[**1. Библиотека Swing, общие черты и особенности**](#_7vyndbbsg3z1) **3**

[**2. Виды контейнеров в Swing**](#_r5bcfv1u5xz5) **4**

[**3. Элементы пользовательского интерфейса Swing.**](#_322h1nlmsqu6) **4**

[**4. Модель событий Swing. Интерфейс EventListener.**](#_30j0zll) **5**

[**5. Менеджеры компоновки Swing.**](#_1fob9te) **6**

[**6. GUI Designer Swing**](#_14uxcyjaro0o) **8**

[**7. Текстовые поля в Swing**](#_3znysh7) **9**

[**8. Компонент управления JButton в Swing**](#_rmwfsnh96orn) **10**

[**9. Платформа JavaFX, особенности, компоненты**](#_3aa26v9iqnhw) **11**

[**10. Шаблон MVC (Model-View-Controller) в JavaFX**](#_in310u11bp4w) **13**

[**11. Классы Stage и Scene в JavaFX.**](#_2et92p0) **14**

[**12. Узлы и графы сцены в JavaFX**](#_tyjcwt) **15**

[**13. Класс Application и жизненный цикл приложения JavaFX.**](#_3dy6vkm) **17**

[**14. Инструмент визуальной компоновки JavaFX Scene Builder**](#_1t3h5sf) **18**

[**15. Компоненты управления Label, TextField в JavaFX**](#_aamn9js7m3sp) **19**

[**16. Компонент управления Button в JavaFX**](#_6jld43hhvjk6) **20**

[**17. Основные фреймворки и задачи, решаемые Spring**](#_4d34og8) **21**

[**18. Spring Inversion of Control (IoC) контейнер Spring**](#_2s8eyo1) **22**

[**19. Dependency Injection (DI) в Spring**](#_604po745v99e) **23**

[**20. Жизненный цикл объекта Bean Spring**](#_hbcsy9g07ued) **24**

[**21. Конфигурация ApplicationContext с помощью xml в Spring**](#_yqptpndkf0sm) **25**

[**22. Область видимости Bean в Spring**](#_jjq820hjode0) **29**

[**23. Фабричные или factory-методы в Spring**](#_ptal74231g3i) **31**

[**24. Конфигурация ApplicationContext с помощью аннотаций в Spring**](#_j3ssul7dfmsb) **32**

[**25. Cвязывание в Spring, аннотация @Autowired**](#_3rdcrjn) **34**

[**26. Архитектурный стиль REST**](#_btc6vv3jpfcr) **36**

[**27. Spring Web-MVC, основная схема и логика работы**](#_muezhk1mkslz) **38**

[**28. Класс DispatcherServlet, его функции**](#_97f67m3cvxy2) **39**

[**29.Маппинг в Spring.**](#_mbp9d4cq8nc) **41**

[**30.Интерфейсы HttpServletRequest и HttpServletResponse**](#_26in1rg) **42**

[**31.Архитектурный стиль CRUD, его соответствие REST и HTTP.**](#_5vt5l3nfm6yt) **44**

[**32.Шаблон Data Access Object (DAO)**](#_k0fj4u79ucat) **45**

[**33.Основные понятия Объектно-реляционного отображения (ORM - Object-Relational Mapping)**](#_lnxbz9) **46**

[**34.Спецификация Java Persistence API (JPA)**](#_35nkun2) **47**

[**35.Архитектура ORM Java Persistence API (JPA)**](#_su7x694o884k) **48**

[**36. Основные аннотации Java Persistence API (JPA)**](#_1ksv4uv) **49**

[**37. Библиотека Hibernate, основные аннотации**](#_gfz9hm1lsmej) **51**

[**38. Объявление сущности и таблицы в Hibernate**](#_monoiwtprq34) **53**

[**39. Интерфейс Session в Hibernate**](#_44sinio) **54**

[**40. Ассоциация сущностей в Hibernate**](#_2jxsxqh) **57**

[**41. Spring Boot: определение, характеристики, преимущества**](#_5gfl8osy9dk3) **58**

[**42. Spring Initializr, особенности и преимущества применения**](#_z337ya) **59**

[**43. Структура фреймворка JUnit**](#_3j2qqm3) **60**

[**44. JUnit аннотации @Test, @DisplayName**](#_1y810tw) **61**

[**45. JUnit аннотации @BeforeEach, @AfterEach**](#_4i7ojhp) **62**

[**46. Тестовые классы и методы JUnit.**](#_2xcytpi) **64**

[**47. Утверждения JUnit. Класс Assert.**](#_1ci93xb) **65**

[**48. Тестирование исключений JUnit.**](#_yqsylkzfsaur) **66**

[**49. Генератор документирования Javadoc. Виды комментариев.**](#_3whwml4) **70**

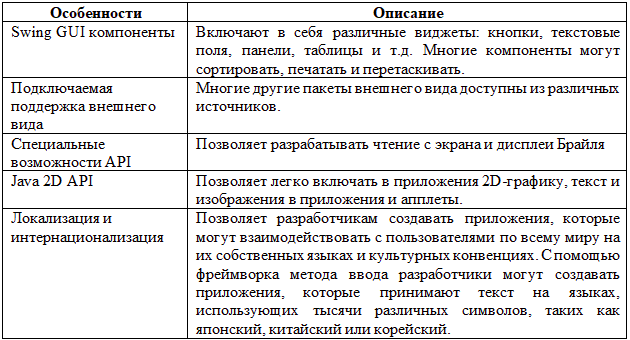
[**50. Дескрипторы Javadoc**](#_2bn6wsx) **71**

ТЕОРИЯ

# 1. Библиотека Swing, общие черты и особенности

Swing - это набор инструментов виджетов GUI для Java, который является частью Oracle Java Foundation Classes (JFC) – API для предоставления графического пользовательского интерфейса (GUI) для программ Java. JFC включают в себя группу функций для создания GUI и разработки графической функциональности и интерактивности в Java-приложениях.

Особенности Swing:



Swing API имеет 18 пакетов:

javax.accessibility, javax.swing.plaf, javax.swing.text, javax.swing javax.swing.plaf.basic, javax.swing.text.html, javax.swing.border javax.swing.plaf.metal, javax.swing.text.html.parser, javax.swing.colorchooser, javax.swing.plaf.multi, javax.swing.text.rtf, javax.swing.event, javax.swing.plaf.synth, javax.swing.tree, javax.swing.filechooser javax.swing.table, javax.swing.undo.

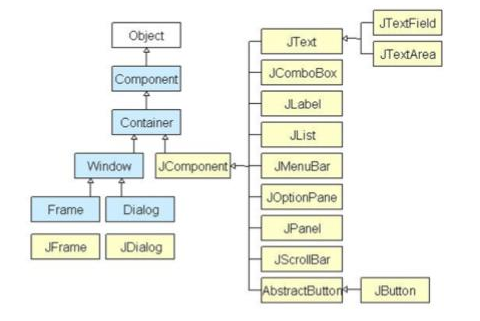
# 2. Виды контейнеров в Swing

Контейнерные классы – это классы, в которых могут быть другие компоненты. Поэтому для создания графического интерфейса нам нужен как минимум один контейнерный объект.

В GUI Swing представлены контейнеры двух типов.

Верхний уровень контейнеров, это классы JFrame, JApplet, JWindow, JDialog. Эти классы являются наследниками классов Component и Container библиотеки АWT. Эти классы, как можно понять из их типа, являются контейнерами, которые находятся на самом верху иерархии – остальные контейнеры и компоненты «вложены» в этот контейнер. В Swing-приложениях чаще всего в качестве контейнера верхнего уровня применяют JFrame.

Вложенные контейнеры второго типа – это JPanel, JScrollPane и JRootPane, они не такие тяжеловесные, как контейнеры верхнего уровня и наследуются от класса JComponent. В качестве примера легковесных контейнеров можно привести классы. Эти контейнеры могут содержаться в других контейнерах, и поэтому они нередко используются для объединения группы взаимосвязанных компонентов. Иерархия классов Swing:



# 3. Элементы пользовательского интерфейса Swing.

JLabel – компонент для размещения текста в контейнере.

JColorChooser – панель управления для выбора цветов пользователем.

JCheckBox – это графический компонент, который может находиться во включенном (true) или выключенном (false) состоянии.

JRadioButton представляет собой графический компонент, который может находиться в состоянии «включено» (true) или «выключено» (false) в группе.

JList предоставляет пользователю список прокручиваемых текстовых элементов.

JComboBox предоставляет пользователю доступ к меню выбора.

JTextField – это текстовый компонент, который позволяет редактировать одну строку текста.

JPasswordField – это текстовый компонент, предназначенный для ввода пароля.

JTextArea – это текстовый компонент, который позволяет редактировать несколько строк текста.

ImageIcon – реализация интерфейса Icon, который рисует значки из изображений.

JOptionPane предоставляет набор стандартных диалоговых окон, которые запрашивают у пользователей значение или информируют их о чем-то.

JFileChooser представляет диалоговое окно, из которого пользователь может выбрать файл.

JSlider позволяет пользователю графически выбирать значение, перемещая ручку в ограниченном интервале.

JSpinner представляет собой однострочное поле ввода, которое позволяет пользователю выбрать число или значение объекта из упорядоченной последовательности.

# 4. Модель событий Swing. Интерфейс EventListener.

Взаимодействие с пользователем осуществляется через событие, то есть изменение состояния объекта. События генерируются в результате взаимодействия пользователя с компонентами графического интерфейса пользователя. Например, нажатие кнопки, перемещение мыши, ввод символа с клавиатуры, выбор элемента из списка и прокрутка страницы – действия, которые вызывают событие.

События могут быть переднего плана и фоновые.

События переднего плана – эти события требуют непосредственного взаимодействия с пользователем. Они создаются как последствия взаимодействия человека с графическими компонентами в графическом интерфейсе пользователя. Например, нажатие кнопки, перемещение мыши, ввод символа с клавиатуры, выбор элемента из списка, прокрутка страницы и т. д.

Фоновые события требуют взаимодействия конечного пользователя. Прерывания операционной системы, аппаратный или программный сбой, истечение таймера и завершение работы и т.д.

Java использует модель событий делегирования для обработки событий. Эта модель определяет стандартный механизм для генерации и обработки событий.

Источник – это объект, на котором происходит событие. Источник отвечает за предоставление информации о произошедшем событии своему обработчику. Java предоставляет нам классы для исходного объекта.

Слушатель – обработчик событий. Слушатель отвечает за генерацию ответа на событие. С точки зрения реализации Java слушатель также является объектом. Слушатель ждет, пока не получит событие. Как только событие получено, слушатель обрабатывает событие и затем возвращается.

Слушатели событий представляют интерфейсы (EventListener), отвечающие за обработку событий. Java предоставляет различные классы слушателей событию Каждый метод метода слушателя событий имеет единственный аргумент в качестве объекта, который является подклассом класса EventObject.

# 5. Менеджеры компоновки Swing.

Менеджеры компоновки управляют размещением компонентов в контейнере: BorderLayout, BoxLayout, CardLayout, FlowLayout, GridBagLayout, GridLayout, GroupLayout, SpringLayout.

BorderLayout помещает компоненты в пять областей: верхнюю, нижнюю, левую, правую и центральную

Класс BoxLayout помещает компоненты в одну строку или столбец. Он учитывает требуемые максимальные размеры компонентов, а также позволяет выравнивать компоненты.

Класс CardLayout позволяет реализовать область, содержащую различные компоненты в разное время. CardLayout часто управляется компонентом ComboBox, при этом состояние ComboBox определяет, какую панель (группу компонентов) отображает CardLayout

FlowLayout – это менеджер компоновки по умолчанию для каждой JPanel. Он просто раскладывает компоненты в одну строку, начиная новую строку, если ее контейнер недостаточно широк. Обе панели в CardLayoutDemo, показанные ранее, используют FlowLayout

GridLayout просто делает набор компонентов равным по размеру и отображает их в требуемом количестве строк и столбцов

GroupLayout – менеджер компоновки, который был разработан для использования инструментами GUI Builder, но его также можно использовать вручную. Groupplayout работает с горизонтальными и вертикальными макетами отдельно

SpringLayout – гибкий менеджер компоновки, предназначенный для использования разработчиками графического интерфейса. Он позволяет указать точные отношения между ребрами компонентов, находящихся под его контролем. Например, вы можете определить, что левый край одного компонента находится на определенном расстоянии (которое может быть динамически вычислено) от правого края второго компонента

# 6. GUI Designer Swing

GUI Designer Swing является утилитой для проектирования и визуализации GUI приложений java, написанных с использованием библиотеки Swing. Представлена в виде плагина для IDEA. В качестве альтернативы можно рассматривать SceneBuilder, который используется для работы с библиотекой JavaFX. По своей сути решение является устаревшим (как и JavaFX SceneBuilder) и при разработке новых проектов не применяется. В данный момент разработчики ушли от написания Desktop GUI приложений на Java и используют такие технологии как Electron, которые запускают экземпляр браузера с движком V8 у вас локально на устройстве, тем самым используя более привычный стек HTML/CSS/JS для фронтенд-разработчика.

# 7. Текстовые поля в Swing

Для ввода текста в Swing предусмотрены два вида компонентов: JTextField и JTextArea. В поле JTextField имеется возможность для ввода только одной строки. В JTextArea, как это можно понять из названия, можно ввести несколько строк. Есть также компонент для ввода пароля JPasswordField, который расширяет JTextField, – это текстовое поле, которое позволяет делать ввод текста и при этом введенный текст не отображается – вводятся звездочки. Все эти классы являются наследниками абстрактного класса JTextComponent.

Основные методы:

- public String getText() – возвращает введенный в поле текст, пример использования:

login = txtFieldLogin.getText();

- void setText(String text) – устанавливает текст в поле компонента.

Методы, которые или устанавливают или получают то, что может ли пользователь редактировать текстовое поле:

- boolean isEditable()

- void setEditable (boolean b)

Для устранения пробелов в начале и в конце текста служит метод trim ():

String txtField = textField.getText().trim();

Метод setFont () задает шрифт текстового поля.

Swing текстовые поля не имеют возможности прокрутить слишком длинный текст, который не умещается в поле. Однако можно в панель прокрутки ввести текстовое поле:

JScrollPane scrollPane = new JScrollPane(textArea);

# 8. Компонент управления JButton в Swing

Класс JButton является реализацией кнопки. Этот компонент имеет метку и генерирует событие при нажатии. Он также может иметь изображение.

JButton button = new JButton("Кнопка"):

button.addActionListener(new ButtonAction());

Конструкторы классов:

* JButton() - cоздает кнопку без заданного текста или значка;
* JButton (Действие) - Создает кнопку, где свойства берутся из предоставленного действия;
* JButton (текст строки) - Создает кнопку с текстом;
* JButton (текст строки, значок значка) - Создает кнопку с исходным текстом и значком;

Методы класса:

* AccessibleContext getAccessibleContext () - получает AccessibleContext, связанный с этим JButton;
* getUIClassID () - возвращает строку, которая указывает имя класса L & F, который отображает этот компонент;
* isDefaultButton () - получает значение свойства defaultButton, которое при значении true означает, что эта кнопка является текущей кнопкой по умолчанию для ее JRootPane;
* isDefaultCapable () - получает значение свойства defaultCapable;
* paramString () - возвращает строковое представление этого JButton;
* void removeNotify () - переопределяет JComponent.removeNotify, чтобы проверить, установлена ли эта кнопка в качестве кнопки по умолчанию на RootPane. И если это так, установите для кнопки RootPane по умолчанию значение null чтобы гарантировать, что RootPane не удерживает недопустимую ссылку на кнопку;
* void setDefaultCapable() - Устанавливает свойство defaultCapable, которое определяет, можно ли сделать эту кнопку кнопкой по умолчанию для ее корневой панели;
* void updateUI () - Сбрасывает свойство пользовательского интерфейса в значение из текущего внешнего вида;

Есть также определенные свойства у кнопок. Margin для управления размеров полей, iconTextGap свойство, которое позволяет изменить расстояние между иконкой и текстом,horizontalTextPosition (verticalTextPosition) для управления текста по горизонтали или вертикали и verticalAlignment (horizontalAlignment) вертикальное и горизонтальное выравнивание всего содержимого кнопки.

# 9. Платформа JavaFX, особенности, компоненты

JavaFX создана как универсальная платформа, предоставляющая современные GUI-компоненты с возможностью их декларативного описания, богатый набор библиотек для работы с медиаконтентом и 2D/3D графикой, а также высокопроизводительную среду выполнения приложений.

**Особенности JavaFX:**

* JavaFX изначально поставляется с большим набором частей графического интерфейса, таких как кнопки, текстовые поля, таблицы, деревья, меню, диаграммы и т.д., что в свою очередь сэкономит нам вагон времени.
* JavaFX часто использует стили CSS, и мы сможем использовать специальный формат FXML для создания GUI, а не делать это в коде Java. Это облегчает быстрое размещение графического интерфейса пользователя или изменение внешнего вида или композиции без необходимости долго изучать код Java.
* JavaFX имеет готовые к использованию части диаграммы, поэтому нам не нужно писать их с нуля.
* JavaFX дополнительно поставляется с поддержкой 3D графики.

Основные составляющие окна:

Stage — это окружающее окно, которое используется как начальное полотно и содержит в себе остальные компоненты. У приложения может быть несколько stage, но один такой компонент должен быть всегда. По сути Stage является основным контейнером и точкой входа.

Scene — отображает содержание stage. Каждый stage может содержать несколько компонентов — scene, которые можно между собой переключать. Внутри это реализуется графом объектов, который называется — Scene Graph (где каждый элемент — узел, ещё называемый как Node).

Node — это элементы управления, например, кнопки метки, или даже макеты (layout), внутри которых может быть несколько вложенных компонентов. У каждой сцены (scene) может быть один вложенный узел (node), но это может быть макет (layout) с несколькими компонентами. Вложенность может быть многоуровневой, когда макеты содержат другие макеты и обычные компоненты. У каждого такого узла есть свой идентификатор, стиль, эффекты, состояние, обработчики событий.

# 10. Шаблон MVC (Model-View-Controller) в JavaFX

MVC (Model View Controller / Модель Представление Контроллер) — это некая концепция, призванная упростить разработку, поддержку и изменение программы. При создании нового JavaFX-приложения генерируется каркас.

* Main.java — это точка запуска программы. Здесь же происходит подключение mainform.fxml, на основе которого строится сама форма;
* Controller.java — это контролёр, который декларируется как часть MVC.

При этом отстутствует Model.java и View.java, поэтому придется создать их самостоятельно. Модель должна сама хранить своё состояние.Это могут быть переменные, которые участвуют в расчетах.Для их установки используются «сеттеры» (set-функции) — прямого доступа к переменным извне нет. Это довольно типовое построение классов.

Контроллеры принимают все action действия (т.е. все нажатия кнопок и здесь прописывают их реализацию). Контроллер завязан на форму и все действия выполняет с ней.

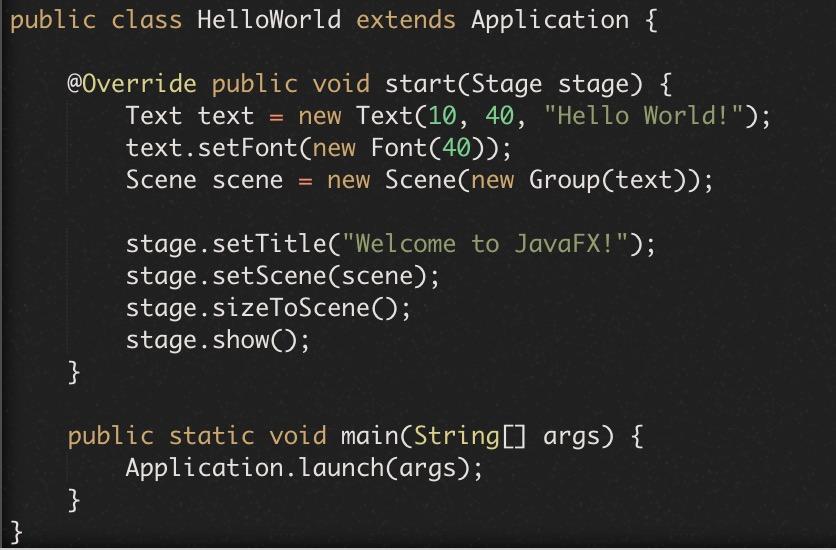
Остался последний компонент Представление (файл *View.java*). В современном представлении MVC компоненту View отводится роль «автоматического отображения» результата модели. То есть, как только мы нажали кнопку, контроллер выполняет функцию(где срабатывает определенная строка) и модель, после всех своих вычислений, должна «уведомить» View, что всё готово, забирай результат.

Минус этого подхода в том, что View оказывается жестко завязан на передаваемый тип Label для вывода результата. Если мы поменяем тип, то придется менять и Контроллер и Представление. Кроме того, Контроллеру необходимо знать, то что ему особо и не нужно. Было бы логичней, чтобы View сам решал куда следует выводить данные.

# 11. Классы Stage и Scene в JavaFX.

Stage — это окружающее окно, которое используется как начальное полотно и содержит в себе остальные компоненты. У приложения может быть несколько stage, но один такой компонент должен быть всегда. По сути Stage является основным контейнером и точкой входа.

Scene — отображает содержание stage. Каждый stage может содержать несколько компонентов — scene, которые можно между собой переключать. Внутри это реализуется графом объектов, который называется — Scene Graph (где каждый элемент — узел, ещё называемый как Node).

****

# 12. Узлы и графы сцены в JavaFX

В JavaFX узлами (nodes) названы отдельные компоненты сцены (кнопки, текстовые поля и т.д.). Узлы также могут иметь дочерние узлы, тогда их называют родительским узлом (parent node) или узлом ветвления (branch node). Конечные узлы без дочерних называются листьями (leaves). Все узлы образуют граф сцены (scene graph).

Узел, не имеющий родительского узла, называется корневым (root) и он является для всех узлов родителем. Все типы узлов описываются в классе Node. От Node наследуются такие классы, как Parent, Group, Region и Control.

Класс Node является суперклассом всех UI-элементов.

Controls: подкласс Parent в javafx.scene.control, (Label, TextField, Button).

Layout Pane (Containers): подкласс Parent в javafx.scene.layout, (StackPane, FlowPane, GridPane, BorderPane).

Geometrical Shapes: подкласс Shape, Shape3D в javafx.scene.shape, (Circle, Rectangle, Polygon, sphere, Box).

Media Elements: ImageView, MediaView (в media player) в javafx.scene.image и javafx.scene.media.

Region является базовым классом для всех элементов управления пользовательского интерфейса на основе узлов JavaFX и всех контейнеров. Это родительский узел с изменяемым размером, который можно стилизовать с помощью CSS. Может иметь несколько фонов и границ.

Публичный абстрактный класс Control, расширяющий Region, имплементирует Skinnable. Это базовый класс для всех элементов управления пользовательским интерфейсом. Control – узел в графе сцены, которым может управлять пользователь. Элементы управления предоставляют дополнительные переменные и поведение, выходящие за рамки узлов, для поддержки общих взаимодействий пользователей таким образом, чтобы они были согласованными и предсказуемыми для пользователя.

# 13. Класс Application и жизненный цикл приложения JavaFX.

В абстрактном классе Application объявляются три метода, которые позволяют управлять жизненным циклом приложения.

Это методы **init (), start() и stop()**, и их можно переопределить в соответствии с логикой разрабатываемого приложения.

Формы объявления в порядке вызова методов.

void init ()

abstract void start(Stage основная\_платформа)

void stop ()

**Метод init()** вызывается при запуске приложения, его применяют для инициализации каких либо переменных, которые, возможно, нужны перед стартом приложения.Этот метод запускается by default и если не требуется какая-либо начальная инициализация, то метод init() можно не переопределять.

Вторым методом вызывается **start().** С него начинается работа приложения, и его можно использовать для конструирования и установки параметров сцены. Этот метод используется для настройки «сцены». Аргументом метода start() является ссылка на объект Stage, который является primary stage, то есть основной платформой. Метод – абстрактный, и его обязательно нужно переопределить.

Работа приложения заканчивается вызовом метода **stop().** Этот метод можно использовать для сборки мусора, освобождения ресурсов, закрытия соединений. Если для приложения этого не требуется, то в явном виде можно метод не вызывать.

Для запуска вызывается метод launch(), который также определен в интерфейсе Application. Метод имеет следующий синтаксис:

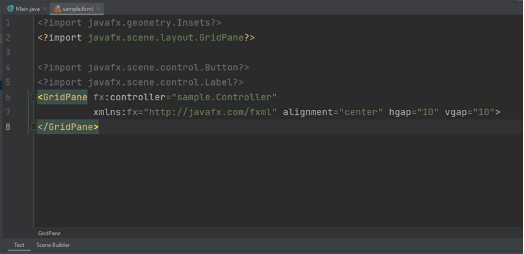
public static void launch(String ... аргументы)

Где аргументы обычно являются аргументами командной строки. Вызов метода launch() приводит к загрузке приложения, сопровождающейся последующими вызовами методов init () и start ().

# 14. Инструмент визуальной компоновки JavaFX Scene Builder

Инструмент визуальной компоновки JavaFX Scene Builder позволяет пользователям быстро создавать пользовательские интерфейсы приложений JavaFX, как утверждает разработчик - без кодирования. Пользователи могут перетаскивать компоненты пользовательского интерфейса в рабочую область, изменять их свойства, применять таблицы стилей, а код FXML для создаваемого макета автоматически генерируется в фоновом режиме. В результате получается файл FXML, который затем можно объединить с проектом Java, привязав пользовательский интерфейс к логике приложения.

Впрочем, можно изменять представление непосредственно в fxml файле:



# 15. Компоненты управления Label, TextField в JavaFX

Компонент **Label** – самый простой в использовании, однако, самый частый в применении. Этот компонент просто отображает текстовое сообщение.

Метка является экземпляром класса **Label**, пакет javafx. scene. control. Класс **Label** расширяется методами от различных классов, таких, как Labeled и Control. Класс Control – предоставляет методы, которые вызываются для любых элементов управления.

Один из конструкторов класса Label:

Label(String text);

В JavaFX есть различные компоненты для текстового ввода данных. Это **TextField** и TextArea.

**TextField** предназначен для ввода текста в одну строку, удобно вводить имена, идентификаторы, адреса, логины и т.д. Все текстовые компоненты наследуются от класса TextinputControl, который предоставляет значительное число методов.

Класс TextField имеет перегрузку двух конструкторов. Один – конструктор по умолчанию, и создает пустое поле:

public TextField()

Второй конструктор позволяет ввести текст:

public TextField(String text)

////

Есть интересная возможность ввода текста по столбцам, и этой возможностью незаслуженно мало пользуются:

public final void setPrefColumnCount(int value)

Текст подсказки предоставляется вызовом метода

final void setPromptText(String text)

////

# 16. Компонент управления Button в JavaFX

Класс **Button** из в пакета javafx.scene.control предоставляет компонент **«кнопка»**. Класс **Button** наследуется от непосредственно от ButtonBase и много функций унаследовано от базового класса.

У кнопок есть возможность содержать текст и рисунки. Поэтому Button имеет *два конструктора*:

1. public Button (String text), где text – это строка на кнопке.

2. public Button (String text, Node graphic) – можно добавить изображение.

При клике на кнопку вызывается событие ActionEvent. Слушатель этого события вызывается методом setOnAction ():

final void setOnAction(EventHandler<ActionEvent> обработчик)

Хотя при использовании fxml обработчики событий указываются там и в явном виде не вызываются.

К примеру:

<Button fx:id="btnOpenPort" layoutX="633.0" layoutY="410.0" mnemonicParsing="false" onAction="#clickbtnOpenPort" prefHeight="35.0" prefWidth="147.0" text="Open Port" />

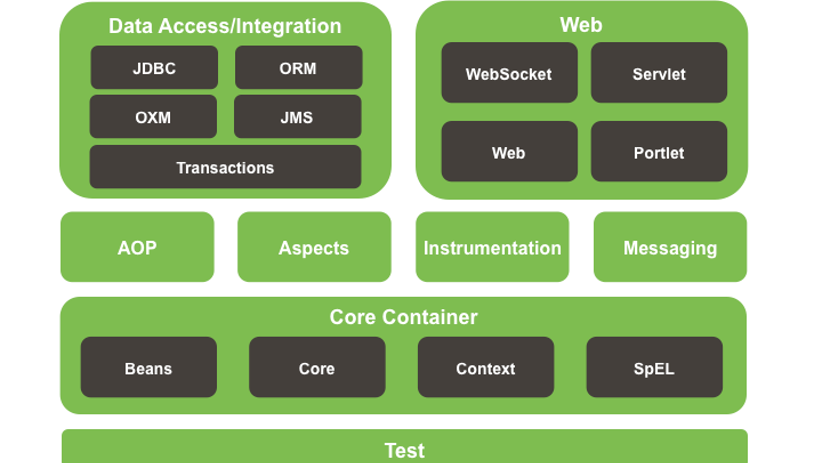
При этом в настройках Scene Builder можно указать на какое действие должна отозваться кнопка:

# 17. Основные фреймворки и задачи, решаемые Spring

Spring Framework – фреймворк с открытым исходным кодом для разработки Java-приложений. Spring позволяет ускорить и облегчить решение многих задач Java-разработки Spring позволяет создавать приложения из «обычных» объектов Java, так называемых POJOs (англ. - Plain Old Java Objects, можно вольно перевести, как «Старые добрые объекты Java»), обертывая их в свои объекты.

Spring состоит из отдельных фреймворков, большинство которых может работать и независимо друг от друга, но если они используются совместно, то обеспечивают большую функциональность.

Spring состоит из следующих модулей:

****

Spring Framework состоит из примерно 20 модулей. Эти модули сгруппированы в Core Container, Data Access/Integration, Web, AOP (Аспектно-ориентированное программирование), Instrumentation, Messaging и Test

# 18. Spring Inversion of Control (IoC) контейнер Spring

Inversion of Control (IoC), также известное как Dependency Injection (DI) – это процесс, при котором объекты определяют свои зависимости через аргументы конструктора или методы внедряемых классов.

Все созданные в программе объекты (POJOs) помещаются в Spring Application Context и управляются Spring. Spring производит Dependency Injection – внедряет зависимости между объектами. ApplicationContext представляет собой Spring IoC контейнер и необходим для инициализации, настройки и сборки объектов Bean для построения приложения. Другими словами, IoC – это процесс, в котором объект определяет свои зависимости, не создавая их. Объект делегирует работу по построению таких зависимостей контейнеру IoC. ApplicationContext – это интерфейс в Spring-приложении, в котором производится конфигурация Java-приложения.

В Spring объекты, составляющие основу приложения и управляемые контейнером Spring IoC, называются beans.

# 19. Dependency Injection (DI) в Spring

Dependency Injection – внедрение зависимости между объектами приложения. Внедрение зависимости при помощи конструктора класса и методов set.

Dependency Injection (внедрение зависимостей) – ключевой шаблон проектирования в Spring. Мы говорим фреймворку создать за нас бины (иначе говоря – объекты) и внедрить их в другие бины. И фреймворк это делает.

Но как объяснить фреймворку Spring, что такой-то бин должен стать зависимостью для другого бина? Вариантов немного, а самых частых всего два: бин внедряется либо через конструктор класса, либо с помощью сеттера. Первое называется constructor-based injection, а второе – setter-based injection.

. Dependency Injection во фреймворке Spring может быть реализован через…

-: создание зависимости в pom.xml

+: аргументы конструктора

-: ресурсный файл

-: JDBC- и ORM-средства

-: вызов метода dependency()

# 

# 20. Жизненный цикл объекта Bean Spring

Жизненный цикл Bean можно представить вместе с развитием контейнера:



Вначале выполняется init-method, который выполняется по умолчанию, но можно сделать это явно, заложив требуемую логику при инициализации. Перед тем, как приложение завершает работу вызывается destroy-method, который также выполняется по умолчанию, но также можно заложить в нем явно выполнение каких-то действий перед тем, как Bean закончит свой «жизненный путь».

Спецификация init- и destroy-методов:

- Может быть любой модификатор доступа.

- Любой тип возвращаемого значения (чаще void).

- Название, как для любого другого метода.

- Не должно быть аргументов.

- Для бинов типа prototype Spring не вызывает destroy-метод по умолчанию.

- Для prototype бинов будет вызвано столько init-методов, сколько объектов бинов.

# 21. Конфигурация ApplicationContext с помощью xml в Spring

Другим способом конфигурации контекста является конфигурация на основе аннотаций, которая использует метаданные байт-кода для подключения компонентов. Вместо того чтобы использовать xml-файл для того, чтобы создать и подключить требуемые компоненты, можно внедрить указания в сам класс, где находится компонент, применив аннотации к соответствующему классу, методу или объявлению поля.

Аннотации в Java – своего рода метки в коде, специальный тип комментариев описывающими метаданные для функции/класса/пакета, с помощью которых можно:

- передавать инструкции для компилятора;

- передавать инструкции для анализаторов исходного кода;

- передавать метаданные.

Преимущества конфигурации при помощи аннотаций очевидны: код получается более компактным и логика приложения становится понятней. Недаром конфигурация Spring-приложений на основе xml-файла была исторически первой и в настоящее время считается устаревшей. Возможно применение обоих подходов. К примеру, аннотация @Required применяется к set-методам свойств bean и указывает, что это свойство bean обязательно должно быть определено в xml-файле конфигурации. В противном случае контейнер создает исключение BeanInitializationException.

Пример:

public class MyUser {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Required

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Соответствующий конфигурационный файл:

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xmlns:context = "http://www.springframework.org/schema/context"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:annotation-config/>

<!-- Определение bean -->

<bean id = "userBean" class = "MyUser">

<property name = "name" value = "IvanIvanov" />

</bean>

Аннотации классов @Configuration и методы с аннотацией @Bean – это основа конфигурации внедрения зависимостей в IoC контейнере.

Аннотация @Bean указывает компилятору, что метод управляется Spring IoC контейнером. Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component. Применение аннотации @Configuration к классу указывает на то, что в нем производится определение бинов. То есть для того, чтобы объявить бин, достаточно указать аннотацию @Bean методу. Аннотация @Bean – это аналог того, как Bean определяется в xml-файле при помощи тегов <bean> … <bean/>.

Пример.

@Configuration

public class AppConfig {

@Bean

public TransferServiceImpl transferService() {

return new TransferServiceImpl();

}

}

По этой аннотации Spring Framework при сканировании классов создает бин.

Начиная с версии Spring 2.5, внедрение зависимостями можно управлять при помощи аннотаций. Одна из основных аннотаций для этих целей - @Autowired. Это позволяет Spring-приложению разрешать и вводить зависимости. Spring framework обеспечивает автоматическое внедрение зависимостей. Другими словами, объявив все бин-зависимости в файле конфигурации Spring, Spring container может автоматически подключать отношения между бинами. При помощи @Autowired помечают конструкторы, поля, set-методы или методы конфигурации, и они автоматически будут подключены средствами внедрения зависимостей Spring.

@Autowired конструкторы.

Только один конструктор любого класса может объявить эту аннотацию. Если несколько конструкторов объявят аннотацию, они будут рассматриваться как кандидаты на автоматическое подключение. Будет выбран конструктор с наибольшим количеством зависимостей, которые могут быть удовлетворены путем сопоставления бинов в контейнере Spring. Если ни один из кандидатов не может быть удовлетворен, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Аналогично, если класс объявляет несколько конструкторов, но ни один из них не аннотируется с помощью @Autowired, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Если класс объявляет только один конструктор для начала, он всегда будет использоваться, даже если он не аннотирован. Аннотированный конструктор не обязательно должен иметь модификатор public.

@Autowired поля.

Поля вводятся сразу после объявления Bean, перед вызовом любых методов конфигурации, хотя считается, что такая практика не очень хорошая. Такой конфигурационный файл не обязательно должен иметь модификатор доступа public.

@Autowired методы.

Методы конфигурации могут иметь произвольное имя и любое количество аргументов; каждый из этих аргументов будет автоматически связан с соответствующим bean’ом в контейнере Spring. Такие методы не обязательно должны быть public.

# 

# 22. Область видимости Bean в Spring

Область действия Bean определяется жизненным циклом и видимостью в контексте, в котором он используется. Spring framework определяет 6 типов областей видимости.

- singleton

По умолчанию Spring IoC контейнер создает единственный экземпляр бина. Как правило, используется для бинов без сохранения состояния(stateless)).

- prototype

Spring IoC контейнер создает любое количество экземпляров бина. Новый экземпляр бина создается каждый раз, когда бин необходим в качестве зависимости, либо через вызов getBean(). Как правило, используется для бинов с сохранением состояния(stateful). Синтаксис в xml-файле: <bean id = "someBean" scope = "prototype">

- request

Жизненный цикл экземпляра ограничен единственным HTTP запросом; для каждого нового HTTP запроса создается новый экземпляр бина. Действует, только если вы используете web-ориентированный ApplicationContext.

- session

Создаётся один экземпляр бина на каждую HTTP сессию. Действует, только если вы используете web-ориентированный ApplicationContext.

- application

Ограничивается одним определением компонента в жизненном цикле ServletContext. Допустимо только в контексте веб-ориентированного Spring ApplicationContext.

# 23. Фабричные или factory-методы в Spring

Конструкторы в java – это механизм по умолчанию для получения экземпляров классов. Они обеспечивают всю инфраструктуру, необходимую для введения зависимостей, либо вручную, либо автоматически.

Вместе с тем, в некоторых случаях лучше воспользоваться фабричными методами для достижения того же результата.

Даже знаменитый Joshua Block в книге Effective Java сказал: «Consider static factory methods instead of constructors» («Применяйте статические фабричные методы вместо конструкторов»). Это связано с тем, что:

- методы могут иметь понятные имена (легче обозначить какой именно объект нам нужен);

- можно не создавать экземпляр каждый раз при вызове метода (при одинаковых параметрах мы возвращаем один и тот же неизменяемый (immutable) экземпляр;

- можем вернуть объект любого наследника класса;

- можем возвращать разных наследников на разные вызовы метода.

# 24. Конфигурация ApplicationContext с помощью аннотаций в Spring

Вместо того чтобы использовать xml-файл для того, чтобы создать и подключить требуемые компоненты, можно внедрить указания в сам класс, где находится компонент, применив аннотации к соответствующему классу, методу или объявлению поля.

Аннотации в Java – своего рода метки в коде, специальный тип комментариев описывающими метаданные для функции/класса/пакета, с помощью которых можно:

- передавать инструкции для компилятора;

- передавать инструкции для анализаторов исходного кода;

- передавать метаданные.

Преимущества конфигурации при помощи аннотаций очевидны: код получается более компактным и логика приложения становится понятней. Недаром конфигурация Spring-приложений на основе xml-файла была исторически первой и в настоящее время считается устаревшей. Возможно применение обоих подходов. К примеру, аннотация @Required применяется к set-методам свойств bean и указывает, что это свойство bean обязательно должно быть определено в xml-файле конфигурации. В противном случае контейнер создает исключение BeanInitializationException.

Аннотации классов @Configuration и методы с аннотацией @Bean – это основа конфигурации внедрения зависимостей в IoC контейнере.

Аннотация @Bean указывает компилятору, что метод управляется Spring IoC контейнером. Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component. Применение аннотации @Configuration к классу указывает на то, что в нем производится определение бинов. То есть для того, чтобы объявить бин, достаточно указать аннотацию @Bean методу. Аннотация @Bean – это аналог того, как Bean определяется в xml-файле при помощи тегов <bean> … <bean/>.

# 25. Cвязывание в Spring, аннотация @Autowired

Начиная с версии Spring 2.5, внедрение зависимостями можно управлять при помощи аннотаций. Одна из основных аннотаций для этих целей - @Autowired. Это позволяет Spring-приложению разрешать и вводить зависимости. Spring framework обеспечивает автоматическое внедрение зависимостей. Другими словами, объявив все бин-зависимости в файле конфигурации Spring, Spring container может автоматически подключать отношения между бинами. При помощи @Autowired помечают конструкторы, поля, set-методы или методы конфигурации, и они автоматически будут подключены средствами внедрения зависимостей Spring.

@Autowired конструкторы.

Только один конструктор любого класса может объявить эту аннотацию. Если несколько конструкторов объявят аннотацию, они будут рассматриваться как кандидаты на автоматическое подключение. Будет выбран конструктор с наибольшим количеством зависимостей, которые могут быть удовлетворены путем сопоставления бинов в контейнере Spring. Если ни один из кандидатов не может быть удовлетворен, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Аналогично, если класс объявляет несколько конструкторов, но ни один из них не аннотируется с помощью @Autowired, то будет использоваться основной конструктор/конструктор по умолчанию (если он присутствует). Если класс объявляет только один конструктор для начала, он всегда будет использоваться, даже если он не аннотирован. Аннотированный конструктор не обязательно должен иметь модификатор public.

@Autowired поля.

Поля вводятся сразу после объявления Bean, перед вызовом любых методов конфигурации, хотя считается, что такая практика не очень хорошая. Такой конфигурационный файл не обязательно должен иметь модификатор доступа public.

@Autowired методы.

Методы конфигурации могут иметь произвольное имя и любое количество аргументов; каждый из этих аргументов будет автоматически связан с соответствующим bean’ом в контейнере Spring. Такие методы не обязательно должны быть public.

# 26. Архитектурный стиль REST

REST является аббревиатурой на английском языке – REpresentational State Transfer, что можно перевести, как «передача состояния представления») – это архитектурный стиль для распределенных гипермедиа-систем, впервые представленный Роем Филдингом в 2000 году в его знаменитой диссертации 3.

Архитектурный стиль REST имеет 6 ограничений, которые должны быть удовлетворены, если интерфейс хочет называться RESTful:

Клиент–серверная структура.

Отделяя проблемы пользовательского интерфейса от проблем хранения данных, мы улучшаем переносимость пользовательского интерфейса на нескольких платформах и улучшаем масштабируемость за счет упрощения серверных компонентов.

Stateless (отсутствие состояния).

Каждый запрос от клиента к серверу должен содержать всю информацию, необходимую для понимания запроса, и не может использовать преимущества любого сохраненного контекста на сервере. Таким образом, состояние сеанса полностью сохраняется на клиенте.

Cacheable (Кэширование)

Ограничения кэша требуют, чтобы данные в ответе на запрос были неявно или явно помечены как кэшируемые или некэшируемые. Если ответ кэшируется, то клиентскому кэшу предоставляется право повторно использовать эти данные ответа для последующих эквивалентных запросов.

Единый интерфейс

Применяя принцип общности программной инженерии к компонентному интерфейсу, упрощается общая архитектура системы и улучшается видимость взаимодействий. Чтобы получить единый интерфейс, необходимо несколько архитектурных ограничений, определяющих поведение компонентов. REST определяется четырьмя ограничениями интерфейса: идентификация ресурсов; манипулирование ресурсами через представления; самоописательные сообщения; и гипермедиа как механизм состояния приложения.

Layered system (Многоуровневая система)

Стиль многоуровневой системы позволяет архитектуре состоять из иерархических слоев, ограничивая поведение компонентов таким образом, что каждый компонент не может «видеть» за пределами непосредственного слоя, с которым они взаимодействуют.

Code on demand (Код по требованию)

Это необязательное ограничение. REST позволяет расширить функциональность клиента путем загрузки и выполнения кода в виде апплетов или скриптов. Это упрощает работу клиентов за счет сокращения количества функций, необходимых для предварительной реализации.

Интересно, что Рой Филдинг является также и одним из создателей HTTP.

# 27. Spring Web-MVC, основная схема и логика работы

В Spring есть специальный фреймворк для web-реализаций. Spring Web MVC – это оригинальный веб-фреймворк, построенный на основе API сервлетов и с самого начала включенный в Spring Framework. Формальное название «Spring Web MVC» происходит от названия его исходного модуля (spring-webmvc), но более широко он известен в сокращенном названии как «Spring MVC». Схематично MVC можно представить на рис. 3.1.

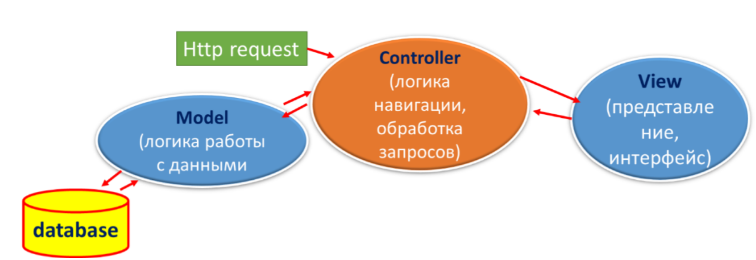


Рис. 3.1. Схема web-MVC

Model (Модель) – содержит данные приложения. Данные могут быть одним объектом или коллекцией объектов.

Controller (Контроллер) – cодержит бизнес-логику приложения. Здесь аннотация @Controller используется для обозначения класса как контроллера.

View (Представление) – отображает предоставленную информацию в определенном формате.

# 28. Класс DispatcherServlet, его функции

Логика работы Spring MVC построена вокруг класса DispatcherServlet (входной точки приложения), который принимает и обрабатывает все HTTP-запросы. Схема работы показана на рис. 3.2.

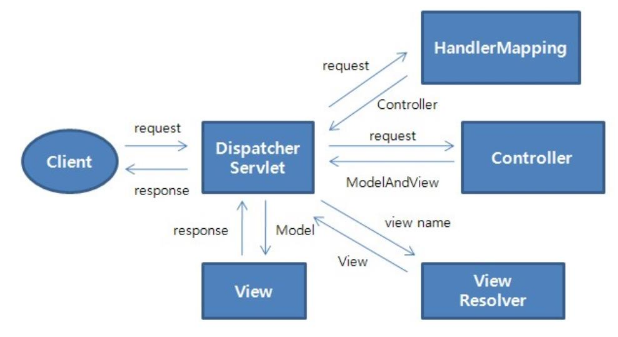


Рис. 3.2.

Последовательность событий, соответствующих входящему HTTP−запросу к DispatcherServlet:

1. После получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой Контроллер должен быть вызван, после чего, отправляет запрос в нужный Контроллер.

2. Контроллер принимает запрос и вызывает соответствующий служебный метод, основанный на GET или POST. Вызванный метод определяет данные Модели, основанные на определённой логике, и возвращает в DispatcherServlet имя View.

3. При помощи интерфейса ViewResolver DispatcherServlet определяет, какой View нужно использовать на основании полученного имени.

4. После того, как View создан, DispatcherServlet отправляет данные Модели в виде атрибутов в View, который в конечном итоге отображается в браузере.

Все вышеперечисленные компоненты являются частями WebApplicationContext, который является расширением ApplicationContext с некоторыми дополнительными функциями, необходимыми для веб-приложений.

# 29.Маппинг в Spring.

URL, по которому мы обращаемся в методе, задаете при помощи маппинга

@GetMapping — Обрабатывает get-запросы (получаем все записи)

@PostMapping - Обрабатывает post-запросы (создаем новую запись)

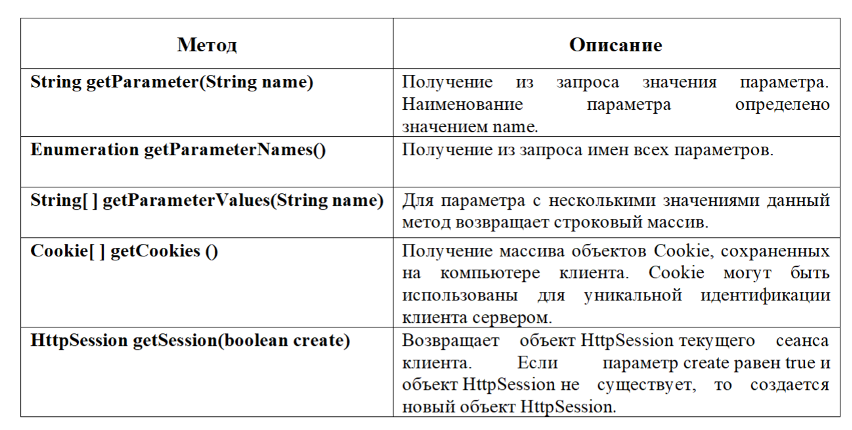
@DeleteMapping - Обрабатывает delete-запросы (удаляем запись)

@PutMapping - Обрабатывает put-запросы (обновляем запись)

По виду HTTP-запроса или @RequestMapping

# 30.Интерфейсы HttpServletRequest и HttpServletResponse

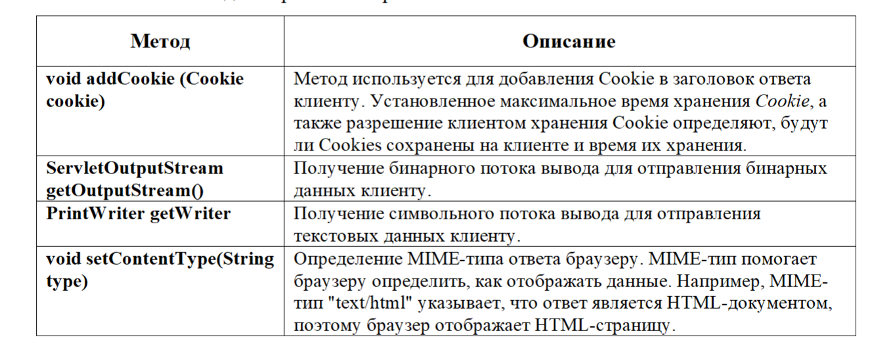
HttpServletRequest является интерфейсом и расширяет интерфейс ServletRequest. Расширяя ServletRequest, этот интерфейс может разрешить запрашивать информацию для HTTP-сервлетов. Объект HttpServletRequest создается контейнером сервлета, а затем передается сервисному методу, реализующий GET, POST и т.д. запросы. Когда браузер запрашивает веб-страницу, он отправляет много информации на веб-сервер, которая не может быть прочитана непосредственно, потому что эта информация перемещается как часть заголовка HTTP-запроса. HttpServletRequest представляет этот HTTP-запрос.



**Интерфейс HttpServletResponse.**

Расширяет интерфейс ServletResponse для обеспечения специфичных для HTTP функций отправки ответа. Например, у него есть методы доступа к HTTP-заголовкам и файлам cookie.

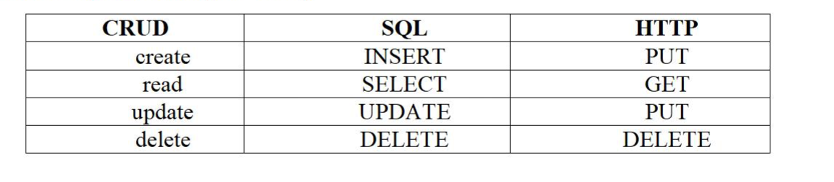
Контейнер сервлета создает объект HttpServletResponse и передает его в качестве аргумента сервисным методам сервлета (doGet, doPost и т. д.).



# 31.Архитектурный стиль CRUD, его соответствие REST и HTTP.

В 1983 году Джеймс Мартин, английский ученый, работавший в области информационных технологий, в свой книге «Managing the Data-base Environment»4 ввел аббревиатуру CRUD (англ. – Create, Read, Update, Delete). Так он обозначил архитектурный стиль программного обеспечения, которое реализует четыре основные операции управления базой данных: Create – создание, Read – чтение, Update – изменение, Delete – удаление. В SQL этим операциям соответствуют операторы Insert, Select, Update, Delete.

Каждая функция в CRUD соответствует стандартной операции Structured Query Language (SQL) и методу HTTP-запроса, которые обычно используются для создания RESTful API приложения



# **32.Шаблон Data Access Object (DAO**)

Data Access Object или шаблон DAO используется для разделения уровня данных и бизнес-служб высокого уровня. Это повышает безопасность данных, так как затрудняет изменение данных напрямую:

- отделяет клиентский интерфейс управления данными от механизмов доступа к данным;

- адаптирует API доступа конкретного ресурса данных к универсальному интерфейсу клиента, то есть шаблон DAO позволяет механизмам доступа к данным изменяться независимо от кода, который использует данные.

То есть вся логика работы с базой данных инкапсулируется в классе, реализующем DAO

# 33.Основные понятия Объектно-реляционного отображения (ORM - Object-Relational Mapping)

ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение, или преобразование) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Задачей ORM является обеспечить работу с данными в терминах классов, а не таблиц данных, и, напротив, преобразовать термины и данные классов в данные, пригодные для хранения в СУБД. Необходимо также обеспечить интерфейс для CRUD-операций над данными. В общем, необходимо избавиться от необходимости писать SQL-код для взаимодействия в СУБД. Объектно-реляционное отображение – это автоматическое сохранение объекта из Java-приложения в таблицах базы данных SQL с использованием метаданных, описывающих отображение между классами приложения и схемой базы данных SQL. По сути, ORM работает как преобразование данных из представления Java в SQL и обратно.

Существует множество библиотек ORM, реализованных на различных языках: Doctrine, Tryton, ActiveRecord, EclipseLink, Hibernate.

Hibernate – это одна из самых популярных библиотек для языка Java, которая реализует спецификацию JPA (Java Persistence API).

# 34.Спецификация Java Persistence API (JPA)

Java Persistence API (JPA) – спецификация API Java EE, система сохранения и управления Java-объектами в базе данных. Это спецификация, предполагается, что должна быть реализация данной спецификации (Hibernate, к примеру).

В Java существует интересный процесс, который позволяет языку развиваться: Java Community Process (JCP). Это родившееся в 1998 году движение или процесс, при участии в котором можно участвовать в формировании спецификаций платформ языка Java.

JPA состоит из следующих компонентов:

- собственно API, (javax.persistence) – это набор интерфейсов, классов;

- объектно-ориентированный язык запросов Java Persistence Query Language;

- метаинформация, описывающая связи между объектами;

-генерация запросов на языке описания данных (Data Definition Language, DDL) для сущностей.

# 35.Архитектура ORM Java Persistence API (JPA)

Архитектура ORM состоит из трех этапов процесса сохранения объекта в реляционной базе данных.

**1 этап. Создание данных объекта.**

Создание класс, имплементирующем Entity, где поля – это столбцы в базе данных. Создание класса, которые обертывают в методы запросы к базе данных (DAO – Data Access Object/Service).

**2 этап Отображение.**

JPA Provider: библиотека релизации JPA, к примеру, Hibernate.

Конфигурационный файл ORM.xml: содержит конфигурацию отображения между данными в классе POJO и данными в реляционной базе данных.

JPA Loader: работает как кеш-память. Может загружать данные реляционной таблицы, работает как копия базы данных для взаимодействия с классами обслуживания для данных POJO (атрибуты класса POJO).

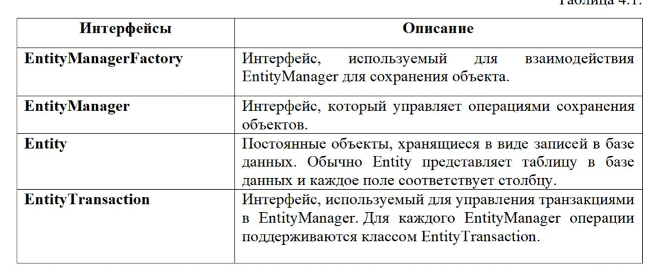
Object grid: это временное расположение, в котором может храниться копия реляционных данных, например, кэш-память. Все запросы к базе данных сначала выполняются на данных в сетке объектов.

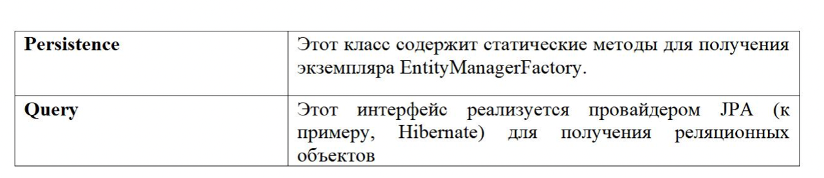
Mapping: Установление связей между сущностями

**3 этап. Сохранение в базе данных.**

Конфигурацию отображения можно производить в файле xml, то есть производится сравнение атрибутов класса POJO с тегами сущностей в файле xml. Также ту же процедуру можно производить при помощи аннотаций. Аннотации используются для классов, свойств и методов. Все аннотации JPA определены в пакете javax.persistence

*интерфейсы и классы, которые показаны в архитектуре*





# 36. Основные аннотации Java Persistence API (JPA)

Java Persistence API (JPA) – спецификация API Java EE, система сохранения и управления Java-объектами в базе данных.

Конфигурацию отображения можно производить как в файле xml, так и при помощи аннотаций. Аннотации используются для классов, свойств и методов. Все аннотации JPA определены в пакете javax.persistence. Среди основных аннотаций можно выделить следующие:

| Аннотации | Описание |
| --- | --- |
| @Entity | Объявляет класс как таблицу |
| @Table | Объявляет имя таблицы |
| @Basic | Задает необязательные поля в явном виде |
| @Embedded | Определяет свойства класса или объекта, значение которого является экземпляром встраиваемого класса |
| @Id | Определяет свойство, используемое для идентификации (первичный ключ таблицы) класса |
| @GeneratedValue | Определяет, как атрибут идентичности может быть инициализирован, например, автоматический, ручной или значение, взятое из таблицы последовательности |
| @Transient | Определяет значение, которое не сохраняется в базе данных |
| @Column | Определяет атрибут столбца для свойства постоянства |
| @SequenceGenerator | Задает значение для свойства, указанного в аннотации @GeneratedValue |
| @TableGenerator | Указывает генератор значений для свойства, указанного в аннотации @GeneratedValue. Создает таблицу для генерации |
| @AccessType | Этот тип аннотации используется для установки типа доступа. Если вы устанавливаете @AccessType (FIELD), то доступ происходит по полю. Если вы установите @AccessType (PROPERTY), то доступ будет происходить по свойству. |
| @JoinColumn | Определяет ассоциацию объекта или коллекции объектов. Это используется в ассоциациях «многие к одному» и «один ко многим». |
| @UniqueConstraint | Определяет поля и уникальные ограничения для первичной или вторичной таблицы |
| @ColumnResult | Ссылается на имя столбца в запросе SQL, используя предложение select |
| @ManyToMany | Определяет отношение «многие ко многим» между таблицами |
| @ManyToOne | Определяет отношение многие-к-одному между таблицами |
| @OneToMany | Определяет отношение «один ко многим» между таблицами |
| @OneToOne | Определяет отношение один к одному между таблицами. |
| @NamedQueries | Указывает список именованных запросов |
| @NamedQuery | Определяет запрос с использованием статического имени |

# 37. Библиотека Hibernate, основные аннотации

Hibernate - одна из наиболее популярных реализаций ORM-модели. Объектно-реляционная модель описывает отношения между программными объектами и записями в БД.

Аннотации являются мощным инструментом для предоставления метаданных, а также намного нагляднее при чтении нашего кода другим разработчиком.

Обязательными аннотациями являются следующие:

**@Entity**

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

**@Table**

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (map) данный класс. Аннотация **@Table** имеет различные атрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблице БД*.

**@Id**

С помощью аннотации **@Id** мы указываем *первичный ключ (Primary Key)* данного класса.

**@GeneratedValue**

Эта аннотация используется вместе с аннотацией **@Id** и определяет такие параметры, как **strategy** и **generator**.

**@Column**

Аннотация **@Column** определяет к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса).

Наиболее часто используемые атрибуты аннотации **@Column** такие:

* **name**Указывает имя столбца в таблице
* **unique**Определяет, должно ли быть данное значение уникальным
* **nullable**Определяет, может ли данное поле быть NULL, или нет.
* **length**Указывает, какой размер столбца (например количество символов, при использовании String).

# 38. Объявление сущности и таблицы в Hibernate

**@Entity**

Эта аннотация указывает Hibernate, что данный класс является сущностью (entity bean). Такой класс должен иметь конструктор по-умолчанию (пустой конструктор).

**@Table**

С помощью этой аннотации мы говорим Hibernate, с какой именно таблицей необходимо связать (map) данный класс. Аннотация **@Table** имеет различные атрибуты, с помощью которых мы можем указать *имя таблицы, каталог, БД и уникальность столбцов в таблице БД*.

**(@Column**

Аннотация **@Column** определяет к какому столбцу в таблице БД относится конкретное поле класса (атрибут класса).

Наиболее часто используемые атрибуты аннотации **@Column** такие:

* **name**Указывает имя столбца в таблице
* **unique**Определяет, должно ли быть данное значение уникальным
* **nullable**Определяет, может ли данное поле быть NULL, или нет.

**length**Указывает, какой размер столбца (например количество символов, при использовании String).)

# 39. Интерфейс Session в Hibernate

В Hibernate работа с БД осуществляется через объект типа *org.hibernate.Session*.

Интерфейс *org.hibernate.Session* является мостом между приложением и Hibernate. С помощью сессий выполняются все CRUD-операции с *объектами-сущностями*. Объект типа *Session* получают из экземпляра типа *org.hibernate.SessionFactory*, который должен присутствовать в приложении в виде *singleton*.

Объект-сущность может находиться в одном из 3-х состояний (статусов):

* *transient object*. Объекты в данном статусе — это заполненные экземпляры классов-сущностей. Могут быть сохранены в БД. Не присоединены к сессии. Поле *Id* не должно быть заполнено, иначе объект имеет статус *detached* ;
* *persistent object*. Объект в данном статусе — так называемая хранимая сущность, которая присоединена к конкретной сессии. Только в этом статусе объект взаимодействует с базой данных. При работе с объектом данного типа в рамках транзакции все изменения объекта записываются в базу;
* *detached object*. Объект в данном статусе — это объект, отсоединённый от сессии, может существовать или не существовать в БД.

Любой объект-сущность можно переводить из одного статуса в другой. Для этого в интерфейсе *Session* существуют следующие методы:

* *persist(Object)* — преобразует объект из *transient* в *persistent*, то есть присоединяет к сессии и сохраняет в БД.
* *merge(Object)* — преобразует объект из *transient* или *detached* в *persistent*. Если из *transient*, то работает аналогично *persist()* (генерирует для объекта новый *Id*, даже если он задан), если из *detached* — загружает объект из БД, присоединяет к сессии, а при сохранении выполняет запрос *update*
* *replicate(Object, ReplicationMode)* — преобразует объект из *detached* в *persistent*, при этом у объекта обязательно должен быть заранее установлен *Id*. Данный метод предназначен для сохранения в БД объекта с заданным *Id*.
* *delete(Object)* — удаляет объект из БД, иными словами, преобразует *persistent* в *transient*. *Object* может быть в любом статусе, главное, чтобы был установлен *Id*.
* *save(Object)* — сохраняет объект в БД, генерируя новый *Id*, даже если он установлен. *Object* может быть в статусе *transient* или *detached*
* *update(Object)* — обновляет объект в БД, преобразуя его в *persistent* (*Object* в статусе *detached*)
* *saveOrUpdate(Object)* — вызывает *save()* или *update()*
* *refresh(Object)* — обновляет *detached*-объект, выполнив *select* к БД, и преобразует его в *persistent*
* *get(Object.class, id)* — получает из БД объект класса-сущности с определённым *Id* в статусе *persistent*

Объект *Session* кэширует у себя загруженные объекты; при загрузке объекта из БД в первую очередь проверяется кэш. Для того, чтобы удалить объект из кэша и отсоединить от сессии, используется *session.evict(Object)*. Метод *session.clear()* применит *evict()* ко всем объектам в сессии.

А теперь обратим внимание на аннотации *@OneToMany* и *@ManyToOne* в классах-сущностях. Параметр *fetch* в *@OneToMany* означает, когда загружать дочерние объекты. Может иметь одно из двух значений, указанных в перечислении *javax.persistence.FetchType*:

*FetchType.EAGER* — загружать коллекцию дочерних объектов сразу же, при загрузке родительских объектов.

*FetchType.LAZY* — загружать коллекцию дочерних объектов при первом обращении к ней (вызове *get*) — так называемая отложенная загрузка.

# 40. Ассоциация сущностей в Hibernate

Ассоциации описывают, как два или более Entity класса формируют отношения между собой на основе отношений реляционной базы данных. Hibernate поддерживает следующие виды ассоциаций:

one-to-one — один к одному

one-to-meny — один ко многим

many-to-one — многие к одному

many-to-many — многие ко многим

Аннотация **@OnetoOne** определяет отношение один к одному — когда один объект сущности (Entity класса) включает атрибут (свойство) ссылающийся на другой объект сущности. Кроме того при отношении один к одному, значение атрибута, ссылающегося на другой объект должно быть уникально для всей коллекции.

Аннотация **@ManyToOne** определяет отношение многие к одному — когда множество объект сущности (Entity класса) ссылается на один объект другой сущности.

Ассоциация **@OneToMany** связывает родительский объект (Entity класс) с одним или несколькими дочерними объектами (Entity классами). Если @OneToMany не имеет зеркальной ассоциации @ManyToOne на дочерней стороне, ассоциация @OneToMany является однонаправленной. Если на дочерней стороне есть ассоциация @ManyToOne, ассоциация @OneToMany является двунаправленной.

Аннотация **@ManyToMany** определяет отношение многие ко многим — когда объект Entity класса с одной стороны содержит коллекцию объекты Entity классов другой стороны. В свою очередь объекты Entity классов со второй стороны могут содержать коллекцию объектов Entity классов первой стороны.

# 41. Spring Boot: определение, характеристики, преимущества

Spring Boot — это полезный проект, целью которого является упрощение создания приложений на основе Spring. Он позволяет наиболее простым способом создать web-приложение, требуя от разработчиков минимум усилий по его настройке и написанию кода.

Spring Boot позволяет разработчикам настраивать и запускать автономные приложения Spring производственного уровня без их развертывания на веб-сервере.

Особенности:

* Простота управления зависимостями;
* Зависимости Opinionated (Opinionated означает, что Spring Boot самостоятельно определяет набор настроенных bean-компонентов по умолчанию, которые вы можете переопределить при необходимости);
* Автоматическая конфигурация;
* Встроенная поддержка сервера приложений — контейнера сервлетов.

Преимущества Spring Boot

* Быстрая и легкая разработка приложений на основе Spring.
* Автоконфигурация всех компонентов для приложения Spring производственного уровня.
* Готовые встроенные серверы (Tomcat, Jetty и Undertow), обеспечивающие ускоренное и более продуктивное развертывание приложений.
* HTTP end-points, позволяющие вводить внутренние функции приложения, такие как показатели, состояние здоровья и другие.
* Отсутствие конфигурации XML.
* Огромный выбор плагинов, облегчающих работу разработчиков со встроенными базами данных и базами данных в памяти.
* Легкий доступ к базам данных и службам очередей, таким как MySQL, Oracle, MongoDB, Redis, ActiveMQ и другим.
* Плавная интеграция с экосистемой Spring.
* Большое сообщество и множество обучающих программ, облегчающих ознакомительный период.

# 42. Spring Initializr, особенности и преимущества применения

Для начальной конфигурация в экосистеме Spring Boot существует проект Spring Initializr7, который предназначен для генерации проектов Maven и Gradle для любых выбранных зависимостей.

Spring Initializr предоставляет расширяемый API для создания проектов на основе JVM с реализациями нескольких общих концепций: Генерация базового языка для Java, Kotlin и Groovy.

Постройте системную абстракцию с реализациями для Apache Maven и Gradle. .gitignore служба поддержки. Несколько точек подключения для генерации пользовательских ресурсов. Различные варианты проектов выражены в модели метаданных, которая позволяет вам настроить список зависимостей, поддерживаемые версии JVM и платформы и т. Д.

Spring Initializr также предоставляет веб-конечные точки для создания реального проекта, а также обслуживает его метаданные в хорошо известном формате, чтобы сторонние клиенты могли оказывать необходимую помощь.

# 43. Структура фреймворка JUnit

# 

В основе фреймворка JUnit лежит идея о том, что вначале нужно написать тест, а потом – собственно сам код программы. JUnit – это фреймворк с открытым исходным кодом. Пятый релиз JUnit в отличие от предыдущих версий разросся и состоит из отдельных подпроектов:

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage9

JUnit Platform – платформа для запуска на JVM фреймворков тестирования, которая реализована на всех самых популярных средах программирования Java (IntelliJ IDEA, Eclipse, NetBeans, Visual Studio Code);

JUnit Jupiter – этот модуль включает в себя новые модели программирования и расширения для написания тестов в JUnit 5;

JUnit Vintage – поддержка тестов, написанных на JUnit 3 и JUnit 4.

В названии фреймворка есть слово Unit, что можно перевести, как «модуль», то есть это модульное тестирование, что означает такой подход к тестированию, когда производится проверка отдельных законченных частей кода, которые связаны единым функциональным управлением, процедурами использования и операционными процедурами.

JUnit 5 требует версии Java не ниже 8.

# 

# 44. JUnit аннотации @Test, @DisplayName

Аннотация *@Test* обозначает тестовые методы. Как и ранее, эти методы должны быть *public void*. Здесь размещаются сами проверки. Кроме того, у данной аннотации есть два параметра, *expected* — задает ожидаемое исключение и *timeout* — задает время, по истечению которого тест считается провалившимся.

Тестам можно присвоить человекопонятное имя для JUnit 5 или описание для TestNG, что позволяет легче читать код тестов и результаты тестирования. В JUnit 4 такой фичи нет.

**JUnit 5** предоставляет аннотацию @DisplayName. При запуске теста имя тестового метода будет выводиться в консоль.

@Test

@DisplayName("Some Test Method")

void someTest(String word) {

//code

}

В **TestNG** описание можно указать в аннотации @Test через атрибут description.

@Test(description = "Some Test Method")

public void someTest() {

//code

}

# 

# 45. JUnit аннотации @BeforeEach, @AfterEach

Аннотации типа Before и After используются для выполнения некоторого кода перед или после выполнения тестов, например для установки переменных или настройки конфигурации.

Запуск перед каждым тестовым методом(@BeforeEach)

Запуск после каждого тестового метода (@AfterEach)

Обратите внимание, что в **JUnit 4** @BeforeClass и @AfterClass объявляются как статичные методы.

@BeforeClass

public static void MethodName() {

// one-time initialization code

System.out.println("@BeforeClass - oneTimeSetUp");

}

В **JUnit 5** @BeforeAll и @AfterAll (пришедшие на смену @BeforeClass и @AfterClass из JUnit 4) статические. Однако @BeforeEach и @AfterEach вызываются для каждого экземпляра теста и статическими быть не должны.

public class AppTest {

@BeforeAll

static void setup(){

System.out.println("@BeforeAll executed");

}

@BeforeEach

void setupThis(){

System.out.println("@BeforeEach executed");

}

@AfterEach

void tearThis(){

System.out.println("@AfterEach executed");

}

@AfterAll

static void tear(){

System.out.println("@AfterAll executed");

}

# 

# 46. Тестовые классы и методы JUnit.

Тестовый класс: любой класс, статический класс или @Nested класс, в котором содержится хотя бы один метод тестирования. Тестовые классы не должны быть абстрактными и должны иметь хотя бы один конструктор.

Метод тестирования: любой метод, который имеет аннотацию @Test, @RepeatedTest, @ParameterizedTest, @TestFactory или @TestTemplate.

Метод жизненного цикла: любой метод, который имеет аннотацию @BeforeAll, @AfterAll, @BeforeEach или @AfterEach.

Методы тестирования и методы жизненного цикла могут быть объявлены локально в текущем тестовом классе, унаследованы от суперклассов или унаследованы от интерфейсов. Кроме того, методы тестирования и методы жизненного цикла не должны быть абстрактными и не должны возвращать значение.

Для создания теста с помощью JUnit следует наследовать тест-класс TestCase, переопределить методы setUp и tearDown при необходимости, ну и самое главное — разработать тестовые методы, наименование которых должно начинаться с аббривиатуры "test". При запуске теста сначала создается экземляр тест-класса (для каждого теста в классе отдельный экземпляр класса), затем выполняется метод setUp, запускается сам тест, ну и в завершение выполняется метод tearDown. Если какой-либо из методов вызывает исключение, тест считается провалившимся. Тестовые методы должны быть public void, могут быть static.

# 47. Утверждения JUnit. Класс Assert.

Утверждения используются для выполнения тестов только при соблюдении определенных условий. Обычно они используются для внешних условий, которые необходимы для правильной работы теста, но которые не связаны напрямую с тем, что тестируется. Все утверждения являются статическими методами в классе org.junit.Assert. Основные утверждения: assertEquals, assertArrayEquals, assertSame, assertNotSame, assertTrue, assertFalse, assertNull, assertNotNull, assertLinesMatch, assertIterablesMatch. К примеру, проверяем будет ли сумма, вычисляемая в тестируемом методе вычислена верно:

@Test

void addNumbers() {

assertEquals(30, CalcNumber.addNumbers(25,5));

}

# 48. Тестирование исключений JUnit.

Иногда возникновение исключения является ожидаемым поведением системы, и в тестах нужно проверять, что оно действительно возникает.

1. @Test

Самый простой способ сообщить тестовому фреймворку о том, что ожидается исключение – указать дополнительный параметр expected в аннотации @Test:

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**public** **class** **MyTest** **{**

**@Test(**expected **=** **IOException.**class**)**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**}**

**}**

Этот параметр должен содержать тип ожидаемого исключения. Если возникнет исключение именно такого типа – тест пройдёт успешно. Если возникнет исключение другого типа или не возникнет вовсе – тест упадёт.

**2. try-catch**

Оба недостатка можно устранить, если перехватывать исключение явно при помощи конструкции try-catch:

**import** org.junit.Assert**;**

**import** org.junit.Test**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**public** **class** **MyTest** **{**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

**try** **{**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

**Assert.**fail**(**"Expected IOException"**);**

**}** **catch** **(IOException** thrown**)** **{**

**Assert.**assertNotEquals**(**""**,** thrown**.**getMessage**());**

**}**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

Если исключение возникает до блока try – тест падает, мы узнаём о том, что у него возникли проблемы.

Если тестируемая функция не выбрасывает вообще никакого исключения – мы попадаем на fail() в следующей строке, тест падает.

Если она выбрасывает исключение неподходящего типа – блок catch не ловит его, тест опять таки падает.

Успешно он завершается только тогда, когда тестируемая функция выбрасывает исключение нужного типа.

Тест стал более надёжным, он больше не пропускает баги. А в блоке catch можно проверить свойства пойманного исключения.

**3. @Rule**

Однако работать с конструкцией try-catch неудобно.

Чтобы избавиться от неё, можно воспользоваться правилом ExpectedException, входящим в стандартный дистрибутив JUnit 4:

**import** org.junit.Rule**;**

**import** org.junit.Test**;**

**import** org.junit.rules.ExpectedException**;**

**import** java.io.IOException**;**

**import** java.nio.file.Files**;**

**import** java.nio.file.Path**;**

**import** static org**.**hamcrest**.**CoreMatchers**.**equalTo**;**

**import** static org**.**hamcrest**.**CoreMatchers**.**not**;**

**public** **class** **MyTest** **{**

**@Rule**

**public** **ExpectedException** thrown **=** **ExpectedException.**none**();**

**@Test**

**public** **void** **testCreateTempFile()** **throws** **IOException** **{**

**Path** tmpDir **=** **Files.**createTempDirectory**(**"tmp"**);**

tmpDir**.**toFile**().**delete**();**

thrown**.**expect**(IOException.**class**);**

thrown**.**expectMessage**(**not**(**equalTo**(**""**)));**

**Path** tmpFile **=** **Files.**createTempFile**(**tmpDir**,** "test"**,** ".txt"**);**

thrown **=** **ExpectedException.**none**();**

*// дальше идёт какой-то другой код*

*// в нём тоже может появиться неожиданный IOException*

*// если это случится -- тест упадёт*

**}**

**}**

Теперь код имеет простую плоскую структуру, хотя общее количество строк кода, к сожалению, увеличилось.

Но главная проблема этого способа заключается в том, что проверки в таком стиле выглядят противоестественно – сначала описывается поведение, а потом вызывается функция.

**4. assertThrows**

Метод JUnit 5 assertThrows получает ожидаемый тип исключения в качестве первого параметра, а исполняемый файл (функциональный интерфейс) потенциально вызывает исключение как второй. Метод завершится ошибкой, если не будет сгенерировано исключение или исключение другого типа. Метод возвращает само исключение, которое можно использовать для дальнейших утверждений:

@Test

void assertException() {

// arrange

Executable throwingExecutable = () -> {

throw new RuntimeException("Unexpected error!");

};

// act and assert

RuntimeException thrown = Assertions.assertThrows(

RuntimeException.class, throwingExecutable::execute, "???"

);

Assertions.assertAll(

() -> Assertions.assertEquals("Unexpected error!", thrown.getMessage()),

() -> Assertions.assertNotNull(thrown.getCause())

);

}

Если возникает ожидаемое исключение, assert Throws возвращает исключение, что позволяет нам также утверждать сообщение. Кроме того, важно отметить, что это утверждение выполняется, когда вложенный код вызывает исключение типа NullPointerException или любого из его производных типов.

Метод assertThrows() позволяет более точно контролировать логику утверждения исключений, поскольку мы можем использовать его в определенных частях кода.

# 49. Генератор документирования Javadoc. Виды комментариев.

**Javadoc – это инструмент, который анализирует аннотации и комментарии к документации в наборе исходных файлов и создает набор HTML-страниц, описывающих классы, интерфейсы, конструкторы, методы и поля.**

**В Java существует три вида комментариев:**

**1. однострочный комментарий, который обозначается двойным знаком // вначале строки:**

**//этот метод служит для сложения двух чисел типа int – однострочный комментарий**

**public static int addNumbers(int number1, int number2){**

**int summNumbers = number1 + number2; //нахождение суммы**

**return summNumbers;**

**}**

**2. Многострочный комментарий находится между знаками /\* … \*/**

**/\***

**А этот комментарий многострочный**

**метод для того, чтобы найти разность двух чисел**

**\*/**

**public static int subNumbers(int number1, int number2){**

**int subNumbers = number1 - number2;**

**return subNumbers;**

**}**

**3. Комментарий, как часть документации, обозначается знаками /\*\* … \*/**

***/\*\****

***\* Программа Калькулятор***

***\* @author Ivan Ivanoff***

***\* @version 1.1***

***\*/***

**public class CalcNumber {**

**private int number1;**

**private int number2;**

# 50. Дескрипторы Javadoc

**Особые знаки, называемые дескрипторами и начинающиеся с символа @, служат для того, чтобы пометить строку комментариев. Дескрипторы в фигурных скобках, называются встраиваемыми и могут быть использованы в других дескрипторах.**

| **Дескриптор** | **Применение** | **Описание** |
| --- | --- | --- |
| **@author** | **Класс, интерфейс** | **Обозначение автора программы** |
| **@version** | **Класс, интерфейс** | **Версия. Может быть только один дескриптор на класс** |
| **@since** | **Класс, интерфейс, поле, метод** | **С какой версии был внедрен метод, поле или класс** |
| **@see** | **Класс, интерфейс, поле, метод** | **Ссылка на что-то в документации, на дополнительные сведения** |
| **@param** | **Метод** | **обозначает параметр метода или входные данные, которые он должен ожидать** |
| **@return** | **Метод** | **возвращаемое значение метода** |
| **@exception** | **Метод** | **Обозначение исключения** |
| **@throws имя\_класса описание** | **Метод** | **Описание исключения, которое может быть возникнуть при выполнении метода** |
| **@deprecated** | **Класс, интерфейс, поле, метод** | **Описание устаревших блоков кода** |
| **{@link reference}** | **Класс, интерфейс, поле, метод** | **Ссылка** |
| **{@value}** | **Статичное поле** | **Описание значения переменной** |

ЗАДАЧИ:

Решения размещены тут:

<https://github.com/StudOborona/JavaSummer2021>

1. Создать Swing-приложение с формой авторизации – логином и паролем. Выполните документирование на основе Javadoc.
2. Создать JavaFX-приложение с формой авторизации – логином и паролем. Выполните документирование на основе Javadoc.
3. Создать на основе Spring MVC приложения c конфигурацией на основе аннотаций, которое выводит в браузер: Hello World. Написать тест к этому приложению на основе JUnit 5.
4. Разработать арифметический калькулятор с интерфейсом на основе Swing. Выполните документирование на основе Javadoc.
5. Разработать калькулятор, реализующий операции сложения, вычитания, умножения, деления чисел любого типа. Интерфейс программы выполнить на JavaFX.
6. Создать приложение с графических интерфейсом Swing, для реализации методов побитовых операций: AND (&), OR ( | ), XOR (^), NOT ( ~). Также нужно выполнить побитовые сдвиг вправо, влево на заданное количество бит. Выводить числа в двоичной и десятичной форме.
7. Разработать приложение с графическим интерфейсом JavaFX, в котором пользователь копирует в поле любой текст. Затем выводится список слов этого текста, рядом с каждым числом – сколько раз это слово встречается в тексте.
8. Написать программу вычисления объема параллелепипеда с графическим интерфейсом на основе JavaFX. Написать тест к этому приложению на основе JUnit 5.
9. Написать программу вычисления сопротивления электрической цепи, состоящей из двух последовательно соединенных сопротивлений. Написать тест к этому приложению на основе JUnit 5. Приложение должно быть с графическим интерфейсом JavaFX.
10. Написать на основе Spring Boot клиент-серверное приложение MyUser, в котором можно управлять данными пользователей из базы данных: имя, фамилия, возраст, номер группы. База данных может быть любой – MySQL, PostgreSQL и т.д.
11. Написать на основе Spring Boot Security форму для авторизации пользователя.
12. Напишите программу, которая вычисляет среднее арифметическое чисел, находящихся в файле, который указывает пользователь. Приложение должно быть с графическим интерфейсом JavaFX.
13. Напишите программу, которая выводит на экран список файлов, находящихся в указанном пользователем каталоге. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.
14. Напишите программу, которая создает и выводит на экран 5 случайных целых чисел. Диапазон чисел задает пользователь. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.
15. Напишите программу, которая выводит на экран все положительные двузначные числа (в возрастающем порядке). Диапазон чисел задает пользователь. Приложение должно быть с графическим интерфейсом JavaFX. Написать тест к этому приложению на основе JUnit 5.
16. Разработка MVC-приложения арифметический калькулятор на основе Spring Boot. Применить шаблонизатор Thymeleaf.
17. Написать программу, которая вычисляет сумму первых n членов ряда: 1+1/2+1/4+… Количество суммируемых членов ряда задается пользователем. Написать тест к этому приложению на основе JUnit 5. Приложение должно быть с графическим интерфейсом JavaFX.
18. Разработать программу, которая выводит на экран в строку все положительные двузначные четные числа (в возрастающем порядке). Диапазон вводит пользователь. Написать тест к этому приложению на основе JUnit 5. Приложение должно быть с графическим интерфейсом JavaFX.
19. Написать программу, которая вычисляет факториал введенного числа. Написать тест к этому приложению на основе JUnit 5. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.
20. Разработать программу, которая выводит положительные двузначные числа, делящиеся на 5 (в убывающем порядке). Диапазон вводит пользователь. Написать тест к этому приложению на основе JUnit 5. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.
21. Создать JavaFX-приложение с формой авторизации – логином и паролем, а также аватаром пользователя.
22. Разработка web-MVC приложения на основе Spring Boot. Приложение должно генерировать последовательность из 10 случайных чисел в диапазоне, заданном пользователем, и выводит эти числа на экран и вычисляет их среднее арифметическое.
23. Напишите программную реализацию бинарного дерева поиска. Приложение должно быть с графическим интерфейсом JavaFX.
24. Написать программу, который при введении даты в формате ДД.ММ.ГГ (к примеру, 22.10.20) выводит номер недели. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.
25. Разработать программу для выполнения операций сложения, вычитания и умножения комплексных чисел. Приложение должно быть с графическим интерфейсом JavaFX. Выполните документирование на основе Javadoc.