openSession();  Transaction transaction = session  .biginTransaction();  System.out.append(transaction  .getLocalStatus().toString());  Session.close();  sessionFactory.close(); |

Раньше инициализация фабрики в версиях hibernate была проще, теперь нужно создать много особенных билдеров. Есть ServiceRegistryBuilder, нужен сделать. Потом у регистриБилдера нужно попросить ServiceRegistry, создать sessionFactory, построить sessionFactory на основе конфигурации и ServiceRegistry. То есть (3‑10) я предлагаю вам скопировать себе в то место, где вы будете с hibernate работать. Те, кому интересно, разберите его подробнее. Зачем же именно вот так было сделано. Раньше можно было sessionFactory по конфигурации сделать. Потом решили, что так deprecated. Решили, что (3‑10) лучше. Больше абстракции, больше фабрик, больше возможностей, модификации. Наверное, они это хотели сделать, разработчики. После того, как вы sessionFactory сделали, вы можете у нее запросить openSession() (14). Вы у сессии можете начать транзакции (16). В моем примере я ничего особенного с базой не делал, я просто данные транзакции запрашиваю. По‑хорошему в (21) должны быть запросы на извлечение, вставку. Я покажу пример как именно это будет выглядеть. И после того, как вы со всем этим поработали, вы можете закрыть сессию (22). Если вы закончили работать и с фабрикой тоже, то закрыть и фабрику тоже (23).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. | Аннотации для DataSet  import javax.persistence.\*;  @Entity  @Table(name = 'users')  public class UserDataService {  @Id  @Column(name='id');  @GeneratedValue(strategy = GenerationType  .IDENTITY)  private long id;  @Column(name = 'name')  private String name;  ...  }  Перед созданием SessionFactory:  Configuration.addAnnotatedClass(UserDataSet  .class); |

В этом же разделе я еще раз хочу вернуться к аннотациям и к DataSet. Помните, у нас была DataSet, UserDataSet. (7) — это он же. Это класс, в котором мы перечислившем поля, соответствующим колонкам таблицы. То есть у нас есть таблица с пользователем и мы ее в соответствии ставим класс. Чтобы передать его в hibernate, мы должны перед созданием фабрики в конфигурацию передать все классы всех DataSet, которых мы хотим использовать. И перед тем, как его передать в коде разметить. Мы должны из javax.persistence его разметить как @Entity (5), сказать, что какой таблице он соответствует, отметить колонки, которые есть в @Entity специальной аннотацией и в моем примере еще есть поле id и стратегия как это id генерировать. Если у вас несколько разных DataSet, вы все эти DataSet должны передать в конфигурацию через вызов addAnnotatedClass() (20). После этого создаете уже фабрику и можете обращаться к базе.

3.2.9 Session

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Session  Основной интерфейс между приложением и библиотекой  Время жизни сессии соответствует времени жизни транзакции  Задача сессии — работа с объектами проанатированными как @Entity |

Session Factory мы сделали. Для этого мы сконфигурировали, мы установили соединение, в момент создания SessionFactory соединение с базой есть. Нам нужно к базе написать запросы. И вот запросы происходят через сессии. Это некий аналог statement или же JDBC. Если так проводить параллели — не совсем точно, но можно так сказать. Там у вас был connection, вы у него получили statement. Здесь у вас Session Factory, вы получаете сессию. На каждый запрос вы создаете по сессии. То есть работа с сессиями можно организовать через транзакции. То есть вы начинаете транзакцию сессии, заканчиваете транзакцию сессии. Задача сессии — это получать, точнее в коде в объект сессии мы передадим запрос, а в момент исполнения этого запроса в сессию придет ответ. И ответ нам будет сформирован уже в виде объекта, содержащего @Entity. Если вы вспомните Executor, то вот здесь Executor спрятан внутри сессии.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | Session  public void save(UserDataSet dataset) {  Session session = sessionFactory  .openSession();  Transaction trx =  session.beginTransaction();  session.save(dataSet);  trx.commit();  session.close();  }  public UserDataSet read(long id) {  Session session = sessionFactory.  openSession();  return (UserDataSet)session.load(  UserDataSet.class, id);  } |

Например, DAO, которое работает с UserDataSet, может выглядеть следующим образом. Вы можете в ней создавать сессию (4), начинать транзакцию (6), передавать в сессию просто вызовом у сессии метода save() (8) объект DataSet. То есть hibernate сам понимает, что это именно UserDataSet. Ему не нужно сообщать класс. Класс у него уже есть. Вы в конфигурации этот класс ему передали. Вы просто передаете объект. И говорите транзакщнКоммит (9). В этот момент, в save() уже полетит запрос к базе. А в транзакшнКоммит этот запрос будет уже сохранен самой базой. И после этого вы закрываете сессию (10). То есть таким образом в этом участке кода ни строки не записано про запрос. В запрос, который будет отправлен к базе, он полностью на совести hibernate. Вы в DAO передаете объект. Вы в сессию передаете этот объект, в функцию save() сессии. Внутри функции save() hibernate берет классы в объекты, понимает какая этому классу соседствует таблица. На основе данных полей, объекты которого мы передали, формирует запрос на вставку данных в эту таблицу и отправляет этот запрос в базу. И примерно то же самое происходит при чтении. При чтении мы допустим хотим извлечь запись по идентификатору. У нас в DataSet есть id. Мы хотим его извлечь. Здесь мы точно так же создаем сессию. Здесь нам уже никакие транзакции не нужны, потому что мы же извлекаем данные. И единственное что мы вызываем у сессии load(), но в этот load() мы должны передать класс, который мы хотим прочитать. Что здесь происходит. Мы создали сессию, в сессию передали класс, который мы хотим прочитать, объекты, которые мы хотим прочитать. И идентификатор записи таблицы, по которому мы хотим искать эту запись. В функции load() hibernate, получив на вход класс, ищет какая таблица соответствует этому классу, обращается к этой таблицы с запросом, в котором указан идентификатор, получает ResaltSet. ResaltSet разбирает ResaltSet, превращая его в UserDataSet. В наш же тип, не в стандартный, в наш собственный и нам его возвращает. Мы потом из UserDataSet можем его уже передать дальше в наше приложение, где хотим использовать. Не написав ни одного sql запроса здесь, мы получили возможность сохранять и читать данные. Естественно, возникает вопрос как быть в случае с запросами более сложными. Например, поиск не по id, а поиск по имени. Или селект какого‑то набора данных. Об этом в следящей части в HQL.

3.2.10 HQL

Продолжаем тему запросов к базе через hibernate. Самое время рассказать про HQL. То есть это такой язык запросов, с помощью которого мы формируем задачу для hibernate. И если SQL — это язык, в котором мы запросы формируем строкой, то в HQL мы запросы формируем объектами, то есть мы формируем запрос, составляя запрос вызовами методами разных объектов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | HQL  Hibernate Query Language  «Язык» написания запросов «к hibernate»  Вместо написания запросов к базе — описание запроса вызовами методов сессии  Ссылка  Ссылка |

Давайте посмотрим как это происходит. Вместо того, чтобы писать запросы к базе напрямую, мы можем составить в коде последовательность вызовов методов объектов так, чтобы hibernate понял, что именно мы от него хотим. Вот это как раз и есть HQL. Я сейчас сделаю только краткий обзор основных возможностей. За деталями обращайтесь к документации. Этот минимум, который я нашел. Дальше по ключевым словам вы сможете найти уже все, что вам нужно будет. У hibernate есть режим, когда вы делаете запросы над строкой. То есть вы говорите хочу обратиться к базе числам SQL. Делать стоит в случае крайней необходимости, потому что ни диалекты вы таким образом не используете, ни кеширования hibernate, ничего. То есть могут некие странные … эффекты, если вы пишите прямые SQL запросы вместо того, чтобы использовать hibernate язык.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | HQL вставка  public void save(UserDataSet dataset) {  Session session = sessionFactory  .openSession();  Transaction trx = session  .beginTransaction();  session.save(dataSet);  trx.commit();  session.close();  } |

Пример с сохранением я вам уже показывал. Вот он еще раз. И если вам нужно сохранить DataSet, то вы передаете этот dataSet в функцию session save(). И hibernate сам понимает, как ее сохранять. То есть это один из самых простых примеров. Что именно и как сохранять hibernate понимает внутри себя, то есть save() (8) — это уже пример HQL. Потому что мы обратились к базе, сказали вставить данные через вызов метода save() сессии и передачи туда нужного нам dataSet.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | HQL чтение  public UserDataSet read(long id) {  Session session = sessionFactory  .openSession();  return (UserDataSet) session.load(  UserDataSet.class, id);  } |

Если мы хотим прочитать DataSet, то мы должны точно так же сформировать запрос через вызов метода сессии. Но в этом случае метод уже будет load(). Мы в него передадим класс того, что мы хотим загрузить и идентификатор того, где искать (7). То есть это тоже пример HQL. И обратите внимание, что на выход (6) вам сессия вернет объект просто object. Тип базовый для всех объектов, всех классов. И вы сами должны его скастить к тому, что вы передали (7). По‑хорошему hibernate должен вторым … первым параметром в load() понимать, то есть надо было бы, мне кажется, здесь бы хорошо подошла типизация. То есть функция load() должна была брать на вход класс определенного типа и понимать какого именно типа она вам должна вернуть параметры. Но библиотека hibernate не типизирована. То есть это некий минус ее. Не знаю почему, может быть какие‑то исторические предпосылки к этому есть.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | HQL поиск по ключу  public UserDataSet readByname(String name) {  Session session = sessionFactory  .openSession();  Criteria criteria = session  .createCriteria(  UserDataSet.class);  return (UserDataSet) criteria.add(  Restrictions.eq('name', name))  .uniqueResult();  } |

Более сложный пример. Представим, что вам нужно найти не просто запись по id. В hibernate, кстати, вот думаю хорошо будет сейчас об этом сказать. Hibernate требует, чтобы у всех таблиц был первичный ключ BigInteger идентификатор. Именно имея у каждой записи этот идентификатор. Почему он еще должен быть автокриментый. hibernate позволяет удобно по id находить записи без использования того поиска, который я сейчас вам покажу в случае, если вам нужно найти не по id, а по какому‑то другому полю, не важно есть ли на нем индекс или нет на нем индекса. Задача. Нам нужно найти userDataSet в таблице, которой этому usetDataSet соответствует по имени. Предположим, что у нас в userDataSet id и имя. И мы хотим не по id найти, а найти по имени. Мы так же, как в предыдущих примерах, создаем сессию (4). То есть в этом смысле ничего у нас не поменялось, но в добавок мы теперь создаем критерий. Это как раз один из элементов HQL. Language, состоящий из объектов и обращений методов к этим объектов. То есть вы запрос формируете объектами. Что такое критерий? Критерий — это вы запрашиваете у сессии (6) дайте мне критерию для класса (8). И после этого вы в критерии можете добавлять некие сужения (9), рестришены на то, что искать. В данном случае я говорю Restrictions проверь, чтобы поле name было равно тому значению, которое я передаю. Обратите внимание вот это name. Здесь на самом деле очень интересно. name — это не имя колонки в базе. Если бы это было имя колонки в базе, то произошло бы некое нарушение абстракции. Вам бы нужно было знать структуры колонок в базе для того, чтобы написать запрос. А по задумке вы не обязаны это знать. Поэтому это name — это не колонка в базе так называется, это название поля, в котором лежит имя внутри userDateSet. Обычно название поля совпадает с названием колонки. Поэтому часто, если вы думаете, что здесь название колонки пишите, у вас будет работать, но не всегда. Может быть такая ситуация, когда поле в dataSet отличается от названия поля в колонке. И в этом случае надо помнить, что здесь при формировании запроса нужно указывать имя поля, а не имя колонки. Вернемся к примерам. Вы задаете Restrictions (10) и в него передаете имя поля и значение поля в userDataSet, которое нужно искать. И после этого вы подразумеваете, что ответ будет всего один, поэтому пишите (11). И после выполнения этого (11) вам на выход hibernate снова вернет собранный userDataSet, либо null, если он ничего не найдет.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | HQL получение всех записей  public List<UserDataSet> readAll() {  Session session = sessionFactory  .openSession();  Criteria criteria = session  .createCriteria(  UserDataSer.class);  return (List<UserDataSet>) criteria  .list();  } |

И в заключении, если вам нужно найти не один элемент, а набор элементов. То вы можете точно так же создать критерий (6) и спросить List (9). Если вернуться к предыдущему примеру, взять из этого предыдущего примера добавление Restrictions, то вы ее же можете добавить в этот пример, но только вместо equals поставить, например, in и передать массив, в котором искать среди каких элементов искать. Тогда вы возьмете не весь список. То есть вот в текущем виде она вам вернет вообще все значения, какие найдет. То есть все колонки, для каждой колонки из таблицы, для всех какие найдет в этой таблице, она вам сформирует UserDataSet, запишет его в лист и вернет. Можно сюда же в этот критерий добавлять сужения на извлечение данных. Можно с помощью HQL устанавливать порядок выдачи. Можно считать количество элементов. Можно задавать лимит по запросу. Это уже просто дополнительные вещи. Если они вам понадобиться, то вы сможете сами в документации о них почитать.

Видеоинструкция