Оглавление

1. Библиотека Swing, общие черты и особенности.

Библиотека Swing является частью платформы Java и предоставляет набор компонентов и инструментов для создания графического пользовательского интерфейса (GUI) в Java приложениях. Это часть JFC (Java Foundation Classes). Он построен на основе AWT API и полностью написан на Java. Он не зависит от платформы в отличие от AWT и имеет легкие компоненты. Он включает в себя(предоставляет готовый набор) такие компоненты, как кнопка, полоса прокрутки, текстовое поле и т. д.

Общие черты и особенности:

1. Переносимость: Swing построен на основе Java, что позволяет легко переносить приложения на различные операционные системы без изменения исходного кода.(он не зависит от платформы)
2. Легкость программирования: Swing предоставляет простой и интуитивно понятный интерфейс программирования, что делает разработку и поддержку GUI приложений более удобными.
3. Архитектура MVC: Swing следует архитектуре MVC, разделяя между собой данные, пользовательский интерфейс и логику управления, что облегчает разработку и поддержку приложения.
4. Многофункциональность: Swing предоставляет широкий спектр компонентов, включая кнопки, текстовые поля, списки, таблицы, деревья и многое другое. Эти компоненты могут быть настроены и использоваться для создания сложных пользовательских интерфейсов.
5. Поддержка языков: благодаря использованию юникода, Swing позволяет создавать приложения с поддержкой различных языков.

2. Виды контейнеров в Swing.

Контейнеры являются неотъемлемой частью компонентов SWING GUI. Контейнер предоставляет пространство, в котором может быть расположен компонент. Контейнер в AWT является самим компонентом и предоставляет возможность добавлять компонент к себе. Ниже приведены некоторые заметные моменты, которые следует учитывать.

- Подклассы Контейнера называются Контейнером. Например, JPanel, JFrame и JWindow.
- Контейнер может добавить только Компонент к себе.
- Макет по умолчанию присутствует в каждом контейнере, который можно переопределить с помощью метода **setLayout** .

Ниже список самых наиболее часто используемых контейнеров:

1. JPanel – самый простой контейнер. Он обеспечивает пространство, в которое может быть помещен любой другой компонент, включая другие панели.
2. JFrame – это окно верхнего уровня с заголовком и рамкой.
3. Объект JWindow – это окно верхнего уровня без границ и без меню.
4. JScrollPane панель прокрутки - позволяет добавить горизонтальную и/или вертикальную полосы прокрутки к компоненту, когда содержимое компонента превышает размеры и нужна возможность прокрутить его, такому как JTextArea, JTable или JPanel.
5. Java BorderLayout - BorderLayout размещает компоненты в пяти областях: сверху, снизу, слева, справа и по центру. Это менеджер компоновки по умолчанию для каждого JFrame.

3. Элементы пользовательского интерфейса Swing.

Этот вопрос забил в чатку

Пользовательский интерфейс (UI) в библиотеке Swing является набором различных компонентов, которые могут использоваться для создания графического интерфейса пользователя. Некоторые из основных элементов пользовательского интерфейса Swing включают:

1. JFrame: Это основное окно приложения, в котором располагаются другие компоненты.
2. JPanel: Этот компонент используется для группировки других компонентов и управления их размещением.
3. JButton: Он представляет собой кнопку, на которую пользователь может нажать.
4. JLabel: Этот компонент служит для отображения текста или изображения на форме.
5. JTextField: Он представляет собой текстовое поле, в которое пользователь может вводить текст.
6. JPasswordField: Этот компонент служит для ввода пароля.
7. JTextArea: Он представляет собой многострочное текстовое поле для ввода или отображения больших объемов текста.
8. JComboBox: Он представляет собой раскрывающийся список, в котором пользователь может выбирать одну из предоставленных опций.

Это лишь некоторые основные элементы пользовательского интерфейса Swing.

4. Модель событий Swing. Интерфейс EventListener.

Модель событий Swing - Модель обработки событий из набора компонентов Swing ассоциирует каждый компонент с его наблюдателем таким образом, чтобы при возникновении события об этом уведомлялись все его наблюдатели. Наблюдатель можно назвать объектом, «заинтересованным» в возникновении события. При использовании компонентов Swing наблюдателей называют слушателями событий. Они должны реализовывать пустой интерфейс java.util.EventListener, а также (почти во всех случаях) интерфейс слушателя подкласса, который должен иметь хотя бы один метод. События остаются событиями, однако они должны создавать подкласс класса java.util.EventObject. Чрезвычайно важно знать, с какими событиями компонента ассоциированы определенные слушатели событий.

Вся Обработка событий в Swing происходит через интерфейс java.util.EventListener.

Swing содержит множество интерфейсов-слушателей, они используется для обработки событий:

1. KeyListener - используется для обработки событий нажатия клавиш на клавиатуре.
2. MouseListener - используется для обработки событий мыши, таких как нажатие, отпускание кнопки мыши, двойной клик и перемещение курсора мыши.
3. ComponentListener - используется для обработки событий изменения размеров или положения компонента.
4. ChangeListener - используется для обработки событий изменения состояния компонента, например, изменения положения ползунка в полосе прокрутки.
5. HyperlinkListener - используется для обработки событий клика по гиперссылке в текстовом компоненте.
6. ListSelectionListener - используется для обработки событий выбора элементов в списке.
7. DocumentListener - используется для обработки событий изменения текста в текстовом поле или текстовой области.
8. ItemListener - используется для обработки событий выбора элемента в выпадающем списке или флажке.

5. Менеджеры компоновки Swing. Или Менеджеры расположения Layout

Менеджер расположения (layout manager) определяет, каким образом на форме будут располагаться компоненты. Независимо от платформы, виртуальной машины, разрешения и размеров экрана менеджер расположения гарантирует, что компоненты будут иметь предпочтительный или близкий к нему размер и располагаться в том порядке, который был указан программистом при создании программы.

1. BorderLayout: Полярное расположение BorderLayout.
   Менеджер BorderLayout специально предназначен для обычных и диалоговых окон. Он позволяет быстро и просто расположить наиболее часто используемые элементы любого окна: панель инструментов, строку состояния и основное содержимое. Для этого BorderLayout разбивает окно на четыре области, а все оставшееся место заполняется компонентом, выполняющим основную функцию приложения - в редакторе это будет текстовое поле, в многооконном приложении — рабочий стол. Менеджер расположения **BorderLayout** иммет отличия от остальных. Чтобы добавить с его помощью компонент в методе add() необходимо использовать дополнительный параметр, который определяет область контейнера для размещения компонента. В таблице перечислены допустимые значения этого параметра.

2. FlowLayout: **Последовательное расположение FlowLayout.**
   Менеджер последовательного расположение **FlowLayout** размещает компоненты в контейнере слева направо, сверху вниз. При полном заполнении компонентами строки контейнера FlowLayout переходит на следующую строку вниз. Данное расположение устанавливается по умолчанию в панелях JPanel. Основным свойством FlowLayout является определение предпочтительного размера компонентов. Например, размер метки с текстом JLabel соответствует размеру текста.

3. GridLayout: Табличное расположение GridLayout.
   Менеджер расположения GridLayout представляет контейнер в виде таблицы с ячейками одинакового размера. Количество строк и столбцов можно указать в конструкторе. Имеется возможность задать произвольное количество либо строк, либо столбцов, но не одновременно. Все ячейки таблицы имеют одинаковый размер, равный размеру самого большого компонента, находящегося в таблице.

4. GroupLayout: Менеджер расположения GroupLayout
   Менеджер расположения компонентов GroupLayout появился только в Java 6. Он раскладывает компоненты по группам. Группы имеют горизонтальное и вертикальное направление и могут быть параллельными и последовательными. В последовательной группе у каждого следующего компонента координата вдоль оси на единицу больше (имеется в виду координата в сетке), в параллельной – компоненты имеют одну и ту же координату.

6. GUI Designer Swing

GUI Designer Swing позволяет создавать графический интерфейс пользовательских приложений на языке Java. Этот инструмент делает процесс создания GUI более интуитивным и удобным благодаря функционалу перетаскивания и размещения компонентов.

Многие IDE, такие как IntelliJ IDEA и NetBeans, предоставляют документацию и руководства на русском языке по использованию GUI Designer Swing. Вы можете обратиться к этой документации для более подробной информации о работе с GUI Designer Swing на русском языке.

Вот основные функции и возможности GUI Designer Swing:

1. Drag-and-drop (перетаскивание и размещение): GUI Designer Swing позволяет разработчикам добавлять и располагать компоненты Swing на форме GUI с помощью простых операций перетаскивания и размещения мышью. Это значительно упрощает создание пользовательского интерфейса.

2. Визуальная редактирование: Разработчики могут визуально настраивать свойства компонентов, такие как размеры, расположение, цвет, шрифт и другие атрибуты. Это позволяет получить представление о том, как будет выглядеть GUI во время разработки.

3. Панель свойств (Properties panel): GUI Designer Swing предоставляет панель свойств, где можно изменять и настраивать свойства компонентов, включая их текст, внешний вид, связанные события и другие параметры. Панель свойств облегчает изменение свойств компонента без необходимости в ручном вводе кода.

4. Обработчики событий (Event handlers): GUI Designer Swing позволяет добавлять обработчики событий к компонентам GUI, чтобы определить реакцию при взаимодействии с пользователем. Например, можно добавить обработчик события нажатия кнопки, чтобы выполнить определенное действие при нажатии на нее.

7. Текстовые поля в Swing

JTextField, JTextArea, JTextPane

Пакет javax.swing.text библиотеки Swing содержит компоненты для работы с текстом.
Основные возможности всех текстовых компонентов Swing и их базовая архитектура описаны в абстрактном классе JTextComponent. Именно от этого класса унаследованы все текстовые компоненты Swing, будь то простое текстовое поле или многофункциональный редактор.
Текстовые компоненты имеют архитектуру MVC. Модель текстовых компонентов представлена довольно простым интерфейсом Document, который позволяет получать информацию об изменениях в документе и хранящийся в нем текст, а также при необходимости изменять полученный текст. Вид реализован в UI-представителях текстовых компонентов; но составляется он на основе специальных объектов Element и View, больше отвечающих именно текстовым компонентам.

Текстовое поле **JTextField** является самым простым компонентом и наиболее часто встречающимся в пользовательских интерфейсах. Как правило, поле является однострочным и служит для ввода текста. В библиотеке Swing имеется два текстовых поля. Первое, представленное классом *JTextField*, позволяет вводить однострочный текст. Второе поле, реализованное классом **JPasswordField** и унаследованное от поля **JTextField**, дает возможность организовать ввод «закрытой» информации (чаще всего паролей), которая не должна напрямую отображаться на экране.

Многострочное поле JTextArea
Многострочное текстовое поле JTextArea предназначено для ввода простого неразмеченного различными атрибутами текста. В отличие от обычных полей, позволяющих вводить только одну строку текста, многострочные поля дают пользователю возможность вводить произвольное количество строк текста.
Для многострочных полей необходимо задавать не только ширину (максимальное количество символов), но и высоту(максимальное количество строк). JTextArea следует размещать в панелях прокрутки JScrollPane.

Текстовый редактор JEditorPane

Редактор JEditorPane является мощным инструментом для отображения на экране текста любого формата. Он поддерживает два широко распространенных формата: HTML и RTF (Rich Text Format — расширенный текстовый формат). Потенциально редактор JEditorPane может отображать текст любого формата, с любыми элементами и любым оформлением. Такую гибкость редактору обеспечивает фабрика классов EditorKit, в обязанности которой входит создание и настройка всех объектов, необходимых для отображения текста некоторого типа, в том числе модели документа (объекта Document), фабрики для отображения элементов документа ViewFactory, курсора и списка команд, поддерживаемых данным типом текста.

8. Компонент управления JButton в Swing

JButton, JGroupButton, Action

Библиотека Swing включает абсолютно все придуманные на сегодняшний день элементы управления. К ним относятся кнопки, флажки, переключатели, меню и его элементы, и многое другое. Все эти элементы в библиотеке связаны, поскольку они унаследованы от абстрактного класса AbstractButton, определяющего поведение любого компонента, претендующего на звание элемента управления.

Кнопки JButton

Кнопки JButton кроме собственного внешнего вида не включают практически ничего уникального. Поэтому всё, что верно для кнопок, будет верно и для остальных элементов управления. Пример кода создания обычной кнопки :

```
JButton button = new JButton("Кнопка"):
button.addActionListener(new ButtonAction());
```

Основное время работы с кнопками связано не столько с их созданием и настройкой, сколько с размещением в контейнере и написанием обработчиков событий.

Интерфейс кнопок

Внешний вид кнопок JButton можно легко изменить, не меняя менеджера внешнего вида и поведения. С интерфейсом кнопок можно делать практически все — сопоставлять каждому действию пользователя своё изображение, убирать рамку, закрашивать в любой цвет, перемещать содержимое по разным углам, не рисовать фокус.

События элементов управления - ActionEvent, ChangeEvent, ItemEvent

Унаследованные от класса AbstractButton элементы управления, в том числе и кнопки JButton, могут посылать сообщения о трех типах событий (за исключением стандартных событий, общих для всех компонентов Swing). В таблице представлены эти события.

Событие	**Описание**
ActionEvent *слушатель ActionListener*	Событие при нажатии и отпускании кнопки.
ChangeEvent	С помощью данного события модель кнопок

слушатель ChangeListener	ButtonModel взаимодействует со своим UI-представителем. Модель при изменении хранящегося в ней состояния кнопки (это может быть смена включенного состояния на выключенное, обычного на нажатое и т. п.) запускает событие ChangeEvent. UI-представитель обрабатывает это событие и соответствующим образом перерисовывает кнопку.
ItemEvent *слушатель ItemListener*	Событие о смене состояния возникает в компонентах, которые имеют несколько равноценных состояний (например, флажки и переключатели).

Переключатели JRadioButton

Переключатель JRadioButton унаследован от выключателя JToggleButton и отличается от него только внешним видом. JRadioButton используют, как правило, при объединении нескольких переключателей в группу и реализации выбора «один из многих». По одиночке в интерфейсе их практически не используют; обычно эту роль исполняют флажки.

9. Платформа JavaFX, особенности, компоненты

JavaFX — это инструментарий создания GUI и не только для Java.
JavaFX нацелен на создание игр и настольных приложений на Java. По сути им заменят Swing из-за предложенного нового инструмента GUI для Java. Также, он позволяет нам стилизовать файлы компоновки GUI (XML) и сделать их элегантнее с помощью CSS, подобно тому, как мы привыкли к сетевым приложениям.

JavaFX дополнительно работает с интегрированной 3D-графикой, а также аудио, видео и встроенными сетевыми приложениями в единый инструментарий GUI… Он прост в освоении и хорошо оптимизирован. Он поддерживает множество операционных систем, а также Windows, UNIX системы и Mac OS.

Особенности JavaFX:

JavaFX изначально поставляется с большим набором частей графического интерфейса, таких как всякие там кнопки, текстовые поля, таблицы, деревья, меню, диаграммы и т.д., что в свою очередь сэкономит нам вагон времени.

JavaFX часто юзает 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 играться в коде Java.

JavaFX имеет готовые к использованию части диаграммы, поэтому нам не нужно писать их с нуля в любое время, когда вам нужна базовая диаграмма.

JavaFX дополнительно поставляется с поддержкой 3D графики, которая часто полезна, если мы разрабатываем какую-то игру или подобные приложения.

Компоненты JavaFX:

Класс Stage:

Основой для создания графического интерфейса в JavaFX является класс javafx.stage.Stage. По сути он является

контейнером, в который помещаются все остальные компоненты интерфейса. Его конкретная реализация зависит от платформы, на которой запускается приложение. Так, на десктопах это будет отдельное графическое окно, а на мобильных устройствах интерфейс может представлять весь экран устройства.
В программе на JavaFX можно использовать множество объектов Stage, но один из них является основным. При запуске приложения основной объект Stage создается средой JavaFX и передается в метод start():

Класс Stage унаследован от класса javafx.stage.Window, который опредляет базовые возможности окна приложения.

некоторые основные методы класса Stage:

close(): закрывает объект Stage (на десктопах по сути закрывает окно)

hide(): скрывает окно

centerOnScreen(): располагает окно в центре экрана

toBack(): перемещает окно на задний план

toFront(): перемещает окно на передний план

sizeToScene(): устанавливает размеры окна в соответствии с размерами содержимого объекта Scene

show(): отображает окно

getX(): возвращает x-координату окна

getY(): возвращает y-координату окна

getWidth(): возвращает ширину

getHeight(): возвращает высоту

Класс Scene:

Класс javafx.scene.Scene представляет контейнер для всех графических элементов внутри объекта Stage в виде графа, который называется Scene Graph. Все узлы этого графа, то есть по сути все вложенные элементы должны представлять класс javafx.scene.Node. Но корневой узел этого графа должен представлять объект класса, который унаследован от

javafx.scene.Parent. По сути Parent - это контейнер, который может содержать другие элементы.

Для установки корневого узла в Scene применяется один из конструкторов объекта Scene. Основные из них:

Scene(Parent root): создает Scene с корневым узлом root

Scene(Parent root, double width, double height): создает Scene с корневым узлом root, с шириной width и высотой height

Scene(Parent root, Paint fill): создает Scene с корневым узлом root и устанавливает фоновый цвет

Scene(Parent root, double width, double height, Paint fill): создает Scene с корневым узлом root, с шириной width и высотой height и устанавливает фоновый цвет

Графические элементы. Класс Node:

Все графические элементы, которые используются в объекте Scene и добавляются в Scene Graph, должны представлять класс javafx.scene.Node или иначе узел. Все встроенные классы визуальных графических элементов или узлы, например, кнопки, текстовые поля и другие, наследуется от класса Node.

При этом одни узлы Node могут содержать несколько других узлов Node. Например, класс Parent наследуется от Node, но при этом сам может содержать другие узлы Node.

Основные классы, которые наследуются от класса Node:

javafx.scene.shape.Shape: является базовым классом для создания геометрических двухмерных примтивов (например, линия, прямоугольник, эллипс)

javafx.scene.shape.Shape3D: является базовым классом для создания трехмерных объектов

javafx.scene.canvas.Canvas: представляет полотно для отрисовки различного содержимого

javafx.scene.Camera: базовый класс камеры, котоый применяется для рендеринга сцены

javafx.scene.LightBase: предоставляет базовый функционал для классов, которые будут представлять источники света

javafx.scene.image.ImageView: элемент для отображения изображений

javafx.scene.media.MediaView: элемент для работы с мультимедиа

javafx.embed.swing.SwingNode: элемент для встраивания содержимого Swing в JavaFX

javafx.scene.SubScene: элемент для части сцены в JavaFX, позвляет разбить сцену на подсцены

javafx.scene.Parent: базовый класс для всех элементов, которые могут содержать другие элементы

Класс Parent:

В данном случае следует выделить класс javafx.scene.Parent, который представляет функциональность для управления вложенными узлами, их добавления и удаления и прочие операции с ними. От него наследуются следующие классы:

javafx.scene.web.WebView: элемент, который позволяет отображать веб-содержимое.

javafx.scene.Group: представляет контейнер для группы объектов

javafx.scene.layout.Region: базовый класс для всех элементов управления, панелей компоновки и диаграмм. Его отличительная особенность состоит в том, что он добавляет функциональность управления границами и размерами элементов.

Класс Region

Здесь выделим класс javafx.scene.layout.Region, который является базовым классом для большинства визуальных компонентов, которые далее будут рассматриваться.

Класс Region определяет свойства и методы для управления и получения ширины, высоты элемента, для управления его отступами, границами, позиционированием. Некоторые из его основных методов:

getHeight(): возвращает высоту элемента
setHeight(double height): устанавливает высоту элемента
getWidth(): возвращает ширину элемента
setWidth(double height): устанавливает ширину элемента
getBackground(): возвращает фон элемента в виде объекта Background
setBackground(Background value): устанавливает фон элемента
getBorder(): возвращает границу элемента в виде объекта Border
setBorder(Border value): устанавливает границу элемента
getPadding(): возвращает отступы элемента в виде объекта Insets
setPadding(Insets value): задает отступы элемента

Компоненты javafx v2:

Давайте немного пройдёмся по основным составляющим нашего окна:

- **Stage** — по сути 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 в любом случае. По сути Stage является основным контейнером и точкой входа.
- **Scene** — отображает содержание **stage** (прям матрёшка). 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 Sping.

В Spring Framework можно использовать шаблон архитектуры MVC (Model-View-Controller) для построения веб-приложений. MVC разделяет приложение на три основных компонента:

1. Model (Модель) представляет данные и бизнес-логику приложения. В Spring MVC, модель обычно представлена классами Java, которые содержат данные, а также методы для их обработки и взаимодействия с базой данных или другими источниками данных.

2. View (Представление) отвечает за отображение данных пользователю. В Spring MVC, представление обычно представлено шаблонами Thymeleaf, JSP или Freemarker, которые могут быть наполнены данными из модели.

3. Controller (Контроллер) обрабатывает пользовательские запросы и управляет процессом обработки этих запросов. В Spring MVC, контроллеры обычно представлены классами Java, которые аннотированы с помощью @Controller и содержат методы, аннотированные с помощью @RequestMapping для обработки определенных URL-адресов.

Весь процесс взаимодействия с клиентом в приложении, построенном на шаблоне MVC в Spring, происходит следующим образом: клиент отправляет запрос на сервер, контроллер обрабатывает этот запрос, взаимодействует с моделью для получения данных или выполнения некоторых операций, а затем передает эти данные в представление, которое отображает их пользователю.

11.Классы StringBuffer и StringBuilder.

StringBuilder

Класс StringBuilder представляет собой альтернативу классу String, поскольку создает мутабельный (изменяемый) набор символов. Класс StringBuilder, как и класс String, содержит набор методов для управления строковыми объектам

Классы StringBuilder и StringBuffer предоставляют важные методы, которых не предоставляет класс String, такие как insert(), delete() и reverse(). Если вы много раз совершаете какие-либо действия над строками в коде, вам следует использовать StringBuilder или StringBuffer, потому что они намного быстрее и потребляют меньше памяти, чем String. Например, если вы используете конкатенацию строк в цикле, то лучше применять StringBuilder.

Класс StringBuilder не обеспечивает синхронизацию: экземпляры класса StringBuilder не могут совместно использоваться несколькими потоками. Для операций со строками в среде, не являющейся многопоточной, стоит использовать StringBuilder, потому что он быстрее, чем StringBuffer.

StringBuffer

Как и класс StringBuilder, класс StringBuffer также создает изменяемый строковый объект. И StringBuffer содержит те же методы, что и StringBuilder. Таким образом, разница между ними в том, что класс StringBuffer — потокобезопасный и синхронизированный: экземпляры класса StringBuffer могут совместно использоваться несколькими потоками. Для операций со строками в многопоточных средах стоит использовать StringBuffer.

12.Архитектура JDBC (Java DataBase Connectivity). Двух и трехуровневые модели доступа к базе данных. Преимущества и недостатки JDBC.

JDBC (Java DataBase Connectivity) - это интерфейс программирования приложений, который позволяет Java-приложениям взаимодействовать с базами данных. Архитектура JDBC включает в себя двухуровневую и трехуровневую модели доступа к базе данных

В двухуровневой модели, апплете Java или приложении говорит непосредственно с источником данных. Это требует драйвера JDBC, который может связаться с определенным получаемым доступ источником данных. Команды пользователя поставляются базе данных или другому источнику данных, и результаты тех операторов отсылают назад к пользователю. Источник данных может быть расположен на другой машине, с которой пользователь соединяется через сеть. Это упоминается как клиент-серверная конфигурация с машиной пользователя как клиент, и машинный корпус источник данных как сервер. Сеть может быть интранет, которая, например, соединяет сотрудников в пределах корпорации, или это может быть Интернет.

В трехуровневой модели команды отправляются "среднему уровню" служб, который тогда отправляет команды источнику данных. Источник данных обрабатывает команды и отсылает результаты назад к среднему уровню, который тогда отправляет их пользователю. Директора MIS считают трехуровневую модель очень привлекательной, потому что средний уровень позволяет обеспечить контроль над доступом и видами обновлений, которые могут быть сделаны к корпоративным данным. Другое преимущество состоит в том, что это упрощает развертывание приложений. Наконец, во многих случаях, трехуровневая архитектура может обеспечить преимущества производительности.

Плюсы и минусы JDBC

Плюсы

- Простота и легкость в использовании
- Поддержка различных баз данных

- Предоставление набора интерфейсов для выполнения операций с базой данных

Минусы

- Необходимость написания большого количества кода для выполнения простых операций
- Отсутствие проверки типов на этапе компиляции
- Отсутствие ORM (Object-Relational Mapping) возможностей

13.Массивы Java: объявление, инициализация. Основные методы класса Arrays. Доступ к элементам массивов, итерация массивов. Двумерные массивы

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют индексом.

Объявление массива

массив можно одним из двух способов. Они равноправны, но первый из них лучше соответствует стилю Java. Второй же — наследие языка Си (многие Си-программисты переходили на Java, и для их удобства был оставлен и альтернативный способ). В таблице приведены оба способа объявления массива в Java:

№	Объявление массива, Java-синтаксис	Примеры	Комментарий

1.	`dataType[] arrayName;`	`int[] myArray;`
		
		`Object[] arrayOfObjects;`
			Желательно объявлять массив именно таким способом, это Java-стиль

2.	`dataType arrayName[];`	`int myArray[];`
		
		`Object arrayOfObjects[];`
			Унаследованный от C/C++ способ объявления массивов, который работает и в Java

Унаследованный от C/C++ способ объявления массивов, который работает и в Java

В обоих случаях dataType — тип переменных в массиве. В примерах мы объявили два массива. В одном будут храниться целые числа типа int, в другом — объекты типа Object. Таким образом при объявлении массива у него появляется имя и тип (тип переменных массива). arrayName — это имя массива.

Инициализация массива в Java может быть выполнена сразу при объявлении или после объявления. Обратите внимание, что при инициализации необходимо указать размер массива:

```
int[] numbers = new int[5]; // инициализация массива с 5 элементами
```

или можно инициализировать сразу значениями:

```
int[] numbers = {1, 2, 3, 4, 5}; // инициализация массива с 5 элементами - 1, 2, 3, 4, 5
```

Класс Arrays в Java предоставляет набор методов для работы с массивами. Некоторые из основных методов:

- toString(arr): возвращает строковое представление массива
- equals(arr1, arr2): сравнивает элементы двух массивов на равенство
- sort(arr): сортирует элементы массива в порядке возрастания
- copyOf(arr, length): создает копию массива с указанной длиной

Доступ к элементам массива осуществляется по индексу. Индексация начинается с 0. Например, для получения первого элемента массива:

```
int[] numbers = {1, 2, 3};
int firstNumber = numbers[0]; // получение первого элемента
```

Итерация массива может быть выполнена с помощью цикла for или для удобства с использованием расширенного цикла foreach. Примеры:

```
int[] numbers = {1, 2, 3};
// Использование цикла for
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
// Использование расширенного цикла foreach
```

```
for (int number : numbers) {
   System.out.println(number);
}
```

Двумерные массивы в Java представляют собой массивы массивов. Для объявления двумерного массива используется двумерное объявление:

тип_данных[][] имя_массива;

Например, объявление двумерного массива целых чисел:

```
int[][] matrix;
```

или

```
int matrix[][];
```

Инициализация двумерного массива проводится аналогично одномерному массиву:

```
int[][] matrix = new int[3][2]; // инициализация двумерного массива размером 3x2
```

Доступ к элементам двумерного массива осуществляется указанием индексов для каждого измерения. Например, для получения элемента в строке 1 и столбце 2:

```
int[][] matrix = {
   {1, 2},
   {3, 4},
   {5, 6}
};
```

```
int element = matrix[1][2]; // получение элемента, значение: 4
```

Итерация двумерного массива производится вложенными циклами. Например, для итерации всех элементов двумерного массива:

```
int[][] matrix = {
    {1, 2},
    {3, 4},
    {5, 6}
};
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.println(matrix[i][j]);
    }
}
```

14.Иерархия наследования Java. Преобразование типов при наследовании. Ключевое слово instanceof.

В Java иерархия наследования представляет собой отношение между классами, где один класс называется суперклассом или родительским классом, а другой класс называется подклассом или дочерним классом. Класс-подкласс наследует все поля и методы из класса-суперкласса, а также может добавить собственные поля и методы.

В Java преобразование типов при наследовании происходит автоматически в определенных случаях.

1. Upcasting (Преобразование "вверх"):

Upcasting позволяет преобразовать объект от класса-потомка к классу-родителю без явного приведения типов. Например, если у вас есть классы "Родитель" и "Потомок", то объект типа "Потомок" может быть безопасно присвоен переменной типа "Родитель":

```
Родитель obj1 = new Потомок();
```

2. Downcasting (Преобразование "вниз"):

Downcasting требует явного приведения типов и может производиться только от суперкласса к его подклассу или от интерфейса к его реализации. Это необходимо, когда вы хотите получить доступ к методам или полям, специфичным для подкласса. Например, если у вас есть классы "Родитель" и "Потомок", и объект типа "Родитель" был присвоен переменной типа "Потомок", для доступа к методам "Потомка" вам нужно явно привести тип:

```
Потомок obj2 = (Потомок) obj1;
```

Java Instanceof

Java instanceof - это ключевое слово. Это двоичный оператор, используемый для проверки, является ли объект (экземпляр) подтипом данного типа. Он возвращает либо true, либо false. Он возвращает true, если левая часть выражения является экземпляром имени класса с правой стороны. Instanceof оценивает значение true, если объект принадлежит определенному классу или его суперклассу; else вызывает ошибку компиляции. Если мы применим

оператор instanceof с любой переменной с нулевым значением, он вернет false. Это полезно, когда ваша программа может получить информацию о типе времени выполнения для объекта. Ключевое слово instanceof также известно как оператор сравнения типов, поскольку он сравнивает экземпляр с типом.

Синтаксис

(Object reference variable) instanceof (class/interface type)

15.Интерфейсы Java: определение интерфейса, реализация интерфейса. Преимущества применения интерфейсов. Переменные интерфейсов. Наследование интерфейсов. Методы по умолчанию. Статические методы интерфейсов.

В Java интерфейс представляет собой коллекцию абстрактных методов, которые класс может реализовать. Интерфейсы могут содержать только абстрактные методы, константы и методы по умолчанию (начиная с Java 8).

Определение интерфейса:

Чтобы определить интерфейс, используется ключевое слово interface. Например:

```
interface Printable{
   void print();
}
```

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

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

Реализация интерфейсов:

Чтобы класс применил интерфейс, надо использовать ключевое слово implements:

```
public class Program{
   public static void main(String[] args) {
      Book b1 = new Book("Java. Complete Referense.", "H. Shildt");
      b1.print();
   }
}
```

```
interface Printable{
   void print();
}
class Book implements Printable{
   String name;
   String author;
   Book(String name, String author){
      this.name = name;
      this.author = author;
   }
   public void print() {
      System.out.printf("%s (%s) \n", name, author);
   }
}
```

В данном случае класс Book реализует интерфейс Printable

Переменные интерфейсов:

В языке Java переменные, объявленные в интерфейсе, неявно всегда являются полями с модификаторами public, static и final. То есть являются константами.

Переменные интерфейса автоматически считаются как статические (т.е. связаны с самим интерфейсом, а не с экземплярами классов, реализующих интерфейс) и public (т.е. доступны из любого места в программе). Пример объявления переменной в интерфейсе:

```
public interface MyInterface {
   int CONSTANT = 10;
}
```

Эта переменная "CONSTANT" является публичной и статической, и можно использовать ее по имени интерфейса:

```
int value = MyInterface.CONSTANT;
```

В Java, интерфейсы могут наследоваться от других интерфейсов с помощью ключевого слова "extends". Это позволяет создавать иерархию интерфейсов и объединять функциональность от нескольких интерфейсов в один.

Наследование интерфейсов:

Синтаксис наследования интерфейсов выглядит следующим образом:

```
public interface Interface1 {
   void method1();
}
public interface Interface2 {
   void method2();
}
public interface Interface3 extends Interface1, Interface2 {
   void method3();
}
```

Методы по умолчанию:

Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содержать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

```
interface Printable {
   default void print(){
      System.out.println("Undefined printable");
   }
}
```

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом default. Затем в классе Journal нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

```
class Journal implements Printable {
    private String name;
   String getName(){
      return name;
   }
   Journal(String name){
      this.name = name;
   }
}
```

Статические методы интерфейсов:

Статические методы

Статические методы похожи на методы по умолчанию, за исключением того, что мы не можем переопределить их в классах, реализующих интерфейс.:

```
interface Printable {
   void print();
   static void read(){
      System.out.println("Read printable");
   }
}
```

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

```
public static void main(String[] args) {
   Printable.read();
}
```

16. Байтовые потоки InputStream и OutputStream. Консольный ввод и вывод Java.Символьные потоки данных. Абстрактные классы Writer, Reader.

Байтовые потоки InputStream и OutputStream:

Потоки байтов

Класс InputStream

Класс InputStream является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его основные методы:

int available(): возвращает количество байтов, доступных для чтения в потоке

void close(): закрывает поток

int read(): возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1

int read(byte[] buffer): считывает байты из потока в массив buffer. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число -1

int read(byte[] buffer, int offset, int length): считывает некоторое количество байтов, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента buffer[offset]. Метод возвращает число успешно прочитанных байтов.

long skip(long number): пропускает в потоке при чтении некоторое количество байт, которое равно number

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

void close(): закрывает поток

void flush(): очищает буфер вывода, записывая все его содержимое

void write(int b): записывает в выходной поток один байт, который представлен целочисленным параметром b

void write(byte[] buffer): записывает в выходной поток массив байтов buffer.

void write(byte[] buffer, int offset, int length): записывает в выходной поток некоторое число байтов, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

Консольный ввод и вывод Java:

Ввод с консоли

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

```
import java.util.Scanner;
public class Program {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Input a number: ");
        int num = in.nextInt();
        System.out.printf("Your number: %d \n", num);
        in.close();
    }
}
```

Так как класс Scanner находится в пакете java.util, то мы вначале его импортируем с помощью инструкции import java.util.Scanner.

Чтобы получить введенное число, используется метод in.nextInt();, который возвращает введенное с клавиатуры целочисленное значение.

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

next(): считывает введенную строку до первого пробела

nextLine(): считывает всю введенную строку

nextInt(): считывает введенное число int

Вывод на консоль

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

```
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello world!");
        System.out.println("Bye world...");
    }
}
```

В метод println передается любое значение, как правило, строка, которое надо вывести на консоль. И в данном случае мы получим следующий вывод:

```
Hello world!
Bye world...
```

При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод System.out.print(), который аналогичен println за тем исключением, что не осуществляет перевода на следующую строку.

```
public class Program {
    public static void main(String[] args) {
        System.out.print("Hello world!");
        System.out.print("Bye world...");
    }
}
```

Консольный вывод данной программы:

```
Hello world!Bye world...
```

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

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

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

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

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

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

%x: для вывода шестнадцатеричных чисел

%f: для вывода чисел с плавающей точкой

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

%c: для вывода одиночного символа

%s: для вывода строковых значений

Символьные потоки данных:

Символьные потоки данных в Java представлены классами Reader и Writer из пакета java.io. Они предназначены для работы с символами, где символом может быть, например, буква, цифра или знак препинания.

Некоторые основные классы символьных потоков данных в Java:

1. BufferedReader и BufferedWriter - предоставляют буферизацию ввода/вывода символов для увеличения производительности.

2. InputStreamReader и OutputStreamWriter - выполняют преобразование байтовых потоков в символьные потоки и обратно с использованием кодировки.

3. FileReader и FileWriter - позволяют работать с текстовыми файлами, читая символы из файла или записывая символы в файл.

4. StringReader и StringWriter - предоставляют возможность удобной работы с символьными данными, представленными в виде строк.

5. CharArrayReader и CharArrayWriter - работают с символьными данными, хранящимися в массиве символов.

Абстрактные классы Reader и Writer

Абстрактный класс Reader предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

absract void close(): закрывает поток ввода

int read(): возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1

int read(char[] buffer): считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

int read(CharBuffer buffer): считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1

absract int read(char[] buffer, int offset, int count): считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count

long skip(long count): пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Класс Writer определяет функционал для всех символьных потоков вывода. Его основные методы:

Writer append(char c): добавляет в конец выходного потока символ c. Возвращает объект Writer

Writer append(CharSequence chars): добавляет в конец выходного потока набор символов chars. Возвращает объект Writer

abstract void close(): закрывает поток

abstract void flush(): очищает буферы потока

void write(int c): записывает в поток один символ, который имеет целочисленное представление

void write(char[] buffer): записывает в поток массив символов

absract void write(char[] buffer, int off, int len) : записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off

void write(String str): записывает в поток строку

void write(String str, int off, int len): записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off

Функционал, описанный классами Reader и Writer, наследуется непосредственно классами символьных потоков, в частности классами FileReader и FileWriter соответственно, предназначенными для работы с текстовыми файлами.

17.Основные фреймворки и задачи, решаемые Spring

Основными фреймворками, входящими в состав Spring, являются:

1. Spring Core: Это основной модуль фреймворка Spring, который предоставляет основные функциональные возможности, такие как IoC контейнер, управление жизненным циклом бинов, внедрение зависимостей и другие.
2. Spring MVC: Этот модуль предоставляет инструменты для разработки веб-приложений на основе архитектурного шаблона Model-View-Controller (MVC). С помощью Spring MVC можно создавать веб-приложения и RESTful API.
3. Spring Data: Этот модуль предоставляет абстракцию и поддержку для взаимодействия с различными типами баз данных, включая реляционные и NoSQL. Spring Data позволяет упростить разработку слоя доступа к данным (Data Access Layer) и предоставляет гибкую и мощную модель управления данными.
4. Spring Security: Этот модуль предоставляет инструменты для обеспечения безопасности приложений. Он позволяет реализовывать механизмы аутентификации, авторизации, управления правами доступа и защиту от различных видов атак, таких как межсайтовый скриптинг (XSS) и подделка запросов между сайтами (CSRF).
5. Spring Boot: Этот модуль предоставляет простой и быстрый способ создания самостоятельных, готовых к работе приложений. Spring Boot включает в себя все необходимые зависимости и настройки, что позволяет разработчикам быстро развернуть приложение и сосредоточиться на реализации бизнес-логики.
6. Spring Cloud: Этот модуль предоставляет инструменты и библиотеки для разработки микросервисных приложений. Spring Cloud включает в себя такие компоненты, как сервис-регистратор, балансировщик нагрузки, конфигурационный сервер, цепочки фильтров и другие.

Spring Framework - это один из самых популярных фреймворков для разработки приложений на языке Java. Он предлагает множество функций и возможностей, которые позволяют упростить и ускорить процесс разработки. Spring Framework может быть использован для решения следующих задач:

1. Управление зависимостями: спринг предоставляет Inversion of Control (IoC) контейнер, который позволяет управлять зависимостями между классами.
2. Упрощенное взаимодействие с базой данных: спринг предоставляет модуль под названием Spring Data, который облегчает взаимодействие с различными типами баз данных, включая реляционные и NoSQL.
3. Разработка веб-приложений: спринг MVC (Model-View-Controller) предлагает мощную модель для разработки веб-приложений, позволяя разделить логику, представление и управление данными.
4. Безопасность: спринг предоставляет инструменты для обеспечения безопасности приложений, включая аутентификацию, авторизацию и защиту от атак.
5. Тестирование: спринг предлагает интеграцию с различными инструментами тестирования, такими как JUnit и Mockito, для обеспечения надежности и качества приложений.
6. Управление транзакциями: спринг предоставляет механизмы для управления и контроля транзакций в приложениях, включая транзакционные аспекты и управление транзакционными границами.
7. Микросервисная архитектура: спринг предлагает ряд инструментов и библиотек для разработки микросервисных приложений, включая Netflix OSS, Spring Cloud и Spring Boot.

18.Spring Inversion of Control (IoC) контейнер Spring

Inversion of Control (IoC), также известное как Dependency Injection (DI), является процессом, согласно которому объекты определяют свои зависимости, т.е. объекты, с которыми они работают, через аргументы конструктора/фабричного метода или свойства, которые были установлены или возвращены фабричным методом. Затем контейнер inject(далее "внедряет") эти зависимости при создании бина. Этот процесс принципиально противоположен, поэтому и назван Inversion of Control, т.к. бин сам контролирует реализацию и расположение своих зависимостей, используя прямое создание классов или такой механизм, как шаблон Service Locator.

Основными пакетами Spring Framework IoC контейнера являются org.springframework.beans и org.springframework.context. Интерфейс BeanFactory предоставляет механизм конфигурации по управлению любым типом объектов. ApplicationContext - наследует нитерфейс BeanFactory и добавляет более специфичную функциональность.

19.Dependency Injection (DI) в Spring

Dependency Injection (DI) - это паттерн программирования, который используется для управления зависимостями между объектами. В Spring Framework DI служит для обеспечения связи между классами и их зависимостями.

Spring предлагает два способа реализации DI: через конструктор (Constructor Injection) и через сеттеры (Setter Injection).

При использовании DI через конструктор, зависимые объекты передаются через параметры конструктора классов.

При использовании DI через сеттеры, зависимости устанавливаются через соответствующие сеттеры классов

20.Жизненный цикл объекта Bean Spring.

Жизненный цикл объекта Bean в Spring состоит из пяти основных этапов:

1. Инициализация (Instantiation): В этом этапе Spring создает экземпляр объекта Bean, используя его определение из контекста приложения. Здесь могут выполниться операции по созданию объекта и его настройке, такие как вызов конструктора и установка значений свойств.

2. Зависимости (Dependencies): После инициализации объекта Bean, Spring обрабатывает зависимости этого объекта. Это включает в себя поиск и внедрение необходимых зависимостей, таких как ссылки на другие объекты Bean или значения параметров.

3. Подготовка (Preparation): В этом этапе Spring подготавливает объект Bean для будущего использования. Это включает в себя вызов методов инициализации, если они определены, и установку любых других свойств.

4. Использование (Usage): После подготовки объект Bean может быть использован в приложении. Сам по себе он становится доступным и готов к выполнению своей функциональности.

5. Уничтожение (Destruction): В конце своего жизненного цикла объект Bean может быть уничтожен. Это происходит, например, при остановке приложения или при явном вызове методов уничтожения. Здесь может быть выполнено закрытие файловых дескрипторов, освобождение ресурсов и другие очистительные операции.

21.Конфигурация ApplicationContext с помощью xml в Spring

Конфигурация ApplicationContext с помощью XML в Spring включает несколько шагов:

1. Создание XML-файла конфигурации: Создайте новый XML-файл, который будет содержать конфигурацию ApplicationContext. Обычно этот файл называется applicationContext.xml или spring-config.xml.

2. Определение пространства имен и схемы: В верхней части XML-файла добавьте описание пространства имен и схемы для Spring. Например:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-
beans.xsd">
```

3. Определение бинов: Внутри тега <beans> определите бины (объекты), которые вы хотите создать и настроить. Например:

```
<bean id="user" class="com.example.User">
  <property name="name" value="John Doe" />
</bean>
```

Здесь мы определяем бин с идентификатором "user", который является экземпляром класса "com.example.User". Также мы задаем значение свойства "name" для этого бина.

4. Определение ApplicationContext: В конце XML-файла определите bean с id "applicationContext", который является экземпляром класса "org.springframework.context.support.ClassPathXmlApplicationContext". В качестве аргумента передайте путь к XML-файлу конфигурации. Например:

```
<bean id="applicationContext"
class="org.springframework.context.support.ClassPathXmlApplicationContext">
  <constructor-arg value="classpath:applicationContext.xml" />
</bean>
```

5. Создание объекта ApplicationContext: Теперь вы можете создать объект ApplicationContext, используя конфигурацию XML. В Java-коде это может выглядеть примерно так:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

22.Область видимости Bean в Spring

Одна из последних версий фреймворка Spring определяет 6 типов областей видимости:

Синглтон Singleton

Прототип Prototype

Запрос Requese

Сеанс Session

Заявление Global session

Последние четыре упомянутые области, request, session, application и websocket , доступны только в веб-приложении.

singleton

Определяет один единственный бин для каждого контейнера Spring IoC (используется по умолчанию).

prototype

Позволяет иметь любое количество экземпляров бина.

request

Создаётся один экземпляр бина на каждый HTTP запрос. Касается исключительно ApplicationContext.

session

Создаётся один экземпляр бина на каждую HTTP сессию. Касается исключительно ApplicationContext.

global-session

Создаётся один экземпляр бина на каждую глобальную HTTP сессию. Касается исключительно ApplicationContext.

Singleton

Если мы устанавливаем свойству scope значение singleton, то в это случае контейнер Spring IoC создаёт только один экземпляр объекта определённого в бине. Этот экземпляр помещается в кэш таких же бинов (синглтонов) и все последующие вызовы бина с таким именем будут возвращать объект из кэша.

По умолчанию область видимости устанавливается singleton, но если вы хотите акцентировать внимание на этом, то можно использовать такую запись:

<bean id = "someBean" scope = "singleton">
<!--some code to configure bean-->
</bean>

Prototype

Когда мы присваиваем свойству **scope** значение **prototype**, контейнер Spring IoC создаёт новый экземпляр бина на каждый полученный запрос.

Бин с областью видимости prototype можно создать следующим образом:В то время как

<bean id = "someBean" scope = "prototype">
<!--some code to configure bean-->
</bean>

23.Фабричные или factory-методы в Spring.

В Spring существуют два основных подхода к созданию экземпляров объектов - использование конструкторов и использование фабричных методов.

Фабричные методы представлены специальными методами, которые создают и возвращают экземпляры классов. Эти методы могут быть статическими или нестатическими. Spring обеспечивает возможность использования фабричных методов для создания бинов в контейнере.

С помощью фабричных методов можно реализовать различные сценарии создания объектов, такие как создание объекта на основе параметров, внешних ресурсов или других логических условий. Это полезно в случаях, когда непосредственное создание экземпляров классов представляет сложность или требует дополнительной логики.

Для использования фабричных методов в Spring можно использовать аннотацию @Bean или XML-конфигурацию. Аннотация @Bean позволяет определить метод как фабричный метод, который возвращает объект, который будет управляться Spring контейнером. В XML-конфигурации используется элемент <bean> с атрибутом factory-method, указывающим на метод, создающий объект.

24.Конфигурация ApplicationContext с помощью аннотаций в Spring

В Spring можно использовать аннотации для конфигурации ApplicationContext. Для этого необходимо выполнить следующие шаги:

1. Убедитесь, что у вас есть зависимости Spring, включая spring-context, в вашем проекте.
2. Создайте класс конфигурации, который будет отмечен аннотацией @Configuration. Это можно сделать путем добавления аннотации @Configuration к вашему классу.

```
@Configuration
public class AppConfig {
}
```

3. Определите бины, которые вы хотите создать, и пометьте их аннотацией @Bean. Это можно сделать посредством добавления методов в класс конфигурации и пометки их аннотацией @Bean.

```
@Configuration
public class AppConfig {
   @Bean
   public UserService userService() {
      return new UserServiceImpl();
   }
   @Bean
   public UserRepository userRepository() {
      return new UserRepositoryImpl();
   }
}
```

4. Установите путь для сканирования компонентов, чтобы Spring мог найти вашу класс конфигурации и создать бины. Вы можете сделать это, добавив аннотацию @ComponentScan с указанием базового пакета сканирования.

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
   // ...
}
```

5. Создайте объект ApplicationContext, используя ваш класс конфигурации.

```
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
```

6. Воспользуйтесь полученным объектом ApplicationContext для получения бинов.

```
UserService userService = context.getBean(UserService.class);
```

25.Связывание в Spring, аннотация @Autowired.

Связывание (или инъекция зависимостей) в Spring является механизмом, который позволяет управлять зависимостями компонентов в приложении. Он обеспечивает возможность создания и внедрения зависимостей для других компонентов, которые их используют.

автосвязывание (autowiring) – механизм, с помощью которого Spring автоматически «удовлетворяет» зависимости компонентов (to satisfy a dependency).

Тот участок кода, где контейнеру необходимо осуществить внедрение зависимости, аннотируется с помощью аннотации @Autowired

```
@Component
public class User {
    private Sender sender;
    @Autowired
    public void setSender(Sender sender) {
        this.sender = sender;
    }
}
```

26.Архитектурный стиль REST

REST— это акроним, сокращение от английского Representational State Transfer — передача состояния представления.

Это архитектурный стиль взаимодействия компонентов распределенной системы в компьютерной сети. Проще говоря, REST определяет стиль взаимодействия (обмена данными) между разными компонентами системы, каждая из которых может физически располагаться в разных местах.
Данный архитектурный стиль представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной системы. Эти ограничения иногда называют принципами REST. Их немного, всего 6 штук

1. Приведение архитектуры к модели клиент-сервер
В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на “клиент” и “сервер” позволяет им развиваться независимо друг от друга.
2. Отсутствие состояния
Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента и наоборот. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.
3. Кэширование
Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные.
4. Единообразие интерфейса
К фундаментальным требованиям REST архитектуры относится и унифицированный, единообразный интерфейс. Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-

серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом

5. Слои

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.

6. Код по требованию (необязательное ограничение)

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев.

Преимущества, которые дает REST

У приложений, которые соблюдают все вышеперечисленные ограничения, есть такие преимущества: надёжность (не нужно сохранять информацию о состоянии клиента, которая может быть утеряна);

- производительность (за счёт использования кэша);
- масштабируемость;
- прозрачность системы взаимодействия;
- простота интерфейсов;
- портативность компонентов;
- лёгкость внесения изменений;
- способность эволюционировать, приспосабливаясь к новым требованиям.

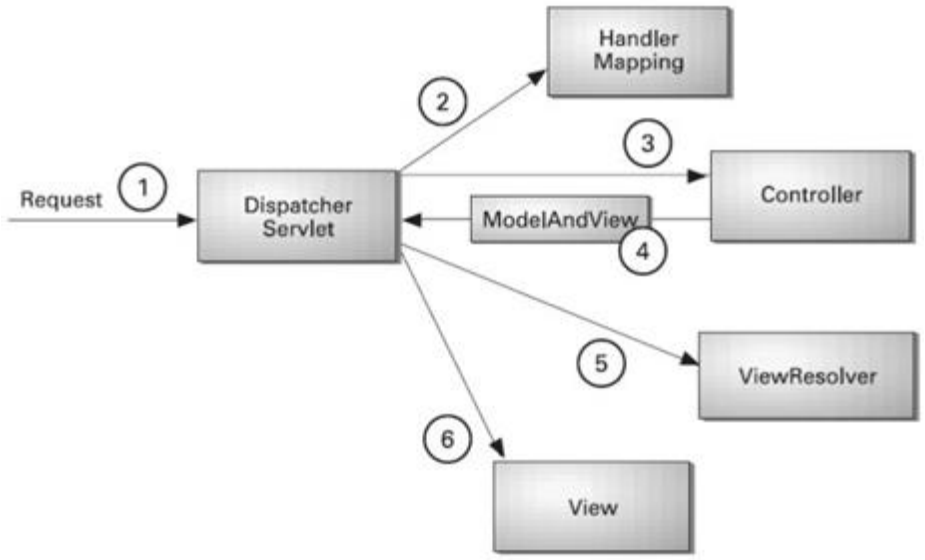
27.Spring Web-MVC, основная схема и логика работы.

Теоретические сведения

Spring MVC – веб-фреймворк, призванный упростить разработку веб-приложений. Опираясь на шаблон модель–представление–контроллер (Model-View-Controller, MVC), фреймворк Spring MVC помогает строить веб-приложения, столь же гибкие и слабо связанные, как сам фреймворк Spring.

Схема работы фреймворка Spring MVC

Схема работы фреймворка выглядит следующим образом:



Краткое описание схемы работы Spring MVC звучит следующим образом:

- вначале **DispatcherServlet** (диспетчер сервлетов) получает запрос, далее он смотрит свои настройки, чтобы понять какой контроллер использовать (на рисунке **Handler Mapping**);
- после получения имени контроллера запрос передается на обработку в этот контроллер (на рисунке **Controller**). В контроллере происходит обработка запроса и обратно посылается **ModelAndView** (модель — сами данные; view (представление) — как эти данные отображать);
- **DispatcherServlet** на основании полученного **ModelAndView**, должен определить, какое представление будет выводить данные. Для этого используется **арбитр представлений** (**View Resolver**), который на основании полученного логического имени представления возвращает ссылку на файл **View**;
- в представление передаются данные (Model) и обратно, если необходимо, посылается ответ от представления.

Логика работы:

1. Клиент отправляет HTTP-запрос на сервер.
2. Диспетчер сервера, который является центральным компонентом фреймворка, принимает запрос и определяет, какой контроллер должен обработать этот запрос, основываясь на маппинге URL-адресов.
3. Диспетчер передает запрос соответствующему контроллеру для дальнейшей обработки.
4. Контроллер получает запрос и обрабатывает его. Это может включать выполнение бизнес-логики, вызов сервисных методов, извлечение данных из базы данных и т.д.
5. Контроллер формирует модель данных, которая будет передана в представление для отображения.
6. Контроллер выбирает подходящее представление, которое будет использоваться для отображения данных. Это может быть JSP, Freemarker, Thymeleaf, AngularJS или другие представления.
7. Контроллер передает модель данных в представление.
8. Представление получает модель данных и использует их для формирования вывода - HTML-страницы, JSON-ответа или любого другого формата.
9. Сформированный ответ отправляется обратно на сервер.
10. Диспетчер сервера принимает ответ и отправляет его обратно клиенту

28. Класс DispatcherServlet, его функции.

Spring MVC, как и многие другие веб-фреймворки, разработан по проектному шаблону "единой точки входа (front controller)", где центральный Servlet, DispatcherServlet, обеспечивает общий алгоритм обработки запросов, а фактическая работа выполняется настраиваемыми компонентами-делегатами. Эта модель является гибкой и поддерживает различные рабочие процессы.

DispatcherServlet, как и любой другой Servlet, необходимо объявлять и отображать в соответствии со спецификацией сервлетов с помощью конфигурации Java или в web.xml. В свою очередь, DispatcherServlet использует конфигурацию Spring для обнаружения компонентов-делегатов, необходимых ему для отображения запросов, распознавание представлений, обработки исключений и т. д.

Функции:

1. Обработка входящего запроса: DispatcherServlet слушает входящие HTTP-запросы и решает, какой обработчик контроллеров (controller handler) должен обрабатывать данный запрос. Он рассматривает различные факторы, такие как URL, метод запроса, заголовки и параметры, чтобы определить, какой контроллер следует вызвать.
2. Управление жизненным циклом контроллеров: DispatcherServlet создает экземпляры контроллеров и управляет их жизненным циклом. Он может создать новый экземпляр контроллера для каждого запроса (при использовании прототипной области видимости), либо использовать существующий экземпляр (при использовании других областей видимости, таких как синглтон).
3. Осуществление взаимодействия с контроллерами: DispatcherServlet вызывает методы контроллеров для обработки запроса. Это может быть метод, помеченный аннотацией @RequestMapping, или другие аннотированные методы, такие как @GetMapping, @PostMapping и т. д. Контроллер возвращает объект ModelAndView, который содержит информацию о представлении (view) и данных для передачи в представление.
4. Обработка исключений: DispatcherServlet также отвечает за обработку исключений, возникающих в процессе обработки запроса. Он может перехватывать исключения и принимать

решения о том, какие действия предпринять (например, отобразить страницу ошибки или выполнить перенаправление).

29.Маппинг в Spring.

Маппинг в Spring относится к процессу связывания (mapping) HTTP-запросов с методами обработки в вашем приложении. Это позволяет определить, какой метод должен быть вызван при получении определенного запроса.

В Spring есть несколько способов определить маппинг:

1. Аннотация @RequestMapping: Эта аннотация может быть применена к методу контроллера и указывает, на какие HTTP-методы и пути должен отвечать данный метод. Например:

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello() {
   return "Hello, World!";
}
```

В этом примере метод hello() будет вызван при получении GET запроса на URL '/hello'.

2. Аннотации для конкретных HTTP-методов: Spring предоставляет аннотации, такие как @GetMapping, @PostMapping, @PutMapping, @DeleteMapping и другие, которые облегчают маппинг только для определенных HTTP-методов. Например:

```
@GetMapping("/hello")
public String hello() {
   return "Hello, World!";
}
```

3. Параметр в URL: Можно использовать переменные в URL для более гибкого маппинга. Например:

```
@GetMapping("/hello/{name}")
public String hello(@PathVariable String name) {
   return "Hello, " + name + "!";
}
```

В этом примере значение переменной {name} будет извлечено из URL и передано в метод hello().

4. Аннотация @RequestParam: Эта аннотация позволяет получить значения параметров запроса. Например:

```
@GetMapping("/hello")
public String hello(@RequestParam("name") String name) {
   return "Hello, " + name + "!";
}
```

В этом примере значение параметра name будет извлечено из запроса и передано в метод hello().

30.Интерфейсы HttpServletRequest и HttpServletResponse

Класс HttpServlet

Класс Servlet существует для стандартизации работы сервлета и контейнера. Непосредственно с этим классом программисты не работают. Ну или работают очень редко. Чаще всего используется класс HttpServlet, унаследованный от Servlet'а.

У этого класса есть несколько методов, которые нам пригодятся. Ты будешь часто их использовать:

	Метод	Описание
1	`init()`	Вызывается один раз при загрузке сервлета
2	`destroy()`	Вызывается один раз при выгрузке сервлета
3	`service(HttpRequest, HttpResponse)`	Вызывается для каждого нового запроса к сервлету
4	`doGet(HttpRequest, HttpResponse)`	Вызывается для каждого нового GET-запроса к сервлету
5	`doPost(HttpRequest, HttpResponse)`	Вызывается для каждого нового POST-запроса к сервлету
6	`doHead(HttpRequest, HttpResponse)`	Вызывается для каждого нового HEAD-запроса к сервлету
7	`doDelete(HttpRequest, HttpResponse)`	Вызывается для каждого нового DELETE-запроса к сервлету
8	`doPut(HttpRequest, HttpResponse)`	Вызывается для каждого нового PUT-запроса к сервлету

Методы init() и destroy() унаследованы от класса Servlet. Поэтому если ты решишь переопределить их в своем сервлете, тебе нужно будет так же вызвать их реализацию из базового класса. Для этого используется команда super.имяМетода().

Метод service(HttpServletRequest, HttpServletResponse)

Если смотреть на обработку клиентского запроса с точки зрения сервлета, то дела обстоят примерно так.

Для каждого клиентского запроса контейнер (веб-сервер) создает объекты HttpServletRequest и HttpServletResponse, а затем вызывает метод service(HttpServletRequest request, HttpServletResponse response) у соответствующего сервлета. В него передаются эти объекты, чтобы метод мог взять нужные данные из request'а и положить результат работы в response.

У метода service() есть реализация по умолчанию. Если ее не переопределить, то выполняться будет именно она. Вот что он делает.

Метод service() определяет из request'а тип HTTP-метода (GET, POST, …) и вызывает метод соответствующий запросу.

	Метод	Описание
1	`service(HttpRequest, HttpResponse)`	Вызывается для каждого нового запроса к сервлету
2	`doGet(HttpRequest, HttpResponse)`	Вызывается для каждого нового GET-запроса к сервлету
3	`doPost(HttpRequest, HttpResponse)`	Вызывается для каждого нового POST-запроса к сервлету
4	`doHead(HttpRequest, HttpResponse)`	Вызывается для каждого нового HEAD-запроса к сервлету
5	`doDelete(HttpRequest, HttpResponse)`	Вызывается для каждого нового DELETE-запроса к сервлету
6	`doPut(HttpRequest, HttpResponse)`	Вызывается для каждого нового PUT-запроса к сервлету

31.Архитектурный стиль CRUD, его соответствие REST и HTTP

Архитектурный стиль CRUD (Create, Read, Update, Delete) описывает базовые операции, которые можно выполнять с данными: создание (Create), чтение (Read), обновление (Update) и удаление (Delete).

REST (Representational State Transfer) - это набор принципов, которые описывают, как должны быть организованы системы для работы с распределенными данными в сети. REST основан на использовании стандартных HTTP-методов (GET, POST, PUT, DELETE) для работы с ресурсами.

Соответствие между CRUD и REST заключается в следующем:

- Создание (Create) соответствует методу POST в REST, который используется для создания новых ресурсов.

- Чтение (Read) соответствует методу GET в REST, который используется для получения данных из ресурсов.

- Обновление (Update) соответствует методу PUT или PATCH в REST, который используется для обновления существующих ресурсов.

- Удаление (Delete) соответствует методу DELETE в REST, который используется для удаления ресурсов.

Таким образом, CRUD операции могут быть реализованы с использованием соответствующих HTTP-методов в REST архитектуре.

32.Шаблон Data Access Object (DAO).

DAO (data access object) - шаблон проектирования для сохранения объектов в базе данных. DAO абстрагирует и инкапсулирует весь доступ к источнику данных, он также управляет соединением с источником данных для получения и хранения данных.

Если говорить упрощенно, DAO - это специальный объект, который предоставляет интерфейс доступа к локальной базе данных. Вместо прямого общения с базой данных для выполнения различных операций, мы будем вызывать методы DAO.

Для создания DAO необходимо объявить интерфейс, который будет помечен аннотацией @Dao. Внутри интерфейса мы должны объявить методы, которые описывают нужные нам операции с БД, после чего методы нужно пометить аннотациями, чтобы Room смогла сгенерировать нужный код для взаимодействия с БД

```
@Dao
public interface NoteDAO {
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   void insertNote(Note note);
   @Delete
   void deleteNote(Note note);
   @Query("DELETE FROM notes WHERE id=:id")
   void deleteNoteById(int id);
   @Query("SELECT * FROM notes WHERE id=:id")
   Note getNoteById(int id);
   @Query("DELETE FROM notes")
   void deleteAll();
   @Query("SELECT * FROM notes ORDER BY dateUpdate DESC")
   List<Note> getAll();
}
```

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

ORM (Object-Relational Mapping, «объектно-реляционное отображение») — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Проще говоря, **ORM – это прослойка, посредник между базой данных и объектным кодом**.

Основные понятия ORM в Java:

1. Сущности (Entities): Они представляют объекты, с которыми вы хотите работать в базе данных. Каждая сущность обычно соответствует одной таблице в базе данных и имеет свои атрибуты (поля), которые могут быть привязаны к столбцам в таблице.
2. Аннотации (Annotations): Они используются для указания маппинга между сущностями и таблицами в базе данных. Например, с помощью аннотации @Entity вы можете указать, что класс является сущностью, а с помощью аннотации @Id - указать поле, которое является первичным ключом в таблице.
3. Repository (Репозиторий): Репозиторий предоставляет интерфейс для доступа к данным в базе данных. Он может содержать методы для поиска, сохранения, обновления и удаления сущностей.
4. Связи (Relationships): ORM позволяет устанавливать связи между сущностями. Это может быть связь "один-к-одному", "один-ко-многим" или "многие-к-многим". Например, с помощью аннотации @OneToOne на поле вы можете указать связь "один-к-одному" между двумя сущностями.
5. Слой доступа к данным (Data Access Layer): ORM позволяет создавать слой доступа к данным, который абстрагирует приложение от конкретной реляционной базы данных. Вы можете работать с объектами и вызывать методы сохранения, поиска или обновления, а ORM будет заботиться о выполнении соответствующих SQL-запросов.

34.Спецификация Java Persistence API (JPA).

Java Persistence API (JPA) - это спецификация Java, которая определяет способ работы с реляционными базами данных в объектно-ориентированной среде. JPA предоставляет набор аннотаций и API для управления объектами и их хранением в базе данных.

Основные компоненты JPA включают в себя:

- Entity: объекты, представляющие данные в базе данных. Они помечаются аннотацией @Entity и имеют связанные с ними атрибуты и отношения.
- EntityManager: основной интерфейс JPA, позволяющий создавать, изменять, удалять и извлекать объекты из базы данных.
- Persistence Unit: конфигурационная единица, описывающая параметры подключения к базе данных и определяющая управляемые сущности.
- JPQL (Java Persistence Query Language): объектно-ориентированный язык запросов, позволяющий разработчикам выполнять запросы к базе данных, не зависимо от используемой СУБД.

35.Архитектура ORM Java Persistence API (JPA).

Архитектура ORM

1. Entity: Это основной компонент в JPA, представляющий отображаемый объект в базе данных. Entity классы декларируются с использованием аннотации @Entity и могут содержать аннотации для указания таблицы, столбцов и связей с другими Entity классами.

2. EntityManager: EntityManager является основным интерфейсом для выполнения операций над Entity классами. Он предоставляет методы для управления жизненным циклом сущностей, выполнения транзакций и выполнения операций с базой данных, таких как сохранение, обновление и удаление объектов.

3. Persistence: Класс Persistence предоставляет статические методы для получения экземпляра EntityManagerFactory, который является ключевым компонентом в архитектуре JPA. EntityManagerFactory используется для создания EntityManager экземпляров.

4. EntityManagerFactory: EntityManagerFactory представляет фабрику EntityManager экземпляров. Он создается с использованием Persistence класса и может быть настроен с помощью файла конфигурации или с использованием аннотаций. EntityManagerFactory обеспечивает соединение с базой данных и предоставляет EntityManager экземпляры для каждого запроса или транзакции.

5. JPQL (Java Persistence Query Language): JPQL - это объектно-ориентированный язык запросов, подобный SQL, который используется в JPA для выполнения запросов к базе данных. JPQL позволяет работать с Entity классами, а не с таблицами и столбцами, что обеспечивает абстракцию от деталей реализации базы данных.

36.Основные аннотации Java Persistence API (JPA).

1. @Entity — Указывает, что данный бин (класс) является сущностью.
2. @Table — указывает на имя таблицы, которая будет отображаться в этой сущности.
3. @Column — указывает на имя колонки, которая отображается в свойство сущности.
4. @Id — id колонки
5. @GeneratedValue — указывает, что данное свойство будет создаваться согласно указанной стратегии.
6. @Version — управление версией в записи сущности. При изменении записи увеличится на 1.
7. @OrderBy — указание сортировки.
8. @Embeddable и @Embedded («встроенный») — Определяет класс, экземпляры которого хранятся как неотъемлемая часть исходного объекта. Каждый из @Embedded экземпляров сопоставляется с таблицей базы данных сущности.
9. @OneToOne — указывает на связь между таблицами «один к одному».
10. @ManyToOne — указывает на связь многие к одному.
11. @JoinColumn — применяется когда внешний ключ находится в одной из сущностей. Может применяться с обеих сторон взаимосвязи. Но рекомендуется применять в сущности, которая является владельцем физической информации (обычно сторона @ManyToOne). ManyToOne (часто) является стороной-владельцем в двунаправленных связях и таким образом противоположная сторона использует @OneToMany(mappedBy=..).
12. @JoinTable — указывает на связь с таблицей
13. @OneToMany — указывает на связь один ко многим. Применяется с другой стороны от сущности с @ManyToOne
14. @ManyToMany — связь многие ко многим.
15. @MapsId — связывает одну колонку с другой. Работает с @Id и @EmbeddedId.

37.Библиотека Hibernate, основные аннотации

Hibernate – это ORM фреймворк для Java с открытым исходным кодом. Эта технология является крайне мощной и имеет высокие показатели производительности.

Hibernate создаёт связь между таблицами в базе данных (далее – БД) и Java-классами и наоборот. Это избавляет разработчиков от огромного количества лишней, рутинной работы, в которой крайне легко допустить ошибку и крайне трудно потом её найти.

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 аттрибуты, с помощью которых мы можем указать имя таблицы, каталог, БД и уникальность столбцов в таблец 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 паметры, как strategy и generator.

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

Наиболее часто используемые аттрибуты аннотации @Column такие:

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

38.Объявление сущности и таблицы в Hibernate.

Для объявления сущности и таблицы в Hibernate необходимо выполнить следующие шаги:

1. Создайте класс, который будет представлять сущность. Этот класс должен быть аннотирован аннотацией @Entity.

```
@Entity
public class EntityName {
   // поля сущности
   // геттеры и сеттеры
}
```

2. Определите первичный ключ сущности, используя аннотацию @Id. Можно использовать автоматическую генерацию первичного ключа с помощью аннотации @GeneratedValue.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

3. Аннотируйте поля сущности, которые соответствуют столбцам таблицы, аннотацией @Column. Укажите дополнительные свойства, такие как имя столбца, длину и т.д., если это необходимо.

```
@Column(name = "column_name", length = 50, nullable = false)
private String columnName;
```

4. Если необходимо установить связь с другой сущностью, используйте соответствующие аннотации, такие как @OneToOne, @OneToMany, @ManyToOne или @ManyToMany.

```
@OneToOne
@JoinColumn(name = "related_entity_id")
private RelatedEntity relatedEntity;
```

5. Создайте файл xml-конфигурации Hibernate (hibernate.cfg.xml), указав в нем информацию о подключении к базе данных и настройках Hibernate.

```
<hibernate-configuration>
   <session-factory>
      <property
name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
      <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/database_name</pr
operty>
      <property name="hibernate.connection.username">username</property>
      <property name="hibernate.connection.password">password</property>
```

```
        <property name="hibernate.show_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

6. В конфигурационном классе (например, классе с методом main) создайте экземпляр SessionFactory, используя Configuration класс Hibernate, и выполните метод buildSessionFactory() для создания фабрики сессий.

```
Configuration configuration = new Configuration().configure("hibernate.cfg.xml");
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

7. Для создания таблицы и схемы базы данных в Hibernate можно использовать автоматическую схему, если включить соответствующую опцию в конфигурационном файле:

```
<property name="hibernate.hbm2ddl.auto">create</property>
```

Это позволит Hibernate автоматически создать таблицы и схему базы данных на основе объявленных сущностей.

39.Интерфейс Session в Hibernate.

Session

Однопоточный объект, устанавливающий связь между объектами/сущностями приложения и базой данных. Сессия создается при необходимости работы с БД и ее необходимо закрыть сразу же после использования. Экземпляр Session является интерфейсом между кодом в java приложении и hibernate framework, предоставляя методы для операций CRUD.

Интерфейс Session в Hibernate представляет собой концептуальную сущность, которая предоставляет методы для выполнения операций базы данных в рамках транзакции.

Методы Session

- save(Object entity): сохраняет новую сущность в базе данных.

- update(Object entity): обновляет существующую сущность в базе данных.

- delete(Object entity): удаляет существующую сущность из базы данных.

- get(Class theClass, Serializable id): извлекает сущность из базы данных по указанному идентификатору.

- createQuery(String queryString): создает запрос HQL (Hibernate Query Language) для выполнения на базе данных.

40.Ассоциация сущностей в Hibernate

Для определения связей между сущностями Hibernate использует аннотации @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

1. Однонаправленная связь между объектами-сущностями в Hibernate может быть реализована с помощью аннотаций @ManyToOne и @OneToMany. Например, связь "многие-к-одному" может быть установлена, когда у одной сущности может быть несколько связанных с ней записей другой сущности.

2. Двусторонняя связь между объектами-сущностями можно устанавливать с помощью аннотаций @OneToOne и @OneToMany (с параметром mappedBy), а также с помощью промежуточной сущности, например, с помощью сущности-соединения или таблицы-связи.

3. Многие-к-многим связи между объектами-сущностями реализуются с помощью аннотаций @ManyToMany и @JoinTable. При многие-к-многим связи создается специальная таблица-связь, которая содержит в себе ключи обоих связанных объектов.

41.Spring Boot: определение, характеристики, преимущества

Spring Boot - это популярный фреймворк для создания веб-приложений с использованием Java. Это часть фреймворка Spring, которая представляет собой набор инструментов и библиотек для создания приложений корпоративного уровня. Spring Boot упрощает создание автономных приложений производственного уровня, которые можно легко развернуть и запустить с минимальной конфигурацией

Характеристики

1. Простота использования: Spring Boot поставляется с удобными средствами автоматической конфигурации, которые позволяют создавать приложения без необходимости настройки их вручную.

2. Быстрая разработка: Spring Boot предлагает широкий спектр функциональности "из коробки", что позволяет разработчикам сосредоточиться на создании бизнес-логики приложения, не тратя времени на настройку инфраструктуры.

3. Встроенный веб-сервер: Spring Boot поставляется с встроенным сервером приложений, что позволяет разработчикам мгновенно запускать и проверять свои приложения без необходимости установки и конфигурации отдельного сервера.

Преимущества:

- 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

Spring Initializr - это онлайн-инструмент, который предоставляет возможность генерировать готовые проекты на основе Spring Boot. Вот некоторые особенности и преимущества применения Spring Initializr:

1. Простота использования: Spring Initializr обеспечивает простой и интуитивно понятный интерфейс, который позволяет быстро и легко создавать проекты на основе Spring Boot.

2. Гибкость: Spring Initializr предоставляет множество параметров для настройки проекта, таких как выбор версии Spring Boot, добавление различных зависимостей и модулей, настройка среды выполнения и многое другое. Это позволяет настроить проект по своим потребностям.

3. Автоматическая конфигурация: Spring Initializr генерирует проект с преднастроенной файлом конфигурации, который автоматически настраивает различные части приложения, такие как база данных, веб-сервер, безопасность и т.д. Это помогает ускорить разработку, так как большая часть необходимой конфигурации уже готова.

4. Удобная интеграция с различными инструментами разработки: Spring Initializr позволяет скачать сгенерированный проект в виде архива проекта или скопировать ссылку на стартовый код в систему контроля версий, такую как Git. Это упрощает интеграцию с различными инструментами разработки и позволяет эффективно управлять проектом.

5. Обновления зависимостей: Spring Initializr предоставляет возможность обновлять зависимости проекта, такие как Spring Boot или другие фреймворки, при помощи простого интерфейса. Это позволяет легко обновлять проект до последних версий и получать новые функциональные возможности и исправления ошибок.

43.Структура фреймворка JUnit

JUnit — это фреймворк автоматического тестирования вашего хорошего или не совсем хорошего кода/

Структура:

1. Аннотации: JUnit использует аннотации для определения различных элементов тестов, таких как тестовые методы, инициализационные методы и т.д. Некоторые из наиболее часто используемых аннотаций включают @Test, @Before, @After, @BeforeClass, @AfterClass и @Ignore.
2. Assert-методы: JUnit предоставляет множество Assert-методов, которые позволяют утверждать ожидаемые результаты тестов. Эти методы используются для сравнения фактического значения с ожидаемым и могут использоваться для проверки различных условий, таких как равенство, неравенство, исключения и т. д.
3. Test Runners: Test runners в JUnit запускают тесты и обрабатывают результаты. Они обеспечивают исполнение и управление тестами, включая передачу данных в тесты и обработку исключений. Некоторые из наиболее распространенных раннеров в JUnit включают JUnitCore и Runner.
4. Test Suites: Test suites используются для объединения нескольких тестовых классов вместе и запуска этих тестов как единую единицу. Они позволяют легко добавлять, удалять или настраивать наборы тестов для выполнения.
5. Rules: Rules в JUnit позволяют настраивать и изменять поведение тестов. Они могут быть использованы для внедрения дополнительной логики до или после выполнения теста, установки временных файлов и т. д.

44.JUnit аннотации @Test, @DisplayName

Аннотация @Test используется для пометки тестирования.

Это значит, что при запуске программы код будет выполняться, как тест, тестовый случай

```
class CalculatorTest {
        @Test
        public void add() {
        }
        @Test
        public void sub() {
       }
        @Test
        public void mul() {
        }
        @Test
        public void div() {
        }
}
```

Данная аннотация находится в модуле **junit-jupiter-api**

Аннотация @DisplayName
И наконец, каждому тесту можно задавать его имя. Бывает удобно, если тестов очень много и вы создаете специальные сценарии (подмножества) тестов. Для этого есть специальная аннотация @DisplayName

45.JUnit аннотации @BeforeEach, @AfterEach.

JUnit аннотации @BeforeEach и @AfterEach используются для выполнения определенных действий перед и после каждого тестового метода соответственно.

1. Аннотация @BeforeEach указывает на метод, который Junit будет вызывать перед каждым тестовым методом. Этот метод может быть использован, например, для инициализации данных или настройки окружения перед каждым тестом.

2. Аннотация @AfterEach указывает на метод, который будет вызываться каждый раз после очередного тестового метода, и подчищать использованные ресурсы, писать что-то в лог и т. П

46.Тестовые классы и методы JUnit.

В JUnit тесты организованы в виде классов, каждый из которых содержит несколько тестовых методов. Каждый тестовый метод должен быть отмечен аннотацией @Test

@Test метода getAllUsers() – это метод который должен вернуть всех пользователей. Тест будет выглядеть примерно так:

```
@Test
public void getAllUsers() {
   //создаем тестовые данные
   User user = new User("Евгений", 35, Sex.MALE);
   User user1 = new User("Марина", 34, Sex.FEMALE);
   User user2 = new User("Алина", 7, Sex.FEMALE);
   //создаем список expected и заполняем его данными нашего метода
   List<User> expected = User.getAllUsers();
   //создаем список actual в него помещаем данные для сравнения
   //то что мы предпологиаем метод должен вернуть
   List<User> actual = new ArrayList<>();
   actual.add(user);
   actual.add(user1);
   actual.add(user2);
   //запускаем тест, в случае если список expected и actual не будут равны
   //тест будет провален, о результатах теста читаем в консоли
   Assert.assertEquals(expected, actual);
}
```

Тестовые классы в JUnit - это классы, в которых содержатся тестовые методы для тестирования кода

JUnit также предоставляет ряд дополнительных аннотаций для управления тестовым потоком, таких как @Before, @After, @BeforeClass и @AfterClass.

Например, аннотация @Before позволяет указать метод, который будет выполняться перед каждым тестовым методом, а аннотация @After - метод, который будет выполняться после каждого тестового метода. Аннотации @BeforeClass и @AfterClass позволяют указать методы, которые будут выполняться единожды, перед началом всех тестов и после их окончания соответственно

47.Утверждения JUnit. Класс Assert

Ассерты (asserts) — это специальные проверки, которые можно вставить в разные места кода. Их задача определять, что что-то пошло не так. Вернее, проверять, что все идет как нужно. Вот это “как нужно” они и позволяют задать различными способами.

Тут важен порядок сравнения, ведь JUnit в итоговом отчете напишет что-то типа “получено значение 1, а ожидалось 3”. Общий формат такой проверки имеет вид:

assertEquals(эталон, значение)

Ниже приведен список самых популярных методов — ассертов:

assertEquals	Проверяет, что два объекта равны
assertArrayEquals	Проверяет, что два массива содержат равные значения
assertNotNull	Проверяет, что аргумент не равен null
assertNull	Проверяет, что аргумент равен null
assertNotSame	Проверят, что два аргумента — это не один и тот же объект
assertSame	Проверят, что два аргумента — это один и тот же объект
assertTrue	Проверяет, что аргумент равен *true*
assertFalse	Проверяет, что аргумент равен *false*

***assertAll*c**

Одно из новых утверждений, введенных в JUnit 5, -*assertAll*.

Это утверждение позволяет создавать сгруппированные утверждения, где все утверждения выполняются и их ошибки сообщаются вместе. В деталях, это утверждение принимает заголовок, который будет включен в строку сообщения для*MultipleFailureError*, и*Stream* для*Executable*..

Давайте определим сгруппированное утверждение:

```
@Test
public void givenMultipleAssertion_whenAssertingAll_thenOK() {
    assertAll(
      "heading",
      () -> assertEquals(4, 2 * 2, "4 is 2 times 2"),
```

```
    () -> assertEquals("java", "JAVA".toLowerCase()),
    () -> assertEquals(null, null, "null is equal to null")
  );
}
```

assertTimeout* и *assertTimeoutPreemptively

В случае, если мы хотим утверждать, что выполнение предоставленного*Executable* заканчивается раньше заданного*Timeout*, мы можем использовать утверждение*assertTimeout*:

```
@Test
public void whenAssertingTimeout_thenNotExceeded() {
  assertTimeout(
    ofSeconds(2),
    () -> {
      // code that requires less then 2 minutes to execute
      Thread.sleep(1000);
    }
  );
}
```

***assertThrows*с**

Чтобы повысить простоту и удобочитаемость, новое утверждение *assertThrows* позволяет нам ясный и простой способ утверждать, генерирует ли исполняемый файл исключение указанного типа.

Давайте посмотрим, как мы можем утверждать возникшее исключение:

```
@Test
void whenAssertingException_thenThrown() {
  Throwable exception = assertThrows(
    IllegalArgumentException.class,
    () -> {
       throw new IllegalArgumentException("Exception message");
    }
  );
  assertEquals("Exception message", exception.getMessage());
}
```

48.Тестирование исключений JUnit

Для тестирования исключений в JUnit существует аннотация @Test и метод assertThrows().

Пример использования:

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
public class ExceptionTest {
   // Тестируемый метод, который может бросить исключение
   public void throwException() throws IllegalArgumentException {
      throw new IllegalArgumentException();
   }
   @Test
   public void testException() {
      assertThrows(IllegalArgumentException.class, () -> {
         throwException(); // метод throwException() должен бросить IllegalArgumentException
      });
   }
}
```

В этом примере в методе testException() мы ожидаем, что метод throwException() бросит исключение IllegalArgumentException. Если исключение не будет брошено, тест не пройдет, и вы получите ошибку.
Важно указывать ожидаемый тип исключения в assertThrows(), чтобы JUnit мог корректно проверить, что исключение было брошено.

49.Генератор документирования Javadoc. Виды комментариев.

javadoc — это генератор документации в HTML-формате из комментариев исходного кода Java и определяет стандарт для документирования классов Java. Для создания доклетов и тэглетов, которые позволяют программисту анализировать структуру Java-приложения, **javadoc** также предоставляет API. В каждом случае комментарий должен находиться перед документируемым элементом.

Виды комментариев:

1. Комментарии javadoc для класса:

```
/**
 * Краткое описание класса.
 *
 * Подробное описание класса.
 *
 * @param <T> параметр обобщения для класса
 */
public class MyClass<T> {
    // код класса
}
```

2. Комментарии javadoc для метода:

```
/**
 * Краткое описание метода.
 *
 * Подробное описание метода.
 *
 * @param param1 описание первого параметра
 * @param param2 описание второго параметра
 * @return описание возвращаемого значения
 * @throws SomeException описание исключения
 */
public int myMethod(int param1, String param2) throws SomeException {
    // код метода
}
```

3. Комментарии javadoc для поля:

```
/**
```

```
 * Краткое описание поля.
 *
 * Подробное описание поля.
 */
private String myField;
```

4. Комментарии javadoc для параметра:

```
/**
 * Краткое описание метода.
 *
 * @param param1 описание первого параметра
 * @param param2 описание второго параметра
 */
public void myMethod(int param1, String param2) {
    // код метода
}
```

5. Комментарии javadoc для возвращаемого значения:

```
/**
 * Краткое описание метода.
 *
 * @return описание возвращаемого значения
 */
public int myMethod() {
    // код метода
}
```

50.Дескрипторы Javadoc.

Дескрипторы javadoc

Утилита javadoc распознает и обрабатывает в документирующих комментариях следующие дескрипторы

Дескриптор	Описание
@author	Обозначает автора программы
{@code}	Отображает данные шрифтом, предназначенным для вывода исходного кода, не выполняя преобразований в формат HTML-документа
@deprecated	Указывает на то, что элемент программы не рекомендован к применению
{@docRoot}	Указывает путь к корневому каталогу документации
@exception	Обозначает исключение, генерируемое методом
{@inheritDoc}	Наследует комментарии от ближайшего суперкласса
{@link}	Вставляет ссылку на другую тему
{@linkplain}	Вставляет ссылку на другую тему, но ссылка отображается тем же шрифтом, что и простой текст
{@literal}	Отображает данные, не выполняя преобразований в формат HTML-документа
@param	Документирует параметр метода
@return	Документирует значение, возвращаемое методом
@see	Указывает ссылку на другую тему
@serial	Документирует поле, упорядочиваемое по умолчанию
@serialData	Документирует данные, записываемые методом writeObject() или writeExternal()
@serialField	Документирует компонент ObjectStreamField
@since	Обозначает версию, в которой были внесены определенные изменения
@throws	То же, что и дескриптор @exception
{@value}	Отображает значение константы, которая должна быть определена как поле типа static
@version	Обозначает версию класса

Дескрипторы, начинающиеся с символа @, называются автономными и помечают строку комментариев. А дескрипторы, заключенные в фигурные скобки, называются встраиваемыми и могут быть использованы в других дескрипторах. В документирующих комментариях можно также использовать стандартные HTML-дескрипторы. Но некоторые HTML-дескрипторы, например дескрипторы заголовков, применять не следует,

поскольку они могут испортить внешний вид HTML-документа, составляемого утилитой javadoc.