

Java Fundamentals

1. JVM-JDK-JRE. Что это такое? Кто кого включает и как взаимодействуют.
2. Как скомпилировать и запустить класс, используя консоль?
3. Что такое classpath. Если в classpath есть две одинаковые библиотеки (или разные версии одной библиотеки), объект класса из какой библиотеки создастся?
4. Какие области памяти использует java для размещения простых типов, объектов, ссылок, констант, методов, пул строк и т.д.
5. Пакеты в java. Зачем применяются? Принцип именования пакетов.
6. Модификаторы доступа.
7. Почему метод main() объявлен как public static void?
8. Что такое package level access. Пример использования.
9. Может ли объект получить доступ к private-переменной класса? Если, да, то каким образом?
10. Классы-оболочки.
11. Autoboxing и unboxing. Принцип действия на примерах.
12. Какой будет результат кода?

```
int a = 1;  
Integer b = 2;  
int c = a+b; - результат?
```

А каков будет результат если Integer b = null?
13. Что такое var? Достоинства и недостатки.

Java Strings

14. В чем разница между созданием строки как new String() и литералом (при помощи двойных кавычек)?
15. Как реализуется класс String, какие поля там есть?
16. Как работает метод substring() класса String?
17. Понятие Юникод. UTF-8, описание кодировки. Отличие от UTF-16.
18. String, StringBuilder, StringBuffer. Отличия.

Java Classes

19. Базовый класс в java. Методы класса.
20. ООП abstraction. Принципы ООП.
21. Правила переопределения метода boolean equals(Object o).
22. Зачем переопределять методы hashCode и equals одновременно?
23. Написать метод equals для класса, содержащего одно поле типа String или StringBuilder.
24. Правила переопределения метода int hashCode(). Можно ли в качестве результата возвращать константу?
25. Правила переопределения метода clone().
26. Чем отличаются finally и finalize? Для чего используется ключевое слово final?
27. JavaBeans: основные требования к классам Bean-компонентов, соглашения об именах.
28. Как работает Garbage Collector. Какие самые распространенные алгоритмы? Можно ли самому указать сборщику мусора, какой объект удалить из памяти.
29. В каких областях памяти хранятся значения и объекты, массивы?
30. Какие идентификаторы по умолчанию имеют поля интерфейса?
31. Чем отличается абстрактный класс от интерфейса?
32. Когда применять интерфейс логичнее, а когда абстрактный класс?
33. Бывают ли интерфейсы без методов. Для чего?
34. Перегрузка и переопределение. Можно ли менять модификатор доступа метода, если да, то каким образом?

35. Перегрузка и переопределение. Можно ли менять возвращаемый тип метода, если да, то как? Можно ли менять тип передаваемых параметров?
36. Каким образом передаются переменные в методы, по значению или по ссылке?
37. Что такое конструктор по умолчанию?
38. Свойства конструктора. Способы его вызова.
39. Mutable и Immutable классы. Привести примеры из java core. Как создать класс, который будет immutable. Класс record.
40. static - что такое? Что будет, если значение атрибута изменить через объект класса? Всегда ли static поле содержит одинаковые значения для всех его объектов?
41. Внутренние классы, какие бывают и для каких целей используются. Области видимости данных при определенных ситуациях.
42. Анонимные классы. Практическое применение.
43. Generics. Что это такое и для чего применяются. Во что превращается во время компиляции и выполнения? Использование wildcards.

Ответ

wildcards: Пример 1 – `List<? extends Animal>`

Пример 2 – `List<? super Animal>`

44. Что такое enum? Область применения. Какое использование перечисления некорректно? Привести примеры.
45. Отличия в применении интерфейсов Comparator и Comparable?
46. Класс Optional. Как помогает бороться с null?
47. Принципы SOLID, Yagni, Kiss, Dry.

Ответ

Принципы SOLID:

a) Single Responsibility Principle (Принцип единственной ответственности).

У модуля должна быть только одна причина для изменения или класс должен отвечать только за что-то одно.

b) Open Closed Principle (Принцип открытости/закрытости).

Модуль должен быть открыт для расширения, но закрыт для изменения.

c) Liskov's Substitution Principle (Принцип подстановки Барбары Лисков).

Подклассы должны служить заменой своим суперклассам (функции, работающие с базовым типом должны работать с под типом).

d) Interface Segregation Principle (Принцип разделения интерфейса).

Сущности не должны зависеть от интерфейсов, которые они используют.

e) Dependency Inversion Principle (Принцип инверсии зависимостей).

Верхне-уровневые сущности не должны зависеть от нижне-уровневых реализаций.

Принципы Yagni — Always Keep It Simple, Stupid (будь проще):

Согласно ему, создавать какую-то функциональность следует только тогда, когда она действительно нужна.

Принципы Kiss — Always Keep It Simple, Stupid (будь проще):

a) Ваши методы должны быть небольшими (40-50 строк).

b) Каждый метод решает одну проблему.

c) При модификации кода в будущем не должно возникнуть трудностей.

d) Система работает лучше всего, если она не усложняется без надобности.

e) Не устанавливайте целую библиотеку ради одной функции из неё.

f) Не делай того, что не просят.

g) Писать код необходимо надежно и «дубово».

Принципы Dry — Don't Repeat Yourself (Не повторяйся):

a) Избегайте копирования кода.

b) Выносите общую логику.

c) Прежде чем добавлять функционал, проверьте в проекте, может, он уже создан.

d) Константы.

48. Если Interf – это интерфейс, а Clazz – это класс, который реализует интерфейс и в нем объявлен метод (которого нет в Interf), например, method(). Корректно ли выражение:

```
Interf a = new Clazz();  
a.method();
```

То же самое, если Interf – не интерфейс, а абстрактный класс.

Java IO

49. Что такое сериализация, для чего нужна, когда применяется? Ключевое слово transient, для чего нужно? Сериализация static-полей.

Ответ

Сериализуемый класс	
<pre>public class User implements Serializable { private static final long serialVersionUID = -3733782742070723489L; private int id; transient private String name; // не должны быть в методе hashCode() private static int staticField; public User(int id, String name) { this.id = id; this.name = name; } public int getId() { return id; } public void setId(int id) { this.id = id; } public String getName() { return name; } public void setName(String name) { this.name = name; } }</pre>	
Сериализация объекта	Десериализация объекта
<pre>public static void main(String[] args) throws IOException { User user = new User(1, "Alex"); FileOutputStream fos = new FileOutputStream("pathName.bin"); ObjectOutputStream oos = new ObjectOutputStream(fos); oos.writeObject(user); oos.close(); }</pre>	<pre>public static void main(String[] args) throws ClassNotFoundException, IOException { FileInputStream fis = new FileInputStream("pathName.bin"); ObjectInputStream ois = new ObjectInputStream(fis); User user = (User) ois.readObject(); ois.close(); }</pre>

Сериализация — это процесс сохранения состояния объекта в последовательность байт. Сериализация используется, когда нужно сохранить объект в байтах на диске, чтобы его можно было в будущем использовать (пример: сохранение старой игры). Порядок действий указан выше.

В Java за процессы сериализации отвечает интерфейс **java.io.Serializable**. **Serializable** – это интерфейс-маркер, который не содержит методов. Он показывает, что данный класс можно смело сериализовать.

Переменная `private static final long serialVersionUID` – это поле содержит уникальный идентификатор версии сериализованного класса.

Идентификатор версии есть у любого класса, который имплементирует интерфейс **Serializable**. Он вычисляется по содержимому класса — полям, порядку объявления, методам. И если мы поменяем в нашем классе тип поля и/или количество полей, идентификатор версии моментально изменится. `serialVersionUID` тоже записывается при сериализации класса.

Когда мы пытаемся провести десериализацию, то есть восстановить объект из набора байт, значение `serialVersionUID` сравнивается со значением `serialVersionUID` класса в нашей программе. Если значения не совпадают, будет выброшено исключение `java.io.InvalidClassException`.

Ключевое слово **transient** используется, когда нужно чтобы какое поле объекта не сериализовалось. Если ставится **transient** – это означает что значение данного поля не участвует при сериализации, то есть значение данного поля будет, если это объект – **null**, если примитивный тип – значение по умолчанию (`int = 0`, `double = 0.0`, `boolean = false` и так далее).

Статические поля класса не сериализуются.

Десериализация — это процесс восстановления объекта из этих байт. Десериализация используется, когда нужно сделать обратное действие сериализации. Порядок действий выше.

50. (сериализация) Возможно ли сохранить объект не в байт-код, а в xml-файл?

Ответ

Чтобы сохранить Java объект в XML файл, мы должны проставить необходимые JAXB аннотации в классе и методах класса, а затем создать объект `Marshaller` для сериализации объекта в XML.

JAXB (Java Architecture for XML Binding) — Java API для маршалинга объекта в XML и восстановления объекта из XML файла. Изначально JAXB был отдельным проектом, но своей простотой и удобством быстро завоевал популярность Java разработчиков. Именно поэтому в Java 6 JAXB стал частью JDK, а в Java 7 прокачался до версии 2.0.

Также есть специальный класс `JAXBContext`, который является точкой входа для JAXB и предоставляет методы для сохранения/восстановления объекта.

51. Что такое ClassLoader? Если изменить static переменную в классе, загруженном одним ClassLoader, что будет видно в том же классе, загруженном другим. Возможно ли синглтоны создавать несколько раз?

Ответ

Типы загрузчиков Java:

В Java существует три стандартных загрузчика, каждый из которых осуществляет загрузку класса из определенного места:

- **Bootstrap** – базовый загрузчик, также называется **Primordial ClassLoader**.

Загружает стандартные классы JDK из архива `rt.jar`

- **Extension ClassLoader** – загрузчик расширений.

Загружает классы расширений, которые по умолчанию находятся в каталоге `jre/lib/ext`, но могут быть заданы системным свойством `java.ext.dirs`

- **System ClassLoader** – системный загрузчик.

Загружает классы приложения, определенные в переменной среды окружения `CLASSPATH`.

В Java используется иерархия загрузчиков классов, где корневым, разумеется, является базовый. Каждый загрузчик, за исключением базового, является потомком абстрактного класса **`java.lang.ClassLoader`**.

Любой класс, который расширяет `ClassLoader`, может предоставить свой способ загрузки классов. Для этого необходимо переопределить соответствующие методы.

Традиционно `Singleton` создает свой собственный экземпляр, и он создает его только один раз. В этом случае невозможно создать второй экземпляр.

Java Exceptions

52. Опишите иерархию исключений.

Ответ

Все исключения наследуются от класса `Throwable`. Далее они делятся на класс `Error` и `Exception`. Все исключения, наследуемые от `Error` – это не проверяемые исключения. Примеры: `StackOverflowError`, `OutOfMemoryError`.

Класс `Exception` делится на **проверяемые** (наследники класса `Exception`) и **непроверяемые** (наследники класса `RuntimeException`).

- К **непроверяемым исключениям** относятся классы наследники класса `RuntimeException`. Примеры: `ClassCastException`, `IndexOutOfBoundsException`, `ArithmeticException`.

- К **проверяемым исключениям** относятся классы наследники класса `Exception`. Примеры: `FileNotFoundException`, `SQLException`, `IOException`, `ClassNotFoundException`.

53. Что такое checked и unchecked Exception? Их отличия.

Ответ

Проверяемые исключения необходимо обрабатывать или в текущем методе или отложить и обработать его в другом методе. Не проверяемые не обязательны для обработки.

54. Опишите работу блока try-catch-finally. Может ли работать данный блок без catch.

Ответ

Блок **try-catch-finally** – это блок для обработки исключений.

В блоке **try** работает основной код, если в этом коде возникает ошибка, пропускается остальной код в блоке **try** и работают блок(и) **catch**. Проверяются блок(и) **catch**, если находится исключение, которое выпало в блоке **try**, и работает код данного блока **catch**. Если блок **catch** с данным исключением не найден – выпадает соответствующее исключение.

Далее, вне зависимости сработал блок **catch** или нет, работает, если он есть, блок **finally**.

Блок **try-finally (без catch)** – работает.

55. Что такое Error. Перехват Error. Можно ли, есть ли смысл, в каких случаях возникает и что с ним делать.

Ответ

Error — это критическая ошибка во время исполнения программы, связанная с работой виртуальной машины Java.

В большинстве случаев Error не нужно обрабатывать, поскольку она свидетельствует о каких-то серьезных недоработках в коде. Наиболее известные ошибки: `StackOverflowError` — возникает, например, когда метод бесконечно вызывает сам себя, и `OutOfMemoryError` — возникает, когда недостаточно памяти для создания новых объектов.

В этих ситуациях чаще всего обрабатывать особо нечего — код просто неправильно написан и его нужно переделывать.

56. Чем отличается OutOfMemoryError от StackOverflowError. Как их избежать?

Ответ

- `OutOfMemoryError` происходит при переполнении кучи, а `StackOverflowError` – при переполнении стека;
- `StackOverflowError` происходит при бесконечном вызове метода, а `OutOfMemoryError` – при заиклиивании создания объектов;
- И того, и другое нужно исправить: найти заиклиивание и устранить его.

57. Оператор throw. Как работает? Какие свойства?

Ответ

Оператор `throw` используется, когда создаётся исключение.

Пример: `throw new Exception("1");`

58. С каким сообщением будет сгенерировано исключение и какое значение примет a?

```
try{
    a = 5;
    throw new Exception("1");
} catch (Exception e) {
    a = 10;
    throw new Exception("2");
} finally {
    a = 15;
    throw new Exception("3");
}
```

Ответ

сгенерировано исключение – "3"

значение a = 15

Java TestNG

59. Механизм assert. Когда возникает AssertionError? В чем его смысл?

Ответ

Assert помогает нам проверять условия теста и принимать решения, когда тест провален или выполнен. Тест считается выполненным только если завершается без вызова какого-либо исключения.

Исключение **AssertionError** возникает, когда тест провалился, то есть результат теста не совпал с ожидаемым.

60. JUnit. TestNG. Что это такое? Принципы написания.

Ответ

JUnit — это фреймворк для языка программирования Java, предназначенный для автоматического тестирования программ

TestNG – это фреймворк для тестирования, написанный на Java, он взял много чего с JUnit и NUnit, но он не только унаследовался от существующей функциональности JUnit, а также внедрил новые инновационные функции, которые делают его мощным, простым в использовании.

Java & XML

61. XML – парсеры.

Ответ

XML – парсеры:

- **SAX**

SAX-обработчик устроен так, что он просто считывает последовательно XML файлы и реагирует на разные события, после чего передает информацию специальному обработчику событий.

У него есть немало событий, однако самые частые и полезные следующие:

- startDocument — начало документа
- endDocument — конец документа
- startElement — открытие элемента
- endElement — закрытие элемента
- characters — текстовая информация внутри элементов.

- **STAX**

STAX – устроен так же как и SAX обработчик только при SAX обработчике мы переопределяем, а при STAX мы вызываем методы из соответствующих классов.

- **DOM**

DOM (Document Object Model) - DOM-обработчик устроен так, что он считывает сразу весь XML и сохраняет его, создавая иерархию в виде дерева, по которой мы можем спокойно двигаться и получать доступ к нужным нам элементам.

В DOM есть множество интерфейсов, которые созданы, чтобы описывать разные данные. Все эти интерфейсы наследуют один общий интерфейс – Node (узел). Потому, по сути, самый частый тип данных в DOM – это Node (узел), который может быть всем.

У каждого Node есть следующие полезные методы для извлечения информации:

- getNodeName – получить имя узла.
- getNodeValue – получить значение узла.
- getNodeType – получить тип узла.
- getParentNode – получить узел, внутри которого находится данный узел.
- getChildNodes – получить все производные узлы (узлы, которые внутри данного узла).
- getAttributes – получить все атрибуты узла.
- getOwnerDocument – получить документ этого узла.
- getFirstChild/getLastChild – получить первый/последний производный узел.
- getLocalName – полезно при обработке пространств имён, чтобы получить имя без префикса.
- getTextContent – возвращает весь текст внутри элемента и всех элементов внутри данного элемента, включая переносы строчек и пробелы.

62. Что лучше использовать в каких случаях, well-formed, valid.

Ответ

Отличие:

well-formed XML - тот, который пропускается парсером

valid XML - тот, который пропускается парсером И валидатором

63. В чем отличие dtd и xsd, какая из этих схем написана в формате xml?

Ответ

Файл XSD — это файл определения, определяющий элементы и атрибуты, которые могут быть частью документа XML. Это гарантирует правильную интерпретацию данных и обнаружение ошибок, что приводит к соответствующей проверке XML.

DTD означает определение типа документа, и это документ, который определяет структуру XML-документа. Он используется для точного описания атрибутов языка XML. Его можно разделить на два типа, а именно внутренний DTD и внешний DTD. Он может быть указан внутри документа или вне документа.

Многопоточность

64. Что такое процесс? Что такое поток? Состояния потока.

Ответ

Процесс – это совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

Поток – это единица реализации программного кода.

Каждый поток пребывает в одном из следующих состояний (state):

- Создан (New) – очередь к кадровику готовится, люди организуются.
- Запущен (Runnable) – наша очередь выстроилась к кадровику и обрабатывается.
- Заблокирован (Blocked) – последний в очереди юноша пытается выкрикнуть имя, но услышав, что девушка в соседней группе начала делать это раньше него, замолчал.
- Завершён (Terminated) — вся очередь оформилась у кадровика и в ней нет необходимости.
- Ожидает (Waiting) – одна очередь ждёт сигнала от другой.

65. Как создать поток? Какими способами можно создать поток, запустить его, прервать (завершить, убить)?

Ответ

Создать поток можно 2-мя способами:

1 способ – наследуясь от класса Thread и переопределяя метод run(). Запуск потока происходит путём создания объекта (в данном случае User.class) и вызова у него метода start().

2 способ – имплементируя интерфейс Runnable и переопределяя метод run(). Запуск потока происходит путём создания объекта класса Thread и передать ему в конструкторе объект с переопределённым интерфейсом Runnable и вызова у объекта класса Thread.class метод start().

1 способ	2 способ
<pre>public class User extends Thread { private long id; private String userName; public User(int id, String userName) { this.id = id; this.userName = userName; } public long getId() { return id; } public void setId(long id) { this.id = id; } public String getUserName() { return userName; } }</pre>	<pre>public class User implements Runnable { private long id; private String userName; public User(int id, String userName) { this.id = id; this.userName = userName; } public long getId() { return id; } public void setId(long id) { this.id = id; } public String getUserName() { return userName; } }</pre>

<pre> public void setUsername(String userName) { this.userName = userName; } @Override public void run() { // doing method ... } } </pre>	<pre> public void setUsername(String userName) { this.userName = userName; } @Override public void run() { // doing method ... } } </pre>
<pre> public static void main(String[] args) { new User(1, "Alex").start(); } </pre>	<pre> public static void main(String[] args) { User user = new User(1, "Alex"); new Thread(user).start(); } </pre>

Чтобы остановить поток нужно вызвать метод `Thread.interrupt()`.

66. ???Как выполнить набор команд в отдельном потоке?

Ответ ???

???

67. Как работают методы `wait` и `notify/notifyAll`?

Ответ

Метод `Object.wait()` – освобождает монитор (объект на котором синхронизируются потоки) от текущего потока. Метод `Object.notify()` – оживляет 1 случайный поток на котором был вызван метод `Object.wait()`. Метод `Object.notifyAll()` – оживляет все потоки на которых был вызван метод `Object.wait()`.

68. Чем отличается работа метода `wait` с параметром и без параметра?

Ответ

Метод `Object.wait` с параметром работает конкретное (указанное) время. А без параметров работает пока не будет вызван метод `Object.notify` или `Object.notifyAll`.

69. Как работает метод `yield()`? Чем отличаются методы `Thread.sleep()` и `Thread.yield()`?

Ответ

Вызов метода **`Thread.yield()`** позволяет досрочно завершить квант времени текущей нити или, другими словами, переключает процессор на следующую нить.

Вызов **`yield`** приводит к тому, что «наша нить досрочно завершает ход», и что следующая за **`yield`** команда начнется с полного кванта времени.

`sleep(timeout)` – останавливает текущую нить (в которой `sleep` был вызван) на `timeout` миллисекунд. Нить при этом переходит в состояние `TIMED_WAITING`.

`yield()` – текущая нить «пропускает свой ход». Нить из состояния `running` переходит в состояние **`ready`**, а Java-машина приступает к выполнению следующей нити. Состояния `running & ready` – это подсостояния состояния **`RUNNABLE`**.

70. Чем отличаются методы `Thread.sleep` и `wait`?

Ответ

`sleep(timeout)` – останавливает текущую нить (в которой `sleep` был вызван) на `timeout` миллисекунд. Нить при этом переходит в состояние `TIMED_WAITING`.

`wait(timeout)` – это одна из версий метода **`wait()`** – версия с таймаутом. Метод **`wait`** можно вызвать только внутри блока **`synchronized`** у объекта-мьютекса, который был «залочен (заблокирован)» текущей нитью, в противном случае метод выкинет исключение **`IllegalMonitorStateException`**.

В результате вызова этого метода, блокировка с объекта-мьютекса снимается, и он становится доступен для захвата и блокировки другой нитью. При этом нить переходит в состояние `WAITING` для метода `wait()` без параметров, но в состояние `TIMED_WAITING` для метода `wait(timeout)`.

71. Как работает метод `join()`?

Ответ

При вызове метода `join()` или `join(timeout)` текущая нить как бы «присоединяется» к нити, у объекта которой был вызван данный метод. Текущая нить засыпает и ждет окончания нити, к которой она присоединилась (чей метод `join()` был вызван).

При этом текущая нить переходит в состояние `WAITING` для метода `join` и в состояние `TIMED_WAITING` для метода `join(timeout)`.

72. Как правильно завершить работу потока? (Иногда говорят, убить поток).

Ответ

Чтобы остановить поток нужно вызвать метод `Thread.interrupt()` и обрабатывать исключение `InterruptedException`.

73. Что такое синхронизация? Зачем она нужна? Для чего нужно ключевое слово `synchronized`? Какие методы синхронизации вы знаете? Какими средствами достигается?

Ответ

Синхронизация — это процесс, который позволяет выполнять все параллельные потоки в программе синхронно.

Когда метод объявлен как синхронизированный — нить держит монитор для объекта, метод которого выполняется. Если другой поток выполняет синхронизированный метод, ваш поток заблокируется до тех пор, пока другой поток не отпустит монитор.

Синхронизация достигается в Java использованием зарезервированного слова `synchronized`. Его можно использовать в классах определяя методы или блоки.

74. Отличия работы `synchronized` от `Lock`?

Ответ

Lock — интерфейс из **lock framework**, предоставляющий гибкий подход по ограничению доступа к ресурсам/блокам по сравнению с `synchronized`. При использовании нескольких локов порядок их освобождения может быть произвольный, плюс его также можно настроить. Еще имеется возможность обработать ситуацию, когда лок уже захвачен.

- **Синхронизированный блок полностью содержится в методе.** У нас могут быть операции `lock()` и `unlock()` API блокировки в отдельных методах.

- Синхронизированный блок не поддерживает справедливость. Любой поток может получить блокировку после освобождения, и никакие предпочтения не могут быть указаны. **Мы можем добиться справедливости в API-интерфейсах блокировки, указав свойство справедливости.** Это гарантирует, что самый длинный ожидающий поток получит доступ к блокировке.

- Поток блокируется, если он не может получить доступ к синхронизированному блоку. API блокировки предоставляет метод `tryLock()`. Поток получает блокировку только в том случае, если он доступен и не удерживается каким-либо другим потоком. Это уменьшает время блокировки потока, ожидающего блокировки.

- Поток, который находится в состоянии «ожидания» получения доступа к синхронизированному блоку, не может быть прерван. API блокировки предоставляет метод `lockInterruptibly()`, который можно использовать для прерывания потока, когда он ожидает блокировки.

75. Есть ли у `Lock` механизм, аналогичный механизму `wait/notify` у `synchronized`?

Ответ

Condition — применение условий в блокировках позволяет добиться контроля над управлением доступа к потокам. Условие блокировки представляет собой объект интерфейса **Condition** из пакета `java.util.concurrent.locks`. Применение объектов **Condition** во многом аналогично использованию методов `wait/notify/notifyAll` класса `Object`.

Получить **Condition** можно следующим образом:

```
ReentrantLock lock = new ReentrantLock(); //получаем lock
```

```
Condition condition = lock.newCondition(); // получаем из lock condition
```

76. Что такое deadlock? Нарисовать схему, как это происходит.

Ответ

Deadlock или взаимная блокировка — это ошибка, которая происходит, когда нити имеют циклическую зависимость от пары синхронизированных объектов.

Пример:

```
public class Deadlock {
    static class Friend {
        private final String name;

        public Friend(String name) {
            this.name = name;
        }
    }
}
```

```

    }

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

    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s" + " has bowed to me!\n", this.name, bower.getName());
        bower.bowBack(this);
    }

    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s" + " has bowed back to me!\n", this.name, bower.getName());
    }
}

public static void main(String[] args) {
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
        @Override
        public void run() {
            // System.out.println("Thread 1");
            alphonse.bow(gaston);
            // System.out.println("Th: gaston bowed to alphonse");
        }
    }).start();

    new Thread(new Runnable() {
        @Override
        public void run() {
            // System.out.println("Thread 2");
            gaston.bow(alphonse);
            // System.out.println("2.gaston waiting alph bowed");
        }
    }).start();
}
}

```

Создали два объекта класса Friend: alphonse и gaston. У каждого из них есть свой лок. Таким образом, этих локов два: альфонсов и гастронов. При входе в синхронизированный метод объекта, его лок запирается, а когда из метода выходят, он освобождается (или отпирается). Теперь о нитях. Назовем первую нить Alphonse (с большой буквы, чтобы отличить от объекта alphonse). Вот что она делает (обозначим её буквой А, сокращённо от Alphonse):

А: alphonse.bow(gaston) — получает лок alphonse;
 А: gaston.bowBack(alphonse) — получает лок gaston;
 А: возвращается из обоих методов, тем самым освобождая лок.

А вот чем в это время занята нить Gaston:

Г: gaston.bow(alphonse) — получает лок gaston;
 Г: alphonse.bowBack(gaston) — получает лок alphonse;
 Г: возвращается из обоих методов, тем самым освобождая лок.

Теперь сведем эти данные вместе и получим ответ. Нити могут переплетаться (то есть, их события совершатся) в разных порядках. Дедлок получится, например, если порядок будет таким:

А: alphonse.bow(gaston) — получает лок alphonse
 Г: gaston.bow(alphonse) — получает лок gaston
 Г: пытается вызвать alphonse.bowBack(gaston), но блокируется, ожидая лока alphonse
 А: пытается вызвать gaston.bowBack(alphonse), но блокируется, ожидая лока gaston

77. Semaphore, CyclicBarrier, CountDownLatch. Чем похожи на Lock и чем от него отличаются?

Ответ

- **Semaphore** (java.util.concurrent.Semaphore)

Самое простое средство контроля за тем, сколько потоков могут одновременно работать — семафор. Как на железной дороге. Горит зелёный — можно. Горит красный — ждём. Что мы ждём от семафора? Разрешения. Разрешение на английском — permit. Чтобы получить разрешение — его нужно получить, что на английском будет acquire. А когда разрешение больше не нужно мы его должны отдать, то есть освободить его или избавиться от него, что на английском будет release. Посмотрим, как это работает.

Пример:

```
public static void main(String[] args) throws InterruptedException {
    Semaphore semaphore = new Semaphore(0);
    Runnable task = () -> {
        try {
            semaphore.acquire();
            System.out.println("Finished");
            semaphore.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    };
    new Thread(task).start();
    Thread.sleep(5000);
    semaphore.release(1);
}
```

- **CountDownLatch** (java.util.concurrent.CountDownLatch)

Это похоже на бега или гонки, когда все собираются у стартовой линии и когда все готовы — дают разрешение, и все одновременно стартуют.

Пример:

```
public static void main(String[] args) {
    CountDownLatch countDownLatch = new CountDownLatch(3);
    Runnable task = () -> {
        try {
            countDownLatch.countDown();
            System.out.println("Countdown: " + countDownLatch.getCount());
            countDownLatch.await();
            System.out.println("Finished");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    };
    for (int i = 0; i < 3; i++) {
        new Thread(task).start();
    }
}
```

await на английском — ожидать. То есть мы сначала говорим countDown. Как говорит гугл переводчик, count down — "an act of counting numerals in reverse order to zero", то есть выполнить действие по обратному отсчёту, цель которого — досчитать до нуля. А дальше говорим await — то есть ожидать, пока значение счётчика не станет ноль.

Такой счётчик — одноразовый, то есть если нужен многократный счёт — надо использовать другой вариант, который называется CyclicBarrier.

- **CyclicBarrier** (java.util.concurrent.CyclicBarrier)

CyclicBarrier — это циклический барьер.

Пример

```
public static void main1(String[] args) throws InterruptedException {
    Runnable action = () -> System.out.println("На старт!");
    CyclicBarrier barrier = new CyclicBarrier(3, action);
    Runnable task = () -> {
        try {
            barrier.await();
            System.out.println("Finished");
        } catch (BrokenBarrierException | InterruptedException e) {
            e.printStackTrace();
        }
    };
    System.out.println("Limit: " + barrier.getParties());
}
```

```

    for (int i = 0; i < 3; i++) {
        new Thread(task).start();
    }
}

```

Поток выполняет await, то есть ожидает. При этом уменьшается значение барьера. Барьер считается сломанным (barrier.isBroken()), когда отсчёт дошёл до нуля.

Чтобы сбросить барьер, нужно вызвать barrier.reset(), чего не хватало в CountdownLatch.

78. Написать deadlock, придумать примеры с использованием synchronized, AtomicInteger.

Ответ

См вопрос 76.

Класс **AtomicInteger** предоставляет операции с значением int, которые могут быть прочитаны и записаны атомарно, в дополнение содержит расширенные атомарные операции.

У него есть методы get и set, которые работают, как чтение и запись по переменным.

79. По каким объектам синхронизируются статические и нестатические методы?

Ответ

При синхронизации статических методов – происходит синхронизация на уровне класса, а при синхронизации не статических методов – происходит синхронизация на уровне объекта.

Примеры:

<i>Синхронизация статических методов</i>	
<pre> class MyThread extends Thread { Resource resource; @Override public void run() { Resource.change(); } } class Resource { private static int i; public static int getI() { return i; } public static void setI(int i) { Resource.i = i; } public synchronized static void change() { int i = Resource.i; i++; Resource.i = i; } } </pre>	<pre> class MyThread extends Thread { Resource resource; @Override public void run() { Resource.change(); } } class Resource { private static int i; public static int getI() { return i; } public static void setI(int i) { Resource.i = i; } public static void change() { synchronized (Resource.class) { int i = Resource.i; i++; Resource.i = i; } } } </pre>
<i>Синхронизация не статических методов</i>	
<pre> class MyThread extends Thread { Resource resource; @Override public void run() { this.resource.change(); } } class Resource { private int i; public int getI() { return i; } public void setI(int i) { </pre>	<pre> class MyThread extends Thread { Resource resource; @Override public void run() { this.resource.change(); } } class Resource { private int i; public int getI() { return i; } public void setI(int i) { </pre>

<pre> this.i = i; } public synchronized void change() { int i = this.i; i++; this.i = i; } } </pre>	<pre> this.i = i; } public void change() { synchronized (this) { int i = this.i; i++; this.i = i; } } } </pre>
--	---

80. Для чего применяется volatile? Пакет java.util.concurrent.atomic.

Ответ

Volatile – это модификатор который показывает, что переменная, рядом с которой стоит volatile, находится не в памяти конкретного потока, а в общей памяти потоков.

Переменная без модификатора volatile, хэшируется и находится в памяти отдельного потока, а другие потоки не имеют доступа к ней. А переменная с модификатором volatile не хэшируется и находится в общей памяти потоков, чтобы все потоки имели доступ к переменной.

Пакет **java.util.concurrent.atomic** содержит девять классов для выполнения атомарных операций. Операция называется атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

Пример:

С точки зрения программиста операции инкремента (i++, ++i) и декремента (i--, --i) выглядят наглядно и компактно. Но, с точки зрения JVM (виртуальной машины Java) данные операции не являются атомарными, поскольку требуют выполнения нескольких действительно атомарных операций: чтение текущего значения, выполнение инкремента/декремента и запись полученного результата.

Блокировка (synchronized) подразумевает **пессимистический** подход, разрешая только одному потоку выполнять определенный код, связанный с изменением значения некоторой «общей» переменной. Таким образом, никакой другой поток не имеет доступа к определенным переменным. Но можно использовать и **оптимистический** подход. В этом случае блокировки не происходит, и если поток обнаруживает, что значение переменной изменилось другим потоком, то он повторяет операцию снова, но уже с новым значением переменной. Так работают атомарные классы.

81. Есть массив из N-ти элементов. Создать N потоков, которые принимают по числу из массива, обрабатывают и возвращают обратно. Собрать все обработанные числа обратно в массив.

Java. Collection

82. Основные интерфейсы коллекций и их иерархия (List, Set, Queue). Какие бывают коллекции? В чём особенности разных видов коллекций? Когда какие стоит применять?

Ответ

Collection – это интерфейс от которого наследуются следующие интерфейсы: List, Set, Queue.

List<T> – это интерфейс от которого наследуются: класс ArrayList, класс Vector (потокобезопасный), класс LinkedList. Интерфейс List работает на основе индексов.

Set<T> – это интерфейс от которого наследуются: класс HashSet, класс LinkedHashSet, интерфейс SortedSet. SortedSet – это интерфейс от которого наследуется класс TreeSet.

Интерфейс Set используется, когда необходим список с уникальными элементами. Коллекция Set не позволяет хранить одинаковые элементы.

Queue<T> – это интерфейс от которого наследуется интерфейс Deque. Deque – это интерфейс от которого наследуется класс LinkedList.

Интерфейс Queue – это очередь. Элементы добавляются в конец очереди, а выбираются из ее начала.

83. Что такое интерфейс Map? Представляет ли он коллекцию?

Ответ

Map<K, V> – это интерфейс от которого наследуются: класс HashMap, класс LinkedHashMap, интерфейс SortedMap, класс Hashtable. SortedMap – это интерфейс от которого наследуется класс TreeMap. Представляет из себя ключ-значение.

Map не является реализацией интерфейса Collection, тем не менее, является частью фреймворка Collections.

84. Сравнить ArrayList и LinkedList.

Ответ

- **Базовая структура данных**

Оба ArrayList а также LinkedList две разные реализации List интерфейса. ArrayList представляет собой реализацию массива с изменяемым размером, тогда как LinkedList представляет собой реализацию двусвязного списка List интерфейса.

- **Реализация**

ArrayList реализует только список, а LinkedList реализует и список, и очередь. LinkedList также часто используется в качестве очередей.

- **Доступ**

ArrayList быстрее хранят и извлекают данные. С другой стороны, LinkedList поддерживает более быструю обработку данных.

- **Емкость**

Емкость ArrayList по крайней мере равна размеру списка, и она автоматически увеличивается по мере добавления к нему новых элементов. Его емкость по умолчанию составляет всего 10 элементов. Поскольку изменение размера снижает производительность, всегда лучше указать начальную емкость ArrayList во время самой инициализации.

С другой стороны, емкость LinkedList точно равна размеру списка, и мы не можем указать емкость во время инициализации списка.

- **Накладные расходы памяти**

LinkedList требует дополнительных затрат памяти, поскольку каждый элемент представляет собой объект узла, в котором хранятся указатели на следующий и предыдущий элементы. Но поскольку память, необходимая для каждого узла, может быть не непрерывной, LinkedList не приведет к серьезным проблемам с производительностью.

Однако ArrayList нуждается в непрерывном блоке памяти в куче для выделения динамического массива. Это может быть эффективным по пространству, но иногда приводит к проблемам с производительностью, когда сборщик мусора в конечном итоге выполняет некоторую работу по освобождению необходимого непрерывного блока памяти в куче.

- **Кэширование**

Проходить через элементы ArrayList всегда быстрее, чем LinkedList. Это из-за последовательной локализации или место ссылки где аппаратное обеспечение будет кэшировать смежные блоки памяти для более быстрого доступа к чтению.

85. Сравнить HashMap и Hashtable.

Ответ

Hashtable — это структура данных для хранения пар ключей и их значений, основанная на хешировании и реализации интерфейса Map.

HashMap также является структурой данных для хранения ключей и значений, основанной на хешировании и реализации интерфейса Map. HashMap позволяет быстро получить значение по ключу.

- HashMap — это **несинхронизированная** неупорядоченная карта пар ключ-значение (key-value). Она **допускает** пустые значения и использует хэш-код в качестве проверки на равенство, в то время как Hashtable представляет собой **синхронизированную** упорядоченную карту пар ключ-значение. Она **не допускает** пустых значений и использует метод equals() для проверки на равенство.

- HashMap по умолчанию имеет емкость 16, а начальная емкость Hashtable по умолчанию — 11.
- Значения объекта HashMap перебираются с помощью итератора, а Hashtable — это единственный класс, кроме вектора, который использует перечислитель (enumerator) для перебора значений объекта Hashtable.

86. Как устроены HashSet, TreeMap, TreeSet?

Ответ

- **HashSet**

Класс HashSet реализует интерфейс Set, основан на хэш-таблице, а также поддерживается с помощью экземпляра HashMap. В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время.

Хэш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хэш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

- **TreeMap**

TreeMap implements интерфейс NavigableMap, который наследуется от SortedMap, а он, в свою очередь от интерфейса Map.

Под капотом TreeMap использует структуру данных, которая называется красно-чёрное дерево. Именно хранение данных в этой структуре и обеспечивает порядок хранения данных.

- **TreeSet**

Обобщенный класс TreeSet<E> представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet, а, следовательно, и интерфейс SortedSet.

87. Принцип работы и реализации HashMap. Изменения HashMap в java8.

Ответ

Класс **HashMap** наследуется от класса AbstractMap и реализует следующие интерфейсы: Map, Cloneable, Serializable.

HashMap имеет следующие поля:

- `transient Node<K, V>[] table` — сама хеш-таблица, реализованная на основе массива, для хранения пар «ключ-значение» в виде узлов. Здесь хранятся наши Node;
- `transient int size` — количество пар «ключ-значение»;
- `int threshold` — предельное количество элементов, при достижении которого размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле (`capacity * loadFactor`);
- `final float loadFactor` — этот параметр отвечает за то, при какой степени загруженности текущей хеш-таблицы необходимо создавать новую хеш-таблицу, т.е. как только хеш-таблица заполнилась на 75%, будет создана новая хеш-таблица с перемещением в неё текущих элементов (затратная операция, так как требуется перехеширование всех элементов);
- `transient Set<Map.Entry<K, V>> entrySet` — содержит кешированный `entrySet()`, с помощью которого мы можем перебирать HashMap.

Node — это вложенный класс внутри HashMap, который имеет следующие поля:

- `final int hash` — хеш текущего элемента, который мы получаем в результате хеширования ключа;
- `final K key` — ключ текущего элемента. Именно сюда записывается то, что вы указываете первым объектом в методе `put()`;
- `V value` — значение текущего элемента. А сюда записывается то, что вы указываете вторым объектом в методе `put()`;
- `Node<K, V> next` — ссылка на следующий узел в пределах одной корзины. Список же связный, поэтому ему нужна ссылка не следующий узел, если такой имеется.

Добавление объектов:

Добавление пары "ключ-значение" осуществляется с помощью метода `put()`.

- а) Вычисляется хеш-значение ключа введенного объекта.
- б) Вычисляем индекс бакета (ячейки массива), в который будет добавлен наш элемент.
- в) Создается объект Node.
- г) Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Если в бакете пусто, тогда просто размещаем в нем элемент. Иначе хэш и ключ нового элемента поочередно сравниваются с хешами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается. Если совпадений не найдено, элемент добавляется в конец списка.

Ситуация, когда разные ключи попадают в один и тот же бакет (даже с разными хешами), называется коллизией или столкновением.

Изменения в Java 8

В случае возникновения коллизий объект node сохраняется в структуре данных "связанный список" и метод `equals()` используется для сравнения ключей. Это сравнения для поиска верного ключа в связанном списке -линейная операция и в худшем случае сложность равна $O(n)$.

Для исправления этой проблемы в Java 8 после достижения определенного порога вместо связанных списков используются сбалансированные деревья. Это означает, что HashMap в начале сохраняет объекты в связанном списке, но после того, как количество элементов в хэше достигает определенного порога происходит переход к сбалансированным деревьям. Что улучшает производительность в худшем случае с $O(n)$ до $O(\log n)$.

88. Чем отличается ArrayList от Vector?

Ответ

Основные различия между ArrayList и Vector:

- **Синхронизация**

Основное различие между ArrayList и Vector, это то что Vector реализация синхронизируется, пока ArrayList реализация не синхронизирована. Это означает, что только один поток может работать с Vector методом за раз, в то время как несколько потоков могут работать над ArrayList одновременно. Чтобы сделать ArrayList потокобезопасный, его можно синхронизировать извне с помощью Collections.synchronizedList() метод.

Vector может быть синхронизирован, но не совсем потокобезопасен. Это потому что Vector синхронизируется по каждой операции, а не по всей Vector сам экземпляра.

- **Производительность**

Vectors очень медленные, поскольку они синхронизированы, и один поток может получить блокировку операции, заставляя другие потоки ждать, пока эта блокировка не будет снята. ArrayList, с другой стороны, намного быстрее, чем Vector поскольку он не синхронизирован, и с ним одновременно могут работать несколько потоков.

- **Управление хранилищем**

Оба ArrayList а также Vector может динамически увеличиваться и уменьшаться для размещения новых элементов, если это необходимо. Вместо добавочного перераспределения хранилище увеличивается порциями. Обычно, когда добавляются новые элементы и емкость заполнена, ArrayList увеличивает свой размер наполовину от текущего размера, а Vector удваивает свой размер. Обо всем этом автоматически заботится виртуальная машина Java (JVM).

- **Безотказность**

Для обхода списка используется ArrayList использует итератор, а Vector использует как перечисление, так и итератор. Перечисление, возвращаемое векторным методом, не является отказоустойчивым. Напротив, итераторы, возвращаемые iterator() а также listIterator() методы обоих Vector а также ArrayList отказоустойчивы и бросает ConcurrentModificationException если коллекция структурно изменена после создания итератора, кроме как через собственный итератор remove() или же add() методы.

89. Особенности интерфейса Set.

Ответ

Интерфейс Set расширяет интерфейс Collection и представляет набор уникальных элементов. Set не добавляет новых методов, только вносит изменения в унаследованные. В частности, метод add() добавляет элемент в коллекцию и возвращает true, если в коллекции еще нет такого элемента.

90. Как добавляются объекты в HashSet?

Ответ

Все классы, реализующие интерфейс Set, внутренне поддерживаются реализациями Map. HashSet хранит элементы с помощью HashMap, то есть код в качестве поля объекта находится объект класса HashMap.

```
public class HashSet<E> {  
    private transient HashMap<E, Object> map;  
  
    public HashSet() {  
        map = new HashMap<>();  
    }  
}
```

При добавлении объекта в HashSet, данный объект добавляется в HashMap, но поскольку в HashMap добавляется пара “ключ-значение”, то в качестве ключа передаётся передаваемый объект, а в качестве значения добавляется объект, занесённый в классе HashSet в константу.

```
private static final Object PRESENT = new Object();  
  
public boolean add(E e) {  
    return map.put(e, PRESENT) == null;  
}
```

91. Какими способами можно отсортировать коллекцию? (привести три способа)

Ответ

- **Использование Collections.sort() метод**

Collections служебный класс предоставляет статический `sort()` метод сортировки указанного списка в порядке возрастания в соответствии с естественным порядком его элементов.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Main{
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10, 4, 2, 6, 5, 8);
        Collections.sort(list); // результат [2, 4, 5, 6, 8, 10]
    }
}
```

Этот метод будет производить стабильную сортировку. Это будет работать, только если все элементы списка реализуют `Comparable` интерфейса и взаимно сравнимы, т. е. для любой пары элементов (a, b) в списке, `a.compareTo(b)` не бросает `ClassCastException`.

- **Использование `List.sort()` метод**

Каждый `List` реализация обеспечивает статическое `sort()` метод, который сортирует список в соответствии с порядком, заданным указанным `Comparator`. Чтобы этот метод работал, все элементы списка должны быть взаимно сравнимы с использованием указанного компаратора.

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10, 4, 2, 6, 5, 8);
        list.sort(new Comparator<Integer>() {

            @Override
            public int compare(Integer o1, Integer o2) {
                return o1 - o2;
            }

        }); // результат [2, 4, 5, 6, 8, 10]
    }
}
```

Если указанный компаратор равен нулю, то все элементы в этом списке должны реализовывать `Comparable` интерфейс, и будет использоваться естественный порядок элементов.

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10, 4, 2, 6, 5, 8);
        list.sort(null); // результат [2, 4, 5, 6, 8, 10]
    }
}
```

- **Использование `Java 8`**

Сортировка `List` стало еще проще с введением `Stream` в `Java 8` и выше. Идея состоит в том, чтобы получить поток, состоящий из элементов списка, отсортировать его в естественном порядке с помощью функции `Stream.sorted()` метод и, наконец, соберите все отсортированные элементы в список, используя `Stream.collect()` с `Collectors.toList()`. Чтобы сортировка работала, все элементы списка должны быть взаимно сравнимы.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(10, 4, 2, 6, 5, 8);
        list = list.stream()
            .sorted()
            .collect(Collectors.toList()); //результат [2, 4, 5, 6, 8, 10]
    }
}
```

92. Как правильно удалить элемент из коллекции при итерации в цикле?

Ответ

Для удаления элемента из List определён метод `remove()`. Им можно пользоваться если мы удаляем элемент передавая индекс или объект. Но если мы удалим объект в цикле (`for` или `for-each`) программа или отработает не корректно или выдаст исключение `java.util.ConcurrentModificationException`.

Пример:

```
public class Main {
    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        Cat thomas = new Cat("Томас");
        Cat behemoth = new Cat("Бегемот");
        Cat philipp = new Cat("Филипп Маркович");
        Cat pushok = new Cat("Пушок");

        cats.add(thomas);
        cats.add(behemoth);
        cats.add(philipp);
        cats.add(pushok);

        for (Cat cat : cats) {
            if (cat.getName().equals("Бегемот")) {
                cats.remove(cat);
            }
        }
    }
}

class Cat {
    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Результат:

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
at ...Main.main(Main.java:30)
```

Данное исключение выпало потому что не соблюдено правило **“Нельзя проводить одновременно итерацию (перебор) коллекции и изменение ее элементов.”**.

В Java для удаления элементов во время перебора нужно использовать специальный объект — итератор (класс `Iterator`).

Класс `Iterator` отвечает за безопасный проход по списку элементов.

Он имеет всего 3 метода:

- `hasNext()` — возвращает `true` или `false` в зависимости от того, есть ли в списке следующий элемент, или мы уже дошли до последнего.
- `next()` — возвращает следующий элемент списка
- `remove()` — удаляет элемент из списка

В классе `ArrayList` уже реализован специальный метод для создания итератора — `iterator()`.

Пример удаления объекта из List с помощью объекта итератор (класс `Iterator`):

```
public class Main {
    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        Cat thomas = new Cat("Томас");
        Cat behemoth = new Cat("Бегемот");
        Cat philipp = new Cat("Филипп Маркович");
        Cat pushok = new Cat("Пушок");
```

```

        cats.add(thomas);
        cats.add(behemoth);
        cats.add(philipp);
        cats.add(pushok);

        Iterator<Cat> catIterator = cats.iterator(); //создаем итератор
        while (catIterator.hasNext()) { //до тех пор, пока в списке есть элементы
            Cat nextCat = catIterator.next(); //получаем следующий элемент
            if (nextCat.getName().equals("Филипп Маркович")) {
                catIterator.remove(); //удаляем кота с нужным именем
            }
        }
    }
}

class Cat {
    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

93. Как правильно удалить элемент из ArrayList (или другой коллекции) при поиске этого элемента в цикле?

Ответ

Вопрос 92.

94. Коллекции из пакета concurrent. Их особенности.

Ответ

Concurrency – это библиотека классов в Java, в которой собрали специальные классы, оптимизированные для работы из нескольких нитей. Эти классы собраны в пакете `java.util.concurrent`.

Пакет `java.util.concurrent` предоставляет следующие инструменты для написания многопоточного кода:

- **Atomic** (`java.util.concurrent.atomic`)

В дочернем пакете `java.util.concurrent.atomic` находится набор классов для атомарной работы с примитивными типами. Контракт данных классов гарантирует выполнение операции `compare-and-set` за «1 единицу процессорного времени». При установке нового значения этой переменной вы также передаете ее старое значение (подход оптимистичной блокировки). Если с момента вызова метода значение переменной отличается от ожидаемого — результатом выполнения будет `false`. Например, это такие классы как: `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicLongArray` и другие.

- **Locks** (`java.util.concurrent.locks`)

➤ **ReentrantLock** (`java.util.concurrent.locks.ReentrantLock`)

В отличие от `synchronized` блокировок, `ReentrantLock` позволяет более гибко выбирать моменты снятия и получения блокировки т.к. использует обычные Java вызовы. Также `ReentrantLock` позволяет получить информацию о текущем состоянии блокировки, разрешает «ожидать» блокировку в течение определенного времени. Поддерживает правильное рекурсивное получение и освобождение блокировки для одного потока. Если вам необходимы честные блокировки (соблюдающие очередность при захвате монитора) — `ReentrantLock` также снабжен этим механизмом.

➤ **ReentrantReadWriteLock** (`java.util.concurrent.locks.ReentrantReadWriteLock`)

Дополняет свойства `ReentrantLock` возможностью захватывать множество блокировок на чтение и блокировку на запись. Блокировка на запись может быть «опущена» до блокировки на чтение, если это необходимо.

➤ **StampedLock** (`java.util.concurrent.locks.StampedLock`)

Реализовывает оптимистичные и пессимистичные блокировки на чтение-запись с возможностью их дальнейшего увеличения или уменьшения. Оптимистичная блокировка реализуется через «штамп» лока (*javadoc*).

- **Collections** (*java.util.concurrent.**)

- **ArrayBlockingQueue** (*java.util.concurrent.ArrayBlockingQueue*)

- Честная очередь для передачи сообщения из одного потока в другой. Поддерживает блокирующие (*put()* и *take()*) и неблокирующие (*offer()* и *poll()*) методы. Запрещает *null* значения. Емкость очереди должна быть указана при создании.

- **ConcurrentHashMap** (*java.util.concurrent.ConcurrentHashMap*)

- Ключ-значение структура, основанная на *hash* функции. Отсутствуют блокировки на чтение. При записи блокируется только часть карты (сегмент). Кол-во сегментов ограничено ближайшей к *concurrencyLevel* степени 2.

- **ConcurrentSkipListMap** (*java.util.concurrent.ConcurrentSkipListMap*)

- Сбалансированная многопоточная ключ-значение структура ($O(\log n)$). Поиск основан на списке с пропусками. Карта должна иметь возможность сравнивать ключи.

- **ConcurrentSkipListSet** (*java.util.concurrent.ConcurrentSkipListSet*)

- ConcurrentSkipListMap* без значений.

- **CopyOnWriteArrayList** (*java.util.concurrent.CopyOnWriteArrayList*)

- Блокирующий на запись, не блокирующий на чтение список. Любая модификация создает новый экземпляр массива в памяти.

- **CopyOnWriteArraySet** (*java.util.concurrent.CopyOnWriteArraySet*)

- CopyOnWriteArrayList* без значений.

- **DelayQueue** (*java.util.concurrent.DelayQueue*)

- PriorityBlockingQueue* разрешающая получить элемент только после определенной задержки (задержка объявляется через *Delayed* интерфейс объекта). *DelayQueue* может быть использована для реализации планировщика. Емкость очереди не фиксирована.

- **LinkedBlockingDeque** (*java.util.concurrent.LinkedBlockingDeque*)

- Двунаправленная *BlockingQueue*, основанная на связанности (*cache-miss* & *cache coherence overhead*). Емкость очереди не фиксирована.

- **LinkedBlockingQueue** (*java.util.concurrent.LinkedBlockingQueue*)

- Однонаправленная *BlockingQueue*, основанная на связанности (*cache-miss* & *cache coherence overhead*). Емкость очереди не фиксирована.

- **LinkedTransferQueue** (*java.util.concurrent.LinkedTransferQueue*)

- Однонаправленная *BlockingQueue*, основанная на связанности (*cache-miss* & *cache coherence overhead*). Емкость очереди не фиксирована. Данная очередь позволяет ожидать, когда элемент «заберет» обработчик.

- **PriorityBlockingQueue** (*java.util.concurrent.PriorityBlockingQueue*)

- Однонаправленная *BlockingQueue*, разрешающая приоритизировать сообщения (через сравнение элементов). Запрещает *null* значения.

- **SynchronousQueue** (*java.util.concurrent.SynchronousQueue*)

- Однонаправленная *BlockingQueue*, реализующая *transfer()* логику для *put()* методов.

- **Synchronization points** (*java.util.concurrent.**)

- **CountDownLatch** (*java.util.concurrent.CountDownLatch*)

- Барьер (*await()*), ожидающий конкретного (или больше) кол-ва вызовов *countDown()*. Состояние барьера не может быть сброшено.

- **CyclicBarrier** (*java.util.concurrent.CyclicBarrier*)

- Барьер (*await()*), ожидающий конкретного кол-ва вызовов *await()* другими потоками. Когда кол-во потоков достигнет указанного будет вызван опциональный *callback* и блокировка снимется. Барьер сбрасывает свое состояние в начальное при освобождении ожидающих потоков и может быть использован повторно.

- **Exchanger** (*java.util.concurrent.Exchanger*)

- Барьер (*exchange()*) для синхронизации двух потоков. В момент синхронизации возможна *volatile* передача объектов между потоками.

- **Phaser** (*java.util.concurrent.Phaser*)

- Расширение *CyclicBarrier*, позволяющая регистрировать и удалять участников на каждый цикл барьера.

➤ **Semaphore** (*java.util.concurrent.Semaphore*)

Барьер, разрешающий только указанному кол-во потоков захватить монитор. По сути расширяет функционал `Lock` возможность находиться в блоке несколькими потоками.

- **Executors** (*java.util.concurrent.**)

➤ **ExecutorService** (*java.util.concurrent.ExecutorService*)

ExecutorService пришел на замену `new Thread(runnable)` чтобы упростить работу с потоками. ExecutorService помогает повторно использовать освободившиеся потоки, организовывать очереди из задач для пула потоков, подписываться на результат выполнения задачи. Вместо интерфейса Runnable пул использует интерфейс Callable (умеет возвращать результат и кидать ошибки).

Метод `invokeAll` класса ExecutorService отдает управление вызвавшему потоку только по завершению всех задач.

Метод `invokeAny` класса ExecutorService возвращает результат первой успешно выполненной задачи, отменяя все последующие.

➤ **ThreadPoolExecutor** (*java.util.concurrent.ThreadPoolExecutor*)

Пул потоков с возможностью указывать рабочее и максимальное кол-во потоков в пуле, очередь для задач.

Более легкий пул потоков для «самовоспроизводящих» задач. Пул ожидает вызовов `fork()` и `join()` методов у дочерних задач в родительской.

➤ **ScheduledThreadPoolExecutor**
(*java.util.concurrent.ScheduledThreadPoolExecutor*)

Расширяет функционал ThreadPoolExecutor возможностью выполнять задачи отложено или регулярно.

- **Accumulators** (*java.util.concurrent.atomic.LongAccumulator, java.util.concurrent.atomic.DoubleAccumulator*)

Аккумуляторы позволяют выполнять примитивные операции (сумма/поиск максимального значения) над числовыми элементами в многопоточной среде без использования CAS.

95. Что происходит при добавлении в ArrayList нового элемента и как это реализовано.

Ответ

ArrayList — реализует интерфейс List. Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

- **Создание объекта:**

```
ArrayList<String> list = new ArrayList<String>();
```

Только что созданный объект list, содержит свойства **elementData** и **size**.

Хранилище значений **elementData** есть ни что иное как массив определенного типа (указанного в generic), в нашем случае **String[]**. Если вызывается конструктор без параметров, то по умолчанию будет создан массив из 10-ти элементов типа Object (с приведением к типу, разумеется).

```
elementData = (E[]) new Object[10];
```

Но можно использовать конструктор **ArrayList(capacity)** и указать свою начальную емкость списка.

- **Добавление элементов:**

Для добавления элементов в ArrayList определен метод **add(value)**.

```
list.add("0");
```

Внутри метода `add(value)` происходят следующие вещи:

a) проверяется, достаточно ли места в массиве для вставки нового элемента;

```
ensureCapacity(size + 1);
```

b) добавляется элемент в конец (согласно значению **size**) массива.

```
elementData[size++] = element;
```

Если места в массиве недостаточно, ёмкость массива увеличивается (ёмкость рассчитывается по формуле **(oldCapacity * 3) / 2 + 1**). Копирование элементов осуществляется с помощью **native** метода **System.arraycopy()**.

```
// newCapacity - новое значение емкости
elementData = (E[]) new Object[newCapacity];
```

```
// oldData - временное хранилище текущего массива с данными
System.arraycopy(oldData, 0, elementData, 0, size);
```

96. Thread-safe and non-thread safe collections.

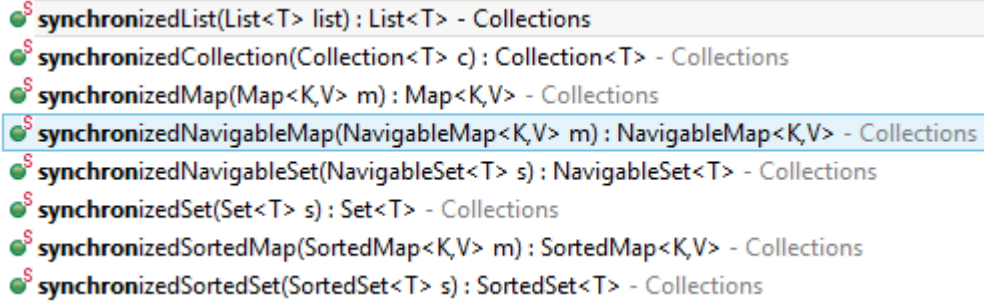
Ответ

Потокобезопасные (синхронизированные) коллекции (thread-safe collections) – это коллекции у которых все методы синхронизированные.

Синхронизированные коллекции находятся в пакете `java.util.concurrent.*` (подробнее о коллекциях описано в вопросе 95) или воспользоваться методами Collections framework.

Чтобы из коллекции сделать синхронизированную коллекцию необходимо вызвать у класса Collections соответствующий метод и передать в параметры метода ссылку на не синхронизированную коллекцию:

```
List<String> list1 = Collections.synchronizedList(new ArrayList<>());
```



The screenshot shows a list of methods from the Collections Framework used for synchronization. Each method is preceded by a green icon with a red 'S'. The methods are:

- `synchronizedList(List<T> list) : List<T> - Collections`
- `synchronizedCollection(Collection<T> c) : Collection<T> - Collections`
- `synchronizedMap(Map<K,V> m) : Map<K,V> - Collections`
- `synchronizedNavigableMap(NavigableMap<K,V> m) : NavigableMap<K,V> - Collections`
- `synchronizedNavigableSet(NavigableSet<T> s) : NavigableSet<T> - Collections`
- `synchronizedSet(Set<T> s) : Set<T> - Collections`
- `synchronizedSortedMap(SortedMap<K,V> m) : SortedMap<K,V> - Collections`
- `synchronizedSortedSet(SortedSet<T> s) : SortedSet<T> - Collections`

97. Метод для преобразования потоко-небезопасной коллекции в потоко-безопасную.

Ответ

Вопрос 96.

98. Написать метод, в котором проверяется HashMap на наличие в нем некоторого значения, и его извлечения, если такого значения нет, надо добавить значение с пустой строкой и её вернуть. Написать код, чтобы он был как можно более эффективным (меньше затратных действий).

99. Какие потокобезопасные коллекции более «быстрые» – legacy(Vector, HashTable) или из пакета concurrent?

Ответ

Коллекции из пакета `java.util.concurrent.*` используют более быстрый алгоритм, когда блокируется не вся коллекция целиком при каждом чихе, а только часть (блок), поэтому Vector, HashTable, Stack не стоит использовать ни в многопоточности (они работают медленнее, чем новые коллекции), ни в однопоточном - они просто избыточны.

100. Если в коллекцию часто добавлять элементы, и удалять, какую лучше использовать? Почему? Как они устроены?

Ответ

Если необходимо вставлять (или удалять) в середину коллекции много элементов, то лучше использовать LinkedList. Во всех остальных случаях – ArrayList.

ArrayList реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

LinkedList реализован внутри по-другому. Он реализован в виде связанного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей. А вот чтобы получить элемент с номером 130, нужно пройти последовательно по всем объектам от 0 до 130. Другими словами, операции set и get тут реализованы очень медленно.

101. Как быстро получить копию коллекции. Записать код преобразования.

Ответ

Методы для копирования коллекции:

- **Использование конструктора копирования**

Мы можем использовать конструктор копирования для клонирования списка, который представляет собой специальный конструктор для создания нового объекта как копии существующего объекта.

```
public static <T> List<T> clone(List<T> original) {  
    List<T> copy = new ArrayList<>(original);  
    return copy;  
}
```

- **Использование `addAll(Collection<? extends E> c)` метод**

List интерфейс имеет `addAll()` метод, который добавляет все элементы указанной коллекции в конец списка. Мы можем использовать то же самое для копирования элементов из исходного списка в пустой список.

```
public static <T> List<T> clone(List<T> original) {  
    List<T> copy = new ArrayList<>();  
    copy.addAll(original);  
    return copy;  
}
```

- **Использование Java 8**

Мы также можем использовать потоки в Java 8 и выше для клонирования списка.

```
public static <T> List<T> clone(List<T> original) {  
    List<T> copy = original.stream().collect(Collectors.toList());  
    return copy;  
}
```

- **Использование `Object.clone()` метод**

Java Object класс обеспечивает `clone()` метод, который можно переопределить, реализуя Cloneable интерфейс. Данный метод должен быть переопределён у объектов, которые лежат в коллекции и класс объектов должен имплементировать данный интерфейс.

Идея состоит в том, чтобы пройти по списку, клонировать каждый элемент и добавить его в клонированный список.

Мы также можем использовать Java 8 Stream, чтобы сделать то же самое.

```
class Person implements Cloneable {  
    private String name;  
    private int age;  
  
    public Person(String name,  
        Integer age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    protected Person clone() {  
        return new Person(name,  
            age);  
    }  
}
```

```
class Main {  
    public static List<Person> clone(List<Person> original) {  
        List<Person> copy = new  
            ArrayList<>(original.size());  
        for (Person person : original) {  
            copy.add(person.clone());  
        }  
        return copy;  
    }  
  
    public static List<Person> cloneStream(List<Person>  
        original) {  
        List<Person> copy = original.stream()  
            .map(Person::clone)  
            .collect(Collectors.toList());  
        return copy;  
    }  
  
    public static void main(String[] args) {  
        List<Person> original = Arrays.asList(new  
            Person("John", 25), new Person("Kim", 20));  
        List<Person> cloneList = clone(original);  
        List<Person> cloneStreamList = cloneStream(original);  
    }  
}
```

Functional Programming

102. Функциональные интерфейсы. Определение. Default & static методы. Область применения.

Ответ

- **функциональные интерфейсы**

В 8 версии Java появилось понятие **функциональные интерфейсы**.

Функциональный интерфейс — это интерфейс, который содержит **ровно один** абстрактный метод, то есть описание метода без тела. Статические методы, методы по умолчанию и переопределённые методы или методы для переопределения (с телом и без) при этом не в счёт, их в функциональном интерфейсе может быть сколько угодно.

Аннотация `@FunctionalInterface` не является чем-то сверхсложным и важным, так как её предназначение — сообщить компилятору, что данный интерфейс функциональный и должен содержать не более одного метода.

- **статические и дефолтные методы**

В 8 версии Java появилось понятие **статические и дефолтные методы в интерфейсах**.

Статические методы — это методы, в шапке которых, находится ключевое слово **static**. Эти методы принадлежат классу, а не объекту класса и должны быть определены в интерфейсе, то есть должны иметь тело метода.

Дефолтные методы — это методы, в шапке которых, находится ключевое слово **default**. Они принадлежат объекту класса, а не классу и должны быть определены в интерфейсе, то есть должны иметь тело метода.

Пример функционального интерфейса:

```
@FunctionalInterface
public interface MyInterface {

    public void sum(int a, int b);

    public default boolean methodOne() {
        return true;
    }

    public static void methodTwo() {
    }

    @Override
    public boolean equals(Object object);
}
```

103. Лямбда-выражение. Замыкания. Синтаксис. Характеристики.

Ответ

Лямбда-выражения появились с 8 версии Java.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку **->**. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Пример переопределения метода из функционального интерфейса с помощью лямбды и анонимного класса:

```
Runnable runnable = () ->
System.out.println("run"); // с помощью лямбды
```

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("run");
    }
}; // с помощью анонимного класса
```

104. Function, Supplier, Predicate, Consumer. Их применение.

Ответ

• **Predicate** (*java.util.function.Predicate*)

Predicate — функциональный интерфейс для проверки соблюдения некоторого условия. Если условие соблюдается, возвращает **true**, иначе — **false**:

Что содержит функциональный интерфейс	Пример использования
<pre>@FunctionalInterface public interface Predicate<T> { boolean test(T t); }</pre>	<pre>public static void main(String[] args) { Predicate<Integer> isEvenNumber = x -> x % 2 == 0; isEvenNumber.test(4); // true isEvenNumber.test(3); // false }</pre>

• **Consumer** (*java.util.function.Consumer*)

Consumer (с англ. — “потребитель”) — функциональный интерфейс, который принимает в качестве входного аргумента объект типа **T**, совершает некоторые действия, но при этом ничего не возвращает:

Что содержит функциональный интерфейс	Пример использования
<pre>@FunctionalInterface public interface Consumer<T> { void accept(T t); }</pre>	<pre>public static void main(String[] args) { Consumer<String> greetings = x -> System.out.println("Hello " + x + " !!!"); greetings.accept("Elena"); // вывод в консоль: Hello Elena !!! }</pre>

• **Supplier** (*java.util.function.Supplier*)

Supplier (с англ. — поставщик) — функциональный интерфейс, который не принимает никаких аргументов, но возвращает некоторый объект типа **T**:

Что содержит функциональный интерфейс	Пример использования
<pre>@FunctionalInterface public interface Supplier<T> { T get(); }</pre>	<pre>public static void main(String[] args) { List<String> nameList = new ArrayList<>(); nameList.add("Elena"); nameList.add("John"); nameList.add("Alex"); nameList.add("Jim"); nameList.add("Sara"); Supplier<String> randomName = () -> { int value = (int) (Math.random() * nameList.size()); return nameList.get(value); }; randomName.get(); // метод вернёт случайный элемент из списка }</pre>

- **Function** (*java.util.function.Function*)

Function — этот функциональный интерфейс принимает аргумент T и приводит его к объекту типа R, который и возвращается как результат:

Что содержит функциональный интерфейс	Пример использования
<pre>@FunctionalInterface public interface Function<T, R> { R apply(T t); }</pre>	<pre>public static void main(String[] args) { Function<String, Integer> valConvert = x -> Integer.valueOf(x); valConvert.apply("678"); // вернётся число 678 }</pre>

105. Ссылка на метод? Что это такое? Или это все же ссылка на объект?

Ответ

Ссылка на метод - это сокращенный синтаксис выражения лямбда, который выполняет только один метод. Это позволяет нам ссылаться на конструкторы или методы, не выполняя их. Для ссылки на метод используется оператор double colon (: :).

Ссылка на метод может использоваться для указания следующих типов методов:

- Статические методы
- Методы экземпляра
- Конструкторы, использующие новый оператор (`TreeSet::new`)

Примеры

Использование лямбды	Использование ссылки на метод
<pre>public static void main(String[] args) { List<String> list = new ArrayList<>(); Collections.addAll(list, "Привет", "как", "дела?"); list.forEach((s) -> System.out.println(s)); }</pre>	<pre>public static void main(String[] args) { List<String> list = new ArrayList<>(); Collections.addAll(list, "Привет", "как", "дела?"); list.forEach(System.out::println); }</pre>

106. Собственные функциональные интерфейсы.

Ответ

При создании собственного функционального интерфейса необходимо сделать следующее:

- Создать интерфейс;
- Сделать в нём один метод без тела (абстрактный);
- Над шапкой интерфейса сделать аннотацию `@FunctionalInterface`;
- Функциональный интерфейс готов.

Java. Streams

107. Чем Stream отличается от коллекции?

Ответ

Разница между коллекцией (`Collection`) данных и потоком (`Stream`) из новой JDK8 в том, что коллекции позволяют работать с элементами по-отдельности, тогда как поток (`Stream`) не позволяет. Например, с использованием коллекций, вы можете добавлять элементы, удалять, и вставлять в середину. Поток (`Stream`) не позволяет манипулировать отдельными элементами из набора данных, но вместо этого позволяет выполнять функции над данными как одним целым.

108. Промежуточные и терминальные операции.

Ответ

Операторы можно разделить на две группы:

- **Промежуточные** (“intermediate”, ещё называют “lazy”) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
- **Терминальные** (“terminal”, ещё называют “eager”) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

Пример:

```
public static void main(String[] args) {
    /* 1. */List<String> list = new ArrayList<>();
    /* 2. */list.add("One");
    //
    /* 11. */list.add("Ten");
    /* 12. */Stream<String> stream = list.stream();
    /* 13. */stream.filter(x -> x.toString().length() == 3).forEach(System.out::println);
}
```

Что здесь происходит:

- 1 — создаём список list;
- 2-11 — заполняем его тестовыми данными;
- 12 — создаём объект Stream;
- 13 — метод filter (фильтр) — промежуточный оператор, x приравнивается к одному элементу коллекции для перебора (как при for each) и после -> мы указываем как фильтруется наша коллекция и так как это промежуточный оператор, отфильтрованная коллекция идёт дальше в метод forEach который в свою очередь является терминальным (конечным) аналогом перебора for each (Выражение System.out::println сокращенно от: x-> System.out.println(x)), которое в свою очередь проходит по всем элементам переданной ему коллекции и выводит её).

109. Методы: map() vs flatMap().

Ответ

Оба map(), а также flatMap() принимает функцию отображения, которая применяется к каждому элементу Stream<T> и возвращает Stream<R>. Отличие лишь в том, что способ отображения в случае flatMap() производит поток новых значений, тогда как для map(), он создает одно значение для каждого входного элемента.

Arrays.stream(), List.stream(), и т. д., обычно используются методы отображения для flatMap(). Поскольку метод отображения для flatMap() возвращает другой поток, мы должны получить поток потоков. Однако, flatMap() имеет эффект замены каждого сгенерированного потока содержимым этого потока. Другими словами, все отдельные потоки, созданные методом, объединяются в один поток.

Пример использования:

map()	flatMap()
<pre>public static void main(String[] args) { Stream.of('1', '2', '3') // Stream<Character> .map(String::valueOf) // Stream<String> .map(Integer::parseInt); // Stream<Integer> }</pre>	<pre>public static void main(String[] args) { List<Integer> a = Arrays.asList(1, 2, 3); List<Integer> b = Arrays.asList(4, 5); List<Integer> c = Arrays.asList(6, 7, 8); List<List<Integer>> listOfListOfInts = Arrays.asList(a, b, c); // [[1, 2, 3], [4, 5], [6, 7, 8]] List<Integer> listOfInts = listOfListOfInts.stream() .flatMap(list -> list.stream()) .collect(Collectors.toList()); // [1, 2, 3, 4, 5, 6, 7, 8] }</pre>

110. Что такое потоковая обработка данных.

Ответ

Потоковая обработка данных — это процедура, когда данные обрабатываются с помощью потока (Stream), то есть элемент за элементом. Для этого нужно:

- а) создать коллекцию или массив;

```
List<Integer> list = new ArrayList<>(); // коллекция
int[] array = new int[5]; // массив
```

b) наполнить данными коллекцию (массив);

```
for (int i = 0; i < 5; i++) {
    list.add(i);
}
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

c) создать Stream с данными. Для этого необходимо:

- если коллекция, вызвать метод **stream()**;

```
list.stream()
```

• если массив, вызвать у коллекции **Arrays** статический метод **stream** и передать в качестве параметра массив в метод;

```
Arrays.stream(array);
```

JDBC

111. Как создать Connection?

Ответ

Порядок действий по созданию Connection к базе данных:

- Скачать и установить срезку разработки и jdk;
- Скачать программу для работы с базой данных MySQL;
- Создать базу данных;

```
CREATE database IF NOT EXISTS market;
```

d) Скачать библиотеку “mysql-connector-java-[version]-bin.jar” для СУБД MySQL;

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.32</version>
</dependency>
```

e) Создать фабрику для создания DriverManager (Пример ниже);

```
public class MySQLDataSourceFactory {

    /** The Constant LOGGER. */
    private static final Logger log = LogManager.getLogger();

    /** The Constant DATABASE_PROPERTIES_PATH. */
    private static final String DATABASE_PROPERTIES_PATH = "database_properties/property.txt";

    /** The Constant PROPERTY_DATABASE_URL. */
    private static final String PROPERTY_DATABASE_URL = "db.url";

    /** The Constant PROPERTY_DATABASE_USER. */
    private static final String PROPERTY_DATABASE_USER = "db.user";

    /** The Constant PROPERTY_DATABASE_PASS. */
    private static final String PROPERTY_DATABASE_PASS = "db.pass";

    /** The properties. */
    private static Properties properties;

    static {
        InputStream inputStream = MySQLDataSourceFactory.class.getClassLoader()
            .getResourceAsStream(DATABASE_PROPERTIES_PATH);
        try {
            properties = new Properties();
            properties.load(inputStream);
        } catch (IOException e) {
            log.log(Level.FATAL, "This file isn't readable by path: " + DATABASE_PROPERTIES_PATH
                + " because pathProperties is null");
            throw new MySQLDataSourceException("This file isn't readable by path: " + DATABASE_PROPERTIES_PATH
                + " because pathProperties is null");
        }
    }

    /**
     * Creates a new MySQLDataSource object.
     *
     * @return the mysql data source
     */
    public static MySQLDataSource createMySQLDataSource() {
        MySQLDataSource dataSource = null;
        dataSource = new MySQLDataSource();
        dataSource.setUrl(properties.getProperty(PROPERTY_DATABASE_URL));
        dataSource.setUser(properties.getProperty(PROPERTY_DATABASE_USER));
        dataSource.setPassword(properties.getProperty(PROPERTY_DATABASE_PASS));
        return dataSource;
    }
}
```

f) Создаём класс прокси соединений:

- Создаём класс ProxyConnection и имплементируем интерфейс Connection;

```
7 public class ProxyConnection implements Connection {  
      
    Делегируем все методы;  
      
    Создаём методы для закрытия соединения;  
      
    @Override  
    public void close() throws SQLException {  
        ConnectionPool.INSTANCE.releaseConnection(this);  
    }  
      
    /**  
     * Really close.  
     *  
     * @throws SQLException the SQL exception  
     */  
    void reallyClose() throws SQLException {  
        this.connection.close();  
    }  
}
```

g) Создаём класс ConnectionPool:

- Создаём класс ConnectionPool (с синхронизированными методами) или enum (он потоко-безопасный), так как ConnectionPool должен быть потоко-безопасным;
- В блоке инициализации создаём, с помощью созданной фабрики, DriverManager и делаем его полем объекта.;

```
/** The data source. */  
private DataSource dataSource;
```

```
{  
    dataSource = MySQLDataSourceFactory.createMySQLDataSource();  
}
```

- В классе создаём поля свободные и занятые соединения;

```
/** The free connections. */  
private BlockingQueue<ProxyConnection> freeConnections;
```

```
/** The busy connections. */  
private BlockingQueue<ProxyConnection> busyConnections;
```

- В классе создаём конструктор, где инициализируем поля очереди с соединениями:

- Создаём константу где укажем количество соединений;

```
private final int DEFAULT_POOL_SIZE = 4;
```

- Создадим метод для создания необходимого количества соединений

```
private void initializeConnectionPool() {  
    for (int i = 0; i < DEFAULT_POOL_SIZE; i++) {  
        try {  
            Connection connection = dataSource.getConnection();  
            ProxyConnection proxyConnection = new ProxyConnection(connection);  
            freeConnections.offer(proxyConnection);  
        } catch (SQLException e) {  
            log.log(Level.FATAL, "connection does not create.", e);  
            throw new ConnectionPoolException("Fatal error. Connection does not create", e);  
        }  
    }  
}
```

- Делаем скрытый конструктор;

```

        private ConnectionPool() {
            this.freeConnections = new LinkedBlockingQueue<>(DEFAULT_POOL_SIZE);
            this.busyConnections = new LinkedBlockingQueue<>();
            initializeConnectionPool();
        }
    
```

- Делаем метод для получения соединения;


```

                public ProxyConnection getConnection() {
                    ProxyConnection connection = null;
                    try {
                        connection = this.freeConnections.take();
                        this.busyConnections.offer(connection);
                    } catch (InterruptedException e) {
                        log.log(Level.ERROR, "error in method getConnection(): " + e);
                        e.printStackTrace();
                    }
                    return connection;
                }
            
```
- Делаем методы для остановки соединения (соединений);


```

                public boolean releaseConnection(ProxyConnection connection) {
                    boolean result = true;
                    if (connection instanceof ProxyConnection) {
                        this.busyConnections.remove(connection);
                        this.freeConnections.offer(connection);
                    } else {
                        result = false;
                        log.log(Level.ERROR, "error in method releaseConnection()");
                    }
                    return result;
                }

                /**
                 * Destroy pool.
                 */
                public void destroyPool() {
                    for (int i = 0; i < DEFAULT_POOL_SIZE; i++) {
                        try {
                            this.freeConnections.take().reallyClose();
                        } catch (SQLException | InterruptedException e) {
                            log.log(Level.ERROR, e.getMessage());
                            Thread.currentThread().interrupt();
                        }
                    }
                    deregisterDrivers();
                }

                /**
                 * Deregister drivers.
                 */
                private void deregisterDrivers() {
                    DriverManager.getDrivers().asIterator().forEachRemaining(driver -> {
                        try {
                            DriverManager.deregisterDriver(driver);
                        } catch (SQLException e) {
                            e.printStackTrace();
                        }
                    });
                }
            
```

h) Получаем соединение.

```

try(ProxyConnection connection =
    ConnectionPool.INSTANCE.getConnection()){
    // ...
} catch (SQLException e) {
    // ...
}

```

```

try {
    ProxyConnection connection =
        ConnectionPool.INSTANCE.getConnection();
    // ...
} catch (SQLException e) {
    // ...
}

```

112. Как правильно закрыть Connection?

Ответ

Закрывается соединение вызовом метода `close()`.

Метод вызывается вручную, или если используется блок `try-resource` или `try-catch-resource` метод вызывается автоматически.

113. Какие есть типы драйверов для соединения с СУБД?

Ответ

Использование драйверов JDBC позволяет открывать соединения с базой данных и взаимодействовать с ними, отправляя команды SQL или базы данных, а затем получая результаты с помощью Java.

Реализации драйвера JDBC различаются из-за большого разнообразия операционных систем и аппаратных платформ, в которых работает Java. Sun поделила типы реализации на четыре категории: типы 1, 2, 3 и 4.

Типы драйверов JDBC:

- **Тип 1: драйвер моста JDBC-ODBC**

В драйвере типа 1 мост JDBC используется для доступа к драйверам ODBC, установленным на каждом клиентском компьютере. Использование ODBC требует настройки в вашей системе имени источника данных (DSN), которое представляет целевую базу данных.

Когда впервые появилась Java, это был полезный драйвер, потому что большинство баз данных поддерживали только доступ ODBC, но теперь этот тип драйвера рекомендуется только для экспериментального использования или когда нет другой альтернативы.

- **Тип 2: JDBC-собственный API**

В драйвере типа 2 вызовы API JDBC преобразуются в собственные вызовы API C / C ++, которые являются уникальными для базы данных. Эти драйверы обычно предоставляются поставщиками баз данных и используются так же, как мост JDBC-ODBC. Драйвер для конкретного поставщика должен быть установлен на каждом клиентском компьютере.

Если мы изменим базу данных, нам придется изменить собственный API, поскольку он специфичен для базы данных, и в настоящее время они в основном устарели, но вы можете заметить некоторое увеличение скорости с драйвером типа 2, потому что это устраняет накладные расходы ODBC.

- **Тип 3: JDBC-Net чистая Java**

В драйвере типа 3 для доступа к базам данных используется трехуровневый подход. Клиенты JDBC используют стандартные сетевые сокеты для связи с сервером приложений промежуточного программного обеспечения. Затем информация о сокете преобразуется сервером приложений промежуточного программного обеспечения в формат вызова, требуемый СУБД, и пересылается на сервер базы данных.

Этот тип драйвера является чрезвычайно гибким, поскольку он не требует кода, установленного на клиенте, и один драйвер может фактически обеспечить доступ к нескольким базам данных.

- **Тип 4: 100% Чистая Ява**

В драйвере типа 4 драйвер на основе чистого Java напрямую связывается с базой данных поставщика через сокетное соединение. Это драйвер самой высокой производительности, доступный для базы данных, который обычно предоставляется самим поставщиком.

Этот тип драйвера чрезвычайно гибок, вам не нужно устанавливать специальное программное обеспечение на клиент или сервер. Далее, эти драйверы могут быть загружены динамически.

Драйвер MySQL Connector / J является драйвером типа 4. Из-за запатентованного характера своих сетевых протоколов поставщики баз данных обычно предоставляют драйверы типа 4.

Какой драйвер следует использовать?

Если вы обращаетесь к базе данных одного типа, например, Oracle, Sybase или IBM, предпочтительный тип драйвера – 4.

Если ваше Java-приложение обращается к нескольким типам баз данных одновременно, предпочтительным драйвером является тип 3.

Драйверы типа 2 полезны в ситуациях, когда драйверы типа 3 или 4 еще не доступны для вашей базы данных.

Драйвер типа 1 не считается драйвером уровня развертывания и обычно используется только в целях разработки и тестирования.

114. Чем отличается Statement от PreparedStatement? Где сохраняется запрос после первого вызова PreparedStatement? Будет ли тот же самый эффект как и от PreparedStatement, если формировать запрос просто в строке и отправлять его в Statement?

Оба **Statement** и **PreparedStatement** могут использоваться для выполнения запросов SQL. Эти интерфейсы очень похожи. Однако они существенно отличаются друг от друга по характеристикам и производительности:

Statement

- используется для выполнения строковых SQL-запросов;
- принимает строки как SQL-запросы. Таким образом, код становится менее читаемым, когда мы объединяем строки SQL:

```
String query = "INSERT INTO persons(id, name) VALUES(" +
    personEntity.getId() + ", '" + personEntity.getName() +
    "')";
Statement statement = connection.createStatement();
statement.executeUpdate(query);
```

- он уязвим для SQL-инъекций. Следующие примеры иллюстрируют эту слабость.

В первой строке обновление установит для столбца «имя» во всех строках значение «хакер», поскольку все, что следует после «-», интерпретируется как комментарий в SQL, а условия оператора обновления будут игнорироваться. Во второй строке вставка завершится ошибкой, потому что кавычка в столбце «имя» не экранирована:

```
dao.update(new PersonEntity(1, "hacker' --"));
dao.insert(new PersonEntity(1, "O'Brien"));
```

- JDBC передает запрос со встроенными значениями в базу данных. Поэтому нет никакой оптимизации запросов, а самое главное, все проверки должен обеспечивать движок базы данных. Кроме того, запрос не будет выглядеть так же в базе данных, и это предотвратит использование кеша. Точно так же пакетные обновления необходимо выполнять отдельно:

```
public void insert(List<PersonEntity>
    personEntities) {
    for (PersonEntity personEntity:
        personEntities) {
        insert(personEntity);
    }
}
```

- интерфейс *Statement* подходит для таких DDL-запросов, как CREATE, ALTER и DROP:

```
public void createTables() {
    String query = "create table if not exists
PERSONS (ID INT, NAME VARCHAR(45))";
    connection.createStatement()
        .executeUpdate(query);
}
```

- интерфейс *Statement* нельзя использовать для хранения и извлечения файлов и массивов.

PreparedStatement

- используется для выполнения параметризованных SQL-запросов;
- PreparedStatement* расширяет интерфейс *Statement*. Он имеет методы для привязки различных типов объектов, включая файлы и массивы. Следовательно, код становится понятным:

```
public void insert(PersonEntity personEntity) {
    String query = "INSERT INTO persons(id, name)
VALUES( ?, ?)";
```

```
PreparedStatement preparedStatement =
    connection.prepareStatement(query);
preparedStatement.setInt(1,
    personEntity.getId());
preparedStatement.setString(2,
    personEntity.getName());
preparedStatement.executeUpdate();
}
```

- он защищает от SQL-инъекций, экранируя текст для всех предоставленных значений параметров;
- PreparedStatement* использует предварительную компиляцию. Как только база данных получит запрос, она проверит кеш перед предварительной компиляцией запроса. Следовательно, если он не кэширован, механизм базы данных сохранит его для следующего использования.

Кроме того, эта функция ускоряет обмен данными между базой данных и JVM через двоичный протокол, отличный от SQL. То есть в пакетах меньше данных, поэтому связь между серверами идет быстрее.

- PreparedStatement* обеспечивает пакетное выполнение во время одного подключения к базе данных.

```
public void insert(List<PersonEntity> personEntities)
    throws SQLException {
    String query = "INSERT INTO persons(id, name)
VALUES( ?, ?)";
    PreparedStatement preparedStatement =
        connection.prepareStatement(query);
    for (PersonEntity personEntity :
        personEntities) {
        preparedStatement.setInt(1,
            personEntity.getId());
        preparedStatement.setString(2,
            personEntity.getName());
        preparedStatement.addBatch();
    }
    preparedStatement.executeBatch();
}
```

Далее, *PreparedStatement* предоставляет простой способ хранения и извлечения файлов с использованием типов данных *BLOB* и *CLOB*. Точно так же он помогает хранить списки путем преобразования *java.sql.Array* в массив SQL.

Наконец, *PreparedStatement* реализует такие методы, как *getMetadata()*, которые содержат информацию о возвращаемом результате.

115. Зачем нужен CallableStatement?

Ответ

Интерфейс **CallableStatement** предоставляет методы для выполнения хранимых процедур. Так как JDBC API предоставляет синтаксис escape хранимых процедур SQL, вы можете вызывать хранимые процедуры всех СУБД одним стандартным способом.

- **Создание CallableStatement;**

Создать объект **CallableStatement** можно используя метод `prepareCall()` интерфейса **Connection**. Этот метод принимает строковую переменную, представляющую запрос на вызов хранимой процедуры, и возвращает объект **CallableStatement**.

```
CallableStatement cstmt = con.prepareCall("{call myProcedure(?, ?, ?)}");
```

- **Создание процедуры в SQL;**

```
DELIMITER $$
CREATE PROCEDURE myProcedure(IN a INT, IN b VARCHAR(255), IN c BOOLEAN, OUT d INT)
BEGIN
SELECT users.id FROM users WHERE users.id > a INTO CD;
END $$
DELIMITER ;
```

- **Регистрация и установка параметров:**

- **Установка значений входных параметров;**

Оператор **Callable** может иметь входные параметры, выходные параметры или и то, и другое. Чтобы передать входные параметры вызову процедуры, вы можете использовать местозаполнитель и установить для них значения с помощью методов установки (`setInt()`, `setString()`, `setFloat()` и другие), предоставляемых интерфейсом **CallableStatement**.

```
cstmt.setInt(1, 5);
cstmt.setString(2, "string");
cstmt.setBoolean(3, false);
```

- **Регистрация выходных (OUT) параметров;**

```
cstmt.registerOutParameter(4, Types.INTEGER);
```

- **Выполнение вызываемого оператора;**

После того, как вы создали объект **CallableStatement**, его можно выполнить, используя один из методов **execute()**.

```
cstmt.execute();
```

- **Получение результата запроса:**

- **Значение параметра с типом OUT (зарегистрированное перед этим);**

```
int i = cstmt.getInt(4);
```

- **Результат запроса в процедуре(ах):**

- **Не более одного запроса;**

```
ResultSet resultSet = cstmt.getResultSet();
while (resultSet.next()) {
    resultSet.getInt("columnName1");
    // ...
}
```

- **Один и более запроса;**

```
boolean hasResultsSet = cstmt.execute(); // проверяем вернулся ли resultSet хоть один
while (hasResultsSet) {
    ResultSet resultSet = cstmt.getResultSet(); // получаем первый resultSet
    while (resultSet.next()) {
        int value = resultSet.getInt("columnName");
    }
    hasResultsSet = cstmt.getMoreResults(); // проверяем есть ли ещё resultSetы
}
```


116. Отличие executeUpdate от executeQuery

Ответ

ResultSet executeQuery () : выполняет команду SELECT, которая возвращает данные в виде ResultSet.

int executeUpdate () : выполняет такие SQL-команды, как INSERT, UPDATE, DELETE, CREATE и возвращает количество измененных строк.

117. Как в объекте ResultSet вернуться в предыдущую строку? Всегда ли можно вернуться в предыдущую строку?

Ответ

```
public boolean next() throws SQLException
public boolean previous() throws SQLException
```

Эти методы позволяют переместиться в результирующем наборе на одну строку вперед или назад. Во вновь созданном результирующем наборе курсор устанавливается перед первой строкой, поэтому первое обращение к методу next() влечет позиционирование на первую строку. Эти методы возвращают true, если остается строка для дальнейшего перемещения. Если строк для обработки больше нет, возвращается false. Если открыт поток InputStream для предыдущей строки, он закрывается. Также очищается цепочка предупреждений SQLWarning

118. Последовательность действий необходимых для выполнения запроса к БД.

Ответ

Последовательность действий для выполнения запроса к БД:

a) *Создание объекта Connection из ConnectionPool;*

Вопрос 111.

b) *Создание объекта Statement или PreparedStatement или CallableStatement;*

Statement

```
try (ProxyConnection connection =
    ConnectionPool.INSTANCE.getConnection();
    Statement statement =
        connection.createStatement()) {
    // ... *
} catch (SQLException e) {
    // ... *
}
```

PreparedStatement

```
try (ProxyConnection connection =
    ConnectionPool.INSTANCE.getConnection();
    PreparedStatement statement = connection.
        prepareStatement("SELECT users.id FROM
users WHERE users.id > ?")) {
    // ... *
} catch (SQLException e) {
    // ... *
}
```

CallableStatement

```
try (ProxyConnection connection = ConnectionPool.INSTANCE.getConnection();
    CallableStatement callableStatement =
        connection.prepareCall("{call myProcedure(?, ?, ?)}")) {
    // ... *
} catch (SQLException e) {
    // ... *
}
```

* - далее код идёт, вместо `...`

c) *Инициализация (регистрация) параметров запроса;*

- Инициализация входных параметров;

PreparedStatement

```
preparedStatement.setBoolean(1, false);
```

CallableStatement

```
callableStatement.setInt(1, 1);
```

- Регистрация выходных параметров (у процедур);

```
callableStatement.registerOutParameter(1, Types.INTEGER);
```

d) *Выполнение запроса;*

Statement

```
ResultSet resultSet =  
    statement.executeQuery("SELECT users.id FROM  
users");
```

PreparedStatement

```
ResultSet resultSet =  
    preparedStatement.executeQuery();
```

CallableStatement

```
callableStatement.execute();
```

e) *Получение результатов*

Statement

```
try(ResultSet resultSet =  
    statement.executeQuery("SELECT users.id FROM  
users")){  
    while(resultSet.next()) {  
        // ...  
    }  
}
```

PreparedStatement

```
try(ResultSet resultSet =  
    preparedStatement.executeQuery()){  
    while (resultSet.next()) {  
        // ...  
    }  
}
```

CallableStatement

Ответ в вопросе 115.

119. Как получить сгенерированный СУБД первичный ключ без выполнения дополнительного запроса к БД?

Ответ

Для первичных ключей разработаны различные механизмы автогенерации уникального значения. При добавлении записи в базу данных запрос на добавление данных не должен содержать информации о первичном ключе. Значение первичного ключа для этой записи будет сгенерировано базой данных автоматически. Метод `getGeneratedKeys()` возвратит значение ключа. Объект `PreparedStatement` должен быть создан с параметром `Statement.RETURN_GENERATED_KEYS`.

```
PreparedStatement statement =  
    connection.prepareStatement("SELECT users.id FROM users WHERE users.id > ?",  
    Statement.RETURN_GENERATED_KEYS);  
  
ResultSet resultSet = statement.getGeneratedKeys();  
if (resultSet.next()) {  
    int key = resultSet.getInt(1);  
}
```

Механизм автогенерации также удобен во избежание ошибки дублирования ключа.

Design Patterns

120. Зачем нужны паттерны? Привести примеры из проекта.

Ответ

Паттерны проектирования (шаблоны проектирования) – это готовые к использованию решения часто возникающих в программировании задач.

Типы паттернов:

- **Порождающие**

Порождающие паттерны – эти паттерны решают проблемы обеспечения гибкости создания объектов.

Порождающие шаблоны помогают создавать объекты удобнее, обеспечить гибкость этого процесса.

Порождающие паттерны:

- **Singleton** (Одиночка) – ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту.
- **Factory** (Фабрика) – используется, когда у нас есть суперкласс с несколькими подклассами и на основе ввода, нам нужно вернуть один из подкласса.
- **Abstract Factory** (Абстрактная фабрика) – используем супер фабрику для создания фабрики, затем используем созданную фабрику для создания объектов.
- **Builder** (Строитель) – используется для создания сложного объекта с использованием простых объектов. Постепенно он создает большой объект от малого и простого объекта.

➤ **Prototype** (Прототип) – помогает создать дублированный объект с лучшей производительностью, вместо нового создается возвращаемый клон существующего объекта.

- **Структурные**

Структурные паттерны – эти паттерны решают проблемы эффективного построения связей между объектами. Целью структурных шаблонов является построение удобных в поддержке иерархий классов и их взаимосвязей.

Структурные паттерны:

➤ **Adapter** (Адаптер) – это конвертер между двумя несовместимыми объектами. Используя паттерн адаптера, мы можем объединить два несовместимых интерфейса.

➤ **Composite** (Компоновщик) – использует один класс для представления древовидной структуры.

➤ **Proxy** (Заместитель) – представляет функциональность другого класса.

➤ **Flyweight** (Легковес) – вместо создания большого количества похожих объектов, объекты используются повторно.

➤ **Facade** (Фасад) – обеспечивает простой интерфейс для клиента, и клиент использует интерфейс для взаимодействия с системой.

➤ **Bridge** (Мост) – делает конкретные классы независимыми от классов реализации интерфейса.

➤ **Decorator** (Декоратор) – добавляет новые функциональные возможности существующего объекта без привязки его структуры.

- **Поведенческие**

Поведенческие паттерны – эти паттерны решают проблемы эффективного взаимодействия между объектами. Целью поведенческих шаблонов является обеспечить гибкость в изменении поведения объектов.

Поведенческие паттерны:

➤ **Template Method** (Шаблонный метод) – определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

➤ **Mediator** (Посредник) – предоставляет класс посредника, который обрабатывает все коммуникации между различными классами.

➤ **Chain of Responsibility** (Цепочка обязанностей) – позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.

➤ **Observer** (Наблюдатель) – позволяет одним объектам следить и реагировать на события, происходящие в других объектах.

➤ **Strategy** (Стратегия) – алгоритм стратегии может быть изменен во время выполнения программы.

➤ **Command** (Команда) – интерфейс команды объявляет метод для выполнения определенного действия.

➤ **State** (Состояние) – объект может изменять свое поведение в зависимости от его состояния.

➤ **Visitor** (Посетитель) – используется для упрощения операций над группировками связанных объектов.

➤ **Interpreter** (Интерпретатор) – определяет грамматику простого языка для проблемной области.

➤ **Iterator** (Итератор) – последовательно осуществляет доступ к элементам объекта коллекции, не зная его основного представления.

➤ **Memento** (Хранитель) – используется для хранения состояния объекта, позже это состояние можно восстановить.

121. Какие паттерны вы знаете и как их применяет Java SE.

Ответ

Вопрос 120.

122. Factory Method, Builder.

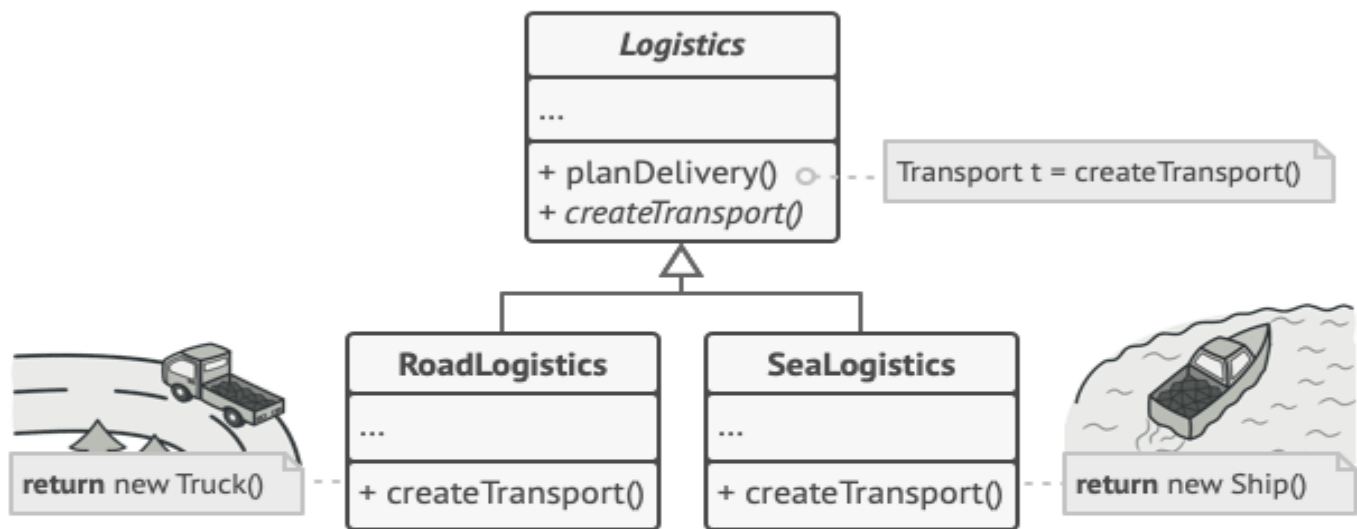
Ответ

Шаблон проектирования Factory Method

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Во всех фабричных паттернах проектирования есть две группы участников — создатели (сами фабрики) и продукты (объекты, создаваемые фабриками).

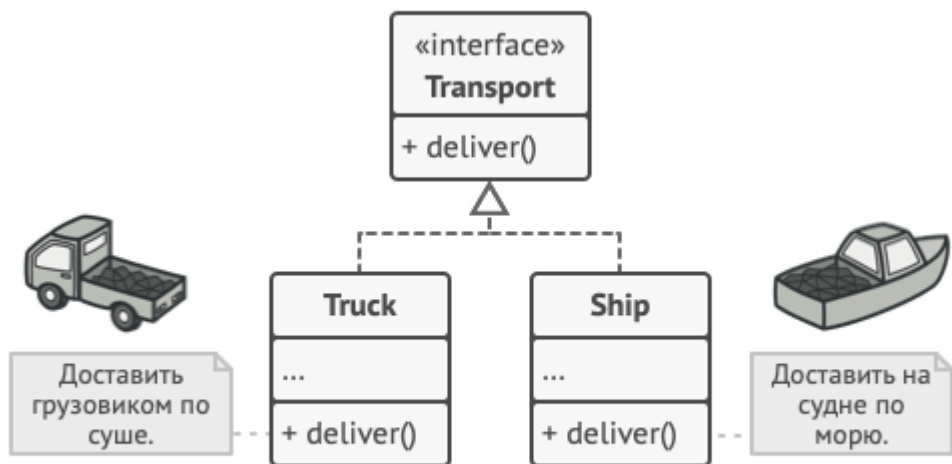
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор `new`, а через вызов особого *фабричного* метода. Не пугайтесь, объекты всё равно будут создаваться при помощи `new`, но делать это будет фабричный метод.



Подклассы могут изменять класс создаваемых объектов.

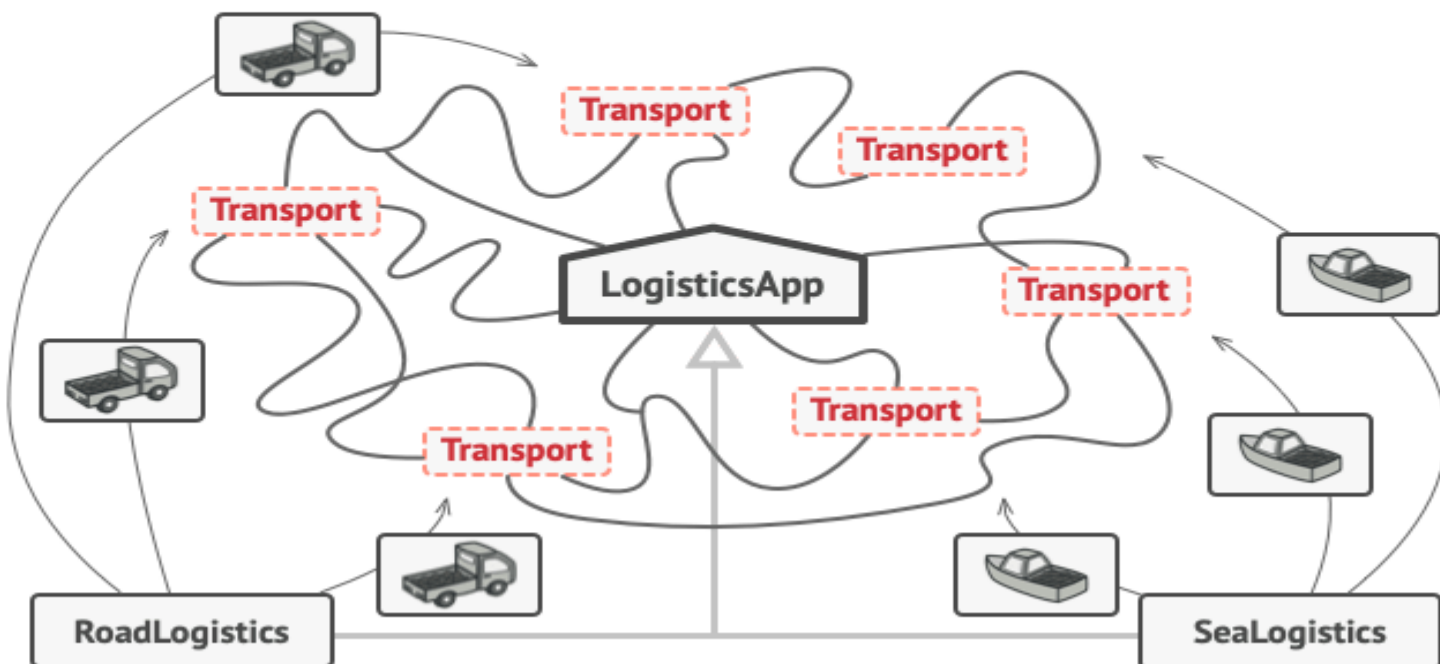
На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



Все объекты-продукты должны иметь общий интерфейс.

Например, классы **Грузовик** и **Судно** реализуют интерфейс **Транспорт** с методом **доставить**. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе **ДорожнойЛогистики** вернёт объект-грузовик, а класс **МорскойЛогистики** — объект-судно.



Пока все продукты реализуют общий интерфейс, их объекты можно взаимозаменять в клиентском коде.

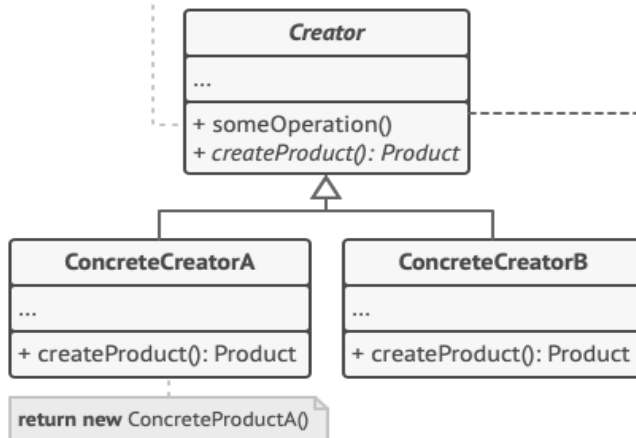
Структура

3 Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

```
Product p = createProduct()
p.doStuff()
```



1 Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

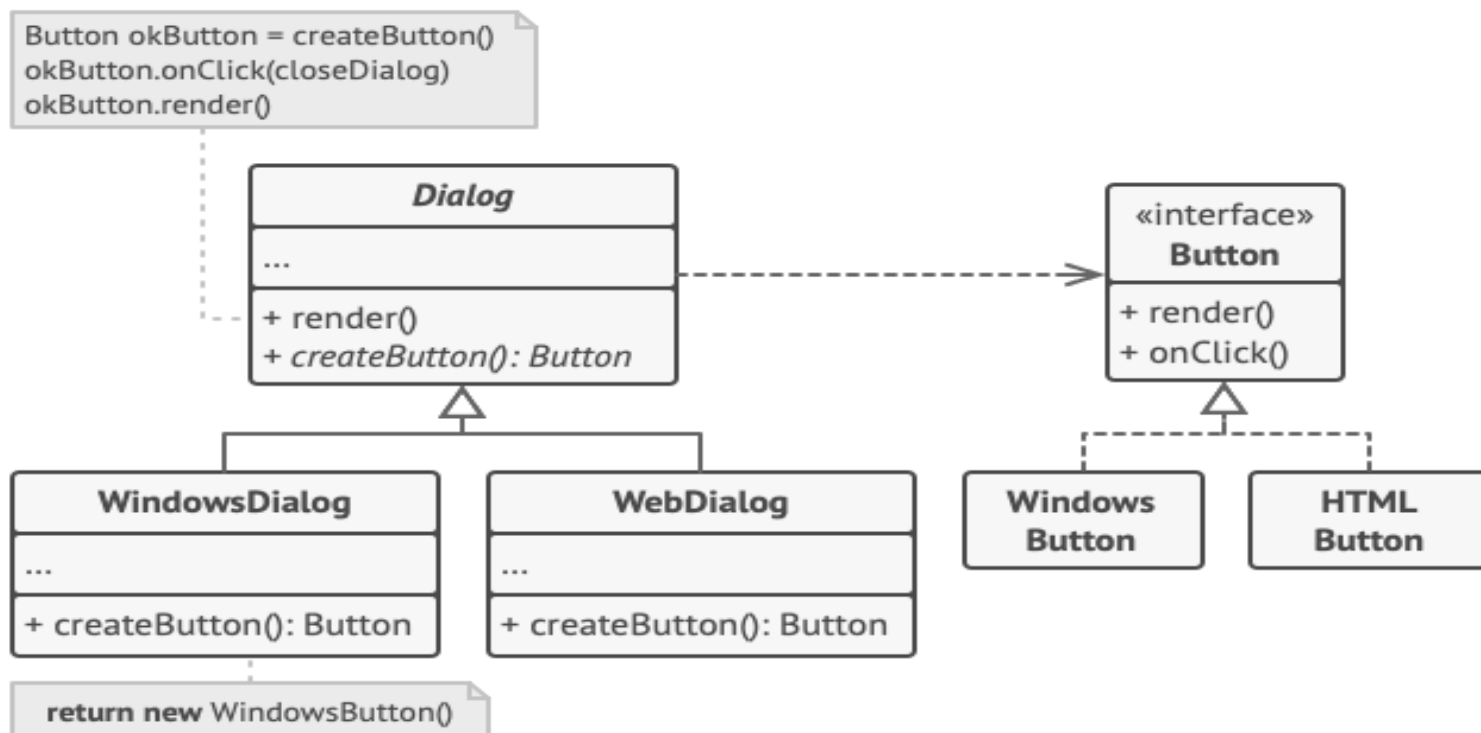
2 Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

4 Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

Пример

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.



Пример кросс-платформенного диалога.

Фабричный метод объявлен в классе диалогов. Его подклассы относятся к различным операционным системам. Благодаря фабричному методу, вам не нужно переписывать логику диалогов под каждую систему. Подклассы могут наследовать почти весь код из базового диалога, изменяя типы кнопок и других элементов, из которых базовый код строит окна графического пользовательского интерфейса.

Базовый класс диалогов работает с кнопками через их общий программный интерфейс. Поэтому, какую вариацию кнопок ни вернул бы фабричный метод, диалог останется рабочим. Базовый класс не зависит от конкретных классов кнопок, оставляя подклассам решение о том, какой тип кнопок создавать.

Такой подход можно применить и для создания других элементов интерфейса. Хотя каждый новый тип элементов будет приближать вас к **Абстрактной фабрике**.


```
// Паттерн Фабричный метод применим тогда, когда в программе
// есть иерархия классов продуктов.
```

```
interface Button is
    method render()
    method onClick(f)
```

```
class WindowsButton implements Button is
    method render(a, b) is
        // Отрисовать кнопку в стиле Windows.
    method onClick(f) is
        // Навесить на кнопку обработчик событий Windows.
```

```
class HTMLButton implements Button is
    method render(a, b) is
        // Вернуть HTML-код кнопки.
    method onClick(f) is
        // Навесить на кнопку обработчик события браузера.
```

```
// Базовый класс фабрики. Заметьте, что "фабрика" – это всего
// лишь дополнительная роль для класса. Скорее всего, он уже
// имеет какую-то бизнес-логику, в которой требуется создание
// разнообразных продуктов.
```

```
class Dialog is
    method render() is
        // Чтобы использовать фабричный метод, вы должны
        // убедиться в том, что эта бизнес-логика не зависит от
        // конкретных классов продуктов. Button – это общий
        // интерфейс кнопок, поэтому все хорошо.
        Button okButton = createButton()
        okButton.onClick(closeDialog)
        okButton.render()

    // Мы выносим весь код создания продуктов в особый метод,
    // который называют "фабричным".
    abstract method createButton():Button
```

```
class Application is
    field dialog: Dialog

    // Приложение создаёт определённую фабрику в зависимости от
    // конфигурации или окружения.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // Если весь остальной клиентский код работает с фабриками и
    // продуктами только через общий интерфейс, то для него
    // будет не важно, какая фабрика была создана изначально.
    method main() is
        this.initialize()
        dialog.render()
```

```
// Базовый класс фабрики. Заметьте, что "фабрика" – это всего
// лишь дополнительная роль для класса. Скорее всего, он уже
// имеет какую-то бизнес-логику, в которой требуется создание
// разнообразных продуктов.
```

```
class Dialog is
    method render() is
        // Чтобы использовать фабричный метод, вы должны
        // убедиться в том, что эта бизнес-логика не зависит от
        // конкретных классов продуктов. Button – это общий
        // интерфейс кнопок, поэтому все хорошо.
        Button okButton = createButton()
        okButton.onClick(closeDialog)
        okButton.render()
```

```
// Мы выносим весь код создания продуктов в особый метод,
// который называют "фабричным".
```

```
abstract method createButton():Button
```

```
// Конкретные фабрики переопределяют фабричный метод и
// возвращают из него собственные продукты.
```

```
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()
```

```
class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()
```

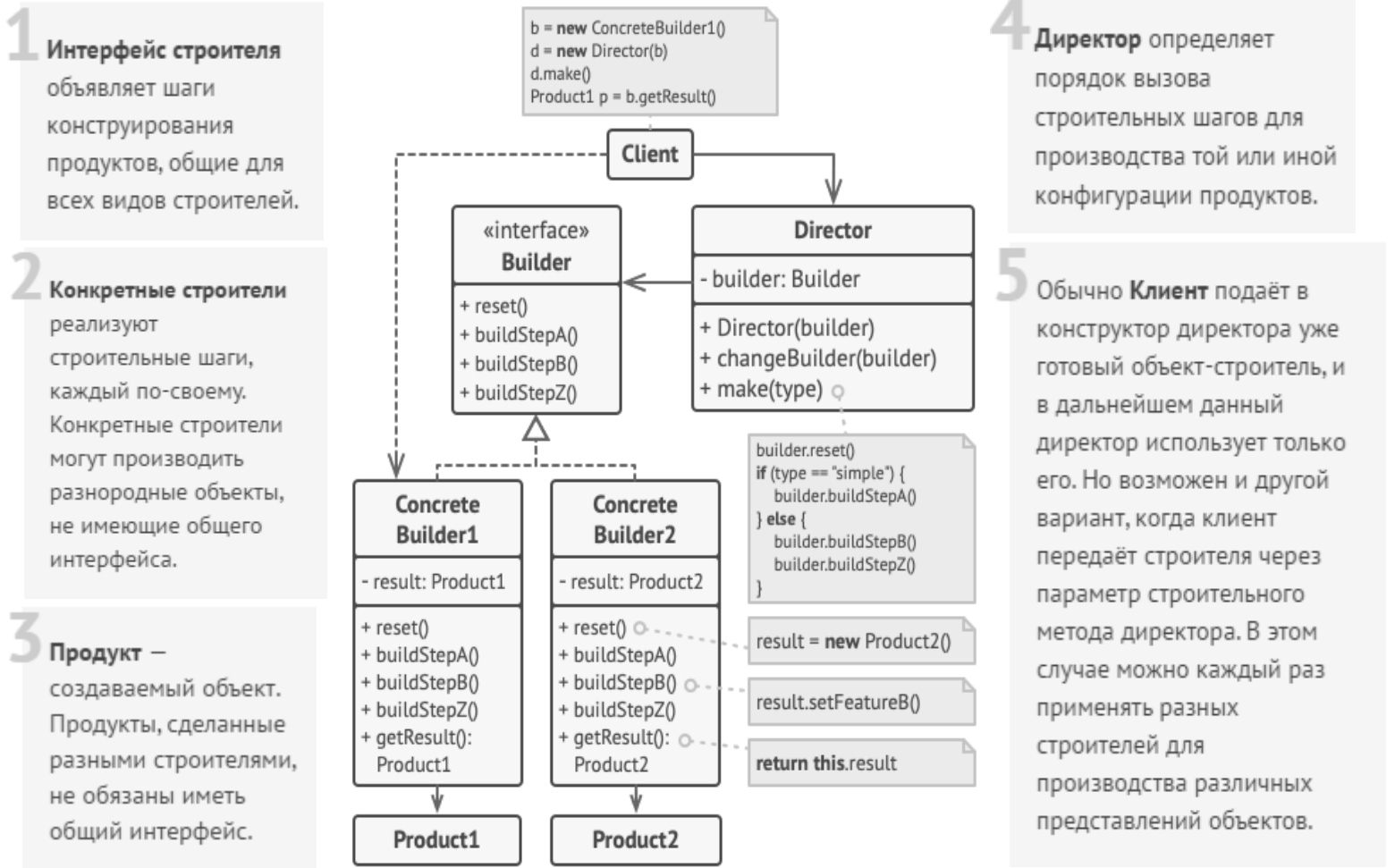
Шаблон проектирования Builder

Строитель (Builder) — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым *строителями*.

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, *построитьСтены*, *вставитьДвери* и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Структура



123. Singleton (class Runtime).

Ответ

Одиночка (Singleton) — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

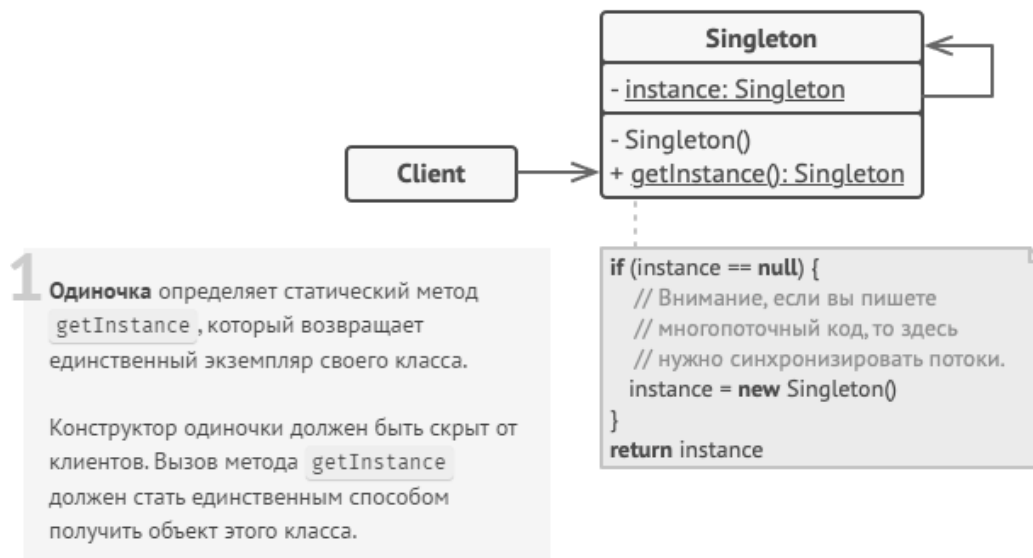
Одиночка решает сразу две проблемы, нарушая *принцип единственной ответственности* класса:

а) **Гарантирует наличие единственного экземпляра класса.** Чаше всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

б) **Предоставляет глобальную точку доступа.** Это не просто глобальная переменная, через которую можно достигаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Любой Singleton-класс отвечает сразу за две вещи: за то, что класс имеет лишь один объект, и за реализацию того, для чего этот класс вообще был создан.

Структура



Преимущества и недостатки

- | | |
|--|--|
| ✓ Гарантирует наличие единственного экземпляра класса. | ✗ Нарушает <i>принцип единственной ответственности класса</i> . |
| ✓ Предоставляет к нему глобальную точку доступа. | ✗ Маскирует плохой дизайн. |
| ✓ Реализует отложенную инициализацию объекта-одиночки. | ✗ Проблемы мультипоточности. |
| | ✗ Требуется постоянное создание Mock-объектов при юнит-тестировании. |

Класс Runtime

Класс **Runtime** (Singleton) имеет следующую структуру:

- Приватное, статическое поле где декларируется и инициализируется объект;
`private static final Runtime currentRuntime = new Runtime();`
- Приватный конструктор (пустой);
`private Runtime() {}`
- Публичный, статический метод (get) для получения объекта из приватного поля.
`public static Runtime getRuntime() {
 return currentRuntime;
}`

124. Как сделать чтобы в Singleton не работала двойная блокировка?

Ответ

Что такое двойная проверка блокировки Singleton?

Двойная проверка блокировки Singleton – это способ обеспечить создание только одного экземпляра класса Singleton в течение жизненного цикла приложения.

При двойной проверке блокировки код проверяет существующий экземпляр класса Singleton дважды с блокировкой и без нее, чтобы гарантировать, что будет создано не более одного экземпляра singleton.

Зачем нужна двойная проверка блокировки синглтон-класса?

Одним из распространенных сценариев, когда класс Singleton нарушает свои контракты, является многопоточность. Если вы попросите новичка написать код для шаблона проектирования Singleton, есть большая вероятность, что он придумает что-то вроде ниже:

```
private static Singleton _instance;

public static Singleton getInstance() {
    if (_instance == null) {
        _instance = new Singleton();
    }
    return _instance;
}
```

Но этот код создаст несколько экземпляров класса Singleton, если он будет вызван несколькими параллельными потоками, он, вероятно, синхронизирует весь этот метод `getInstance()`, как показано в нашем втором примере кода, метод `getInstanceTS()`.

```
public static synchronized Singleton getInstanceTS() {
    if (_instance == null) {
        _instance = new Singleton();
    }
    return _instance;
}
```

Хотя это потокобезопасный и решает проблему нескольких экземпляров, это не очень эффективно. Вы должны нести затраты на синхронизацию все время, когда вы вызываете этот метод, в то время как синхронизация необходима только в первом классе, когда создается экземпляр Singleton.

Это приведет нас к **двойной проверенной схеме блокировки**, где **блокируется** только критическая часть кода. Программист назвал это двойной проверкой блокировки, потому что есть две проверки для `_instance == null`, одна без блокировки, а другая с блокировкой (внутри синхронизированной) блока. Вот как выглядит двойная проверка блокировки в Java:

```
public static Singleton getInstanceDC() {
    if (_instance == null) {                // Single Checked
        synchronized (Singleton.class) {
            if (_instance == null) {        // Double checked
                _instance = new Singleton();
            }
        }
    }
    return _instance;
}
```

На первый взгляд, этот метод выглядит идеально, так как вам нужно заплатить цену за синхронизированный блок только один раз, но он все еще не работает, пока вы не сделаете переменную `_instance` изменчивой.

Без модификатора `volatile` для другого потока в Java можно увидеть половину инициализированного состояния переменной `_instance`, но с переменной `volatile`, гарантирующей связь «происходит до», вся запись будет происходить в `volatile _instance` перед любым чтением переменной `_instance`. Это было не так до Java 5, и поэтому дважды проверенная блокировка была сломана раньше. Теперь, с *гарантией «до и после»*, вы можете смело предполагать, что это работает.

Кстати, это не лучший способ создания, поточно-ориентированного Singleton, вы можете использовать Enum как Singleton, который обеспечивает встроенную потоковую безопасность при создании экземпляра.

Пример корректного Singletona

```
class Singleton {
    private volatile static Singleton _instance;
    private Singleton() {}
    public static Singleton getInstanceDC() {
        if (_instance == null) {
            synchronized (Singleton.class) {
                if (_instance == null) {
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
    }
}
```

```
// other methods ...  
}
```

125. Как у Singleton создать второй объект? И как воспрепятствовать этому.

Ответ

Загрузчик класса

Проблема в том, что классическая реализация не проверяет, существует ли один экземпляр на JVM, он лишь удостоверяется, что существует один экземпляр на classloader. Если вы пишете небольшое клиентское приложение, в котором используется лишь один classloader, то никаких проблем не возникнет. Однако если вы используете несколько загрузчиков класса или ваше приложение должно работать на сервере (где может быть запущено несколько экземпляров приложения в разных загрузчиках классов), то всё становится очень печально.

Десериализация

Singleton запрещает создавать новые объекты **через конструктор**. А ведь существуют и другие способы создать экземпляр класса, и один из них — сериализация и десериализация. Полной защиты от намеренного создания второго экземпляра Singleton'a можно добиться только с помощью использования enum'a с единственным состоянием, но это — неоправданное злоупотребление возможностями языка, ведь очевидно, что enum был придуман не для этого.

Потоконебезопасность

Причина

Один из популярных вариантов реализации Singleton содержит ленивую инициализацию. Это значит, что объект класса создаётся не в самом начале, а лишь когда будет получено первое обращение к нему.

```
public static Singleton getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
  
    return instance;  
}
```

Здесь начинаются проблемы с потоками, которые могут создавать несколько различных объектов. Происходит это примерно так:

- Первый поток обращается к `getInstance()`, когда объект ещё не создан;
- В это время второй тоже обращается к этому методу, пока первый ещё не успел создать объект, и сам создаёт его;
- Первый поток создаёт ещё один, второй, экземпляр класса.

Решение

Разумеется, можно просто пометить метод как `synchronised`, и эта проблема исчезнет. Проблема заключается в том, что, сохраняя время на старте программы, мы теперь будем терять его каждый раз при обращении к Singleton'у из-за того, что метод синхронизирован, а это очень дорого, если к экземпляру приходится часто обращаться. А ведь единственный раз, когда свойство `synchronised` действительно требуется — первое обращение к методу.

Есть два способа решить эту проблему.

- Первый — пометить как `synchronised` не весь метод, а только блок, где создаётся объект:

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized (Singleton.class) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
        }  
    }  
  
    return instance;  
}
```

Не забывайте, что это нельзя использовать в версии Java ниже, чем 1.5, потому что там используется иная модель памяти. Также не забудьте пометить поле `instance` как `volatile`.

- Второй путь — использовать паттерн «Lazy Initialization Holder». Это решение основано на том, что вложенные классы не инициализируются до первого их использования (как раз то, что нам нужно):

```
public class Singleton {

    private Singleton() {
    }

    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }

    private static class SingletonHolder {
        private static final Singleton instance = new Singleton();
    }
}
```

Рефлексия

Мы запрещаем создавать несколько экземпляров класса, пометая конструктор приватным. Тем не менее, используя рефлексия, можно без особого труда изменить видимость конструктора с `private` на `public` прямо во время исполнения:

```
Class clazz = Singleton.class;
Constructor constructor = clazz.getDeclaredConstructor();
constructor.setAccessible(true);
```

Конечно, если вы используете `Singleton` только в своём приложении, переживать не о чем. А вот если вы разрабатываете модуль, который затем будет использоваться в сторонних приложениях, то из-за этого могут возникнуть проблемы. Какие именно, зависит от того, что делает ваш «Одиночка» — это могут быть, как и риски, связанные с безопасностью, так и просто непредсказуемое поведение модуля.

126. Prototype

Ответ

Прототип (Prototype) — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Шаблон прототипа обычно используется, когда у нас есть экземпляр класса (прототип) и мы хотим создать новые объекты, просто скопировав прототип. Реализуется он как класс, который объекты которого могут клонироваться, то есть у объекта класса реализован метод `clone()` и класс имплементирует интерфейс `Cloneable`.

127. Command

Ответ

Команда (Command) — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Реализация шаблона

В классической реализации шаблон команды требует реализации четырех компонентов: **Command**, **Receiver**, **Invoker** и **Client**.

Классы команд

Команда — это объект, роль которого заключается в хранении всей информации, необходимой для выполнения действия, включая вызываемый метод, аргументы метода и объект (известный как получатель), который реализует метод.

Чтобы получить более точное представление о том, как работают объекты команд, давайте начнем разработку простого уровня команд, который включает только один интерфейс и две реализации:

```
@FunctionalInterface
public interface TextFileOperation {
    String execute();
}
```

```

public class OpenTextFileOperation implements TextFileOperation {
    private TextFile textFile;
    // constructors
    @Override
    public String execute() {
        return textFile.open();
    }
}

public class SaveTextFileOperation implements TextFileOperation {
    // same field and constructor as above
    @Override
    public String execute() {
        return textFile.save();
    }
}

```

Интерфейс *TextFileOperation* определяет API объектов команды, а две реализации, *OpenTextFileOperation* и *SaveTextFileOperation*, выполняют конкретные действия. Первый открывает текстовый файл, а второй сохраняет текстовый файл.

Функциональность командного объекта очевидна: команды *TextFileOperation* инкапсулируют всю информацию, необходимую для открытия и сохранения текстового файла, включая объект-получатель, вызываемые методы и аргументы (в данном случае аргументы не требуются, т.е. но они могут быть).

Стоит подчеркнуть, что компонент, выполняющий операции с файлами, является получателем (экземпляром *TextFile*).

Класс получателя

Получатель — это объект, который выполняет набор связанных действий. Это компонент, который выполняет фактическое действие, когда вызывается метод команды *execute()*.

В этом случае нам нужно определить класс получателя, роль которого заключается в моделировании объектов *TextFile*:

```

public class TextFile {
    private String name;
    // constructor
    public String open() {
        return "Opening file " + name;
    }
    public String save() {
        return "Saving file " + name;
    }
    // additional text file methods (editing, writing, copying, pasting)
}

```

Класс Invoker

Вызывающий объект — это объект, который знает, как выполнить данную команду, но не знает, как эта команда была реализована. Он знает только интерфейс команды.

В некоторых случаях инициатор также сохраняет и ставит в очередь команды, помимо их выполнения. Это полезно для реализации некоторых дополнительных функций, таких как запись макросов или функции отмены и повтора.

В нашем примере становится очевидным, что должен быть дополнительный компонент, отвечающий за вызов объектов команд и их выполнение через метод *execute()* команд. Именно здесь в игру вступает класс вызывающего объекта.

```

public class TextFileOperationExecutor {
    private final List<TextFileOperation> textFileOperations = new ArrayList<>();
    public String executeOperation(TextFileOperation textFileOperation) {
        textFileOperations.add(textFileOperation);
        return textFileOperation.execute();
    }
}

```

Класс *TextFileOperationExecutor* — это всего лишь тонкий слой абстракции, который отделяет объекты команд от их потребителей и вызывает метод, инкапсулированный в объектах команд *TextFileOperation*.

В этом случае класс также сохраняет объекты команд в списке. Конечно, это не является обязательным в реализации шаблона, если только нам не нужно добавить дополнительный контроль в процесс выполнения операций.

Клиентский класс

Клиент — это объект, который управляет процессом выполнения команд, указывая, какие команды выполнять и на каких этапах процесса их выполнять.

Итак, если мы хотим быть ортодоксальными с формальным определением шаблона, мы должны создать клиентский класс, используя типичный метод *main*:

```
public static void main(String[] args) {
    TextFileOperationExecutor textFileOperationExecutor = new TextFileOperationExecutor();
    textFileOperationExecutor.executeOperation(new OpenTextFileOperation(new
        TextFile("file1.txt")));
    textFileOperationExecutor.executeOperation(new SaveTextFileOperation(new
        TextFile("file2.txt")));
}
```

128. Composite

Ответ

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Его можно рассматривать как древовидную структуру, состоящую из типов, наследующих базовый тип, и он может представлять собой одну часть или целую иерархию объектов.

Шаблон можно разбить на:

- **компонент** — это базовый интерфейс для всех объектов в композиции. Это должен быть либо интерфейс, либо абстрактный класс с общими методами для управления дочерними композициями.
- **leaf (Листья)** — реализует поведение базового компонента по умолчанию. Он не содержит ссылки на другие объекты.
- **составной** — имеет листовые элементы. Он реализует методы базового компонента и определяет дочерние операции.
- **клиент** — имеет доступ к элементам композиции, используя объект базового компонента.

Пример

Предположим, мы хотим построить иерархическую структуру отделов в компании.

Базовый компонент

В качестве объекта-компонента мы определим простой интерфейс *отдела*:

```
public interface Department {
    void printDepartmentName();
}
```

Листья

Для конечных компонентов определим классы для финансового отдела и отдела продаж:

```
public class FinancialDepartment implements Department {
    private Integer id;
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }
    // standard constructor, getters, setters
}

public class SalesDepartment implements Department {
    private Integer id;
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }
    // standard constructor, getters, setters
}
```

Оба класса реализуют метод ***printDepartmentName()*** из базового компонента, где они печатают имена классов для каждого из них.

Кроме того, поскольку они являются конечными классами, они не содержат других объектов *отдела*.

Составной элемент

В качестве составного класса создадим класс ***HeadDepartment***:

```
public class HeadDepartment implements Department {
    private Integer id;
    private String name;

    private List<Department> childDepartments;
```

```

public HeadDepartment(Integer id, String name) {
    this.id = id;
    this.name = name;
    this.childDepartments = new ArrayList<>();
}

public void printDepartmentName() {
    childDepartments.forEach(Department::printDepartmentName);
}

public void addDepartment(Department department) {
    childDepartments.add(department);
}

public void removeDepartment(Department department) {
    childDepartments.remove(department);
}
}

```

Это составной класс, так как он содержит коллекцию компонентов *отдела*, а также методы добавления и удаления элементов из списка.

Составной метод ***printDepartmentName()*** реализуется путем перебора списка конечных элементов и вызова соответствующего метода для каждого из них.

Тестирование

```

public class CompositeDemo {
    public static void main(String args[]) {
        Department salesDepartment = new SalesDepartment(1, "Sales department");
        Department financialDepartment = new FinancialDepartment(2, "Financial department");

        HeadDepartment headDepartment = new HeadDepartment(3, "Head department");

        headDepartment.addDepartment(salesDepartment);
        headDepartment.addDepartment(financialDepartment);

        headDepartment.printDepartmentName();
    }
}

```

129. Chain of responsibility (Filter, closing io stream, closing connection)

Ответ

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Суть шаблона:

Каждый обрабатывающий объект в цепочке отвечает за определенный тип команды. Далее после завершения обработки он пересылает команду следующему процессору в цепочке.

Пример реализации шаблона:

```

abstract class MessagePrinter {
    private MessagePrinter nextMessagePrinter;

    public MessagePrinter getNextMessagePrinter() {
        return this.nextMessagePrinter;
    }

    public void setNextMessagePrinter(MessagePrinter nextMessagePrinter) {
        this.nextMessagePrinter = nextMessagePrinter;
    }

    public void print(String message) {
        printMessage(message);
        if (this.nextMessagePrinter != null) {
            this.nextMessagePrinter.print(message);
        }
    }
}

```



```

        public abstract void printMessage(String message);
    }

    class ConsoleMessagePrinter extends MessagePrinter {
        @Override
        public void printMessage(String message) {
            System.out.println("print to console: "
+ message);
        }
    }

    class DbMessagePrinter extends MessagePrinter {
        @Override
        public void printMessage(String message) {
            System.out.println("print to db: " + message);
        }
    }

    public static void main(String[] args) {
        MessagePrinter messagePrinter = new ConsoleMessagePrinter();
        FileMessagePrinter fileMessagePrinter = new FileMessagePrinter();
        messagePrinter.setNextMessagePrinter(fileMessagePrinter);
        fileMessagePrinter.setNextMessagePrinter(new DbMessagePrinter());
        messagePrinter.print("start process");
    }

```

130. State (Thread.State)

Ответ

Состояние (State) — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Основная идея паттерна State состоит в том, чтобы **позволить объекту изменять свое поведение без изменения его класса**. Кроме того, благодаря его реализации код должен оставаться чище без множества операторов if/else.

Пример реализации шаблона:

```

interface State {
    public void doAction(Context context);
}

class Context {
    private State state;
    private String name;

    public Context(State state, String name) {
        this.state = state;
        this.name = name;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void doAction() {
        this.state.doAction(this);
    }
}

class LowerCaseState implements State {
    @Override
    public void doAction(Context context) {
        System.out.println(context.getName()
        .toLowerCase());
    }
}

class UpperCaseState implements State {
    @Override
    public void doAction(Context context) {
        System.out.println(context.getName()
        .toUpperCase());
    }
}

```

```

public static void main(String[] args) {
    Context context = new Context(new LowerCaseState(), "Max");
    context.doAction();
    context.setState(new UpperCaseState());
    context.doAction();
}

```

131. Iterator (Enumeration, Iterator, ListIterator)

Ответ

Итератор (Iterator) — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Пример реализации шаблона:

```

interface Iterator {
    public boolean hasNext();

    public Object next();
}

class ArrayContainer implements Container {
    private String[] array;

    public ArrayContainer(String... array) {
        this.array = array;
    }

    @Override
    public Iterator getIterator() {
        return new ArrayIterator();
    }

    class ArrayIterator implements Iterator {
        private int index;

        @Override
        public boolean hasNext() {
            return index < array.length;
        }

        @Override
        public Object next() {
            if (hasNext()) {
                return array[index++];
            }
            return null;
        }
    }
}

public static void main(String[] args) {
    ArrayContainer arrayContainer = new ArrayContainer("Max", "Alex", "Jon");
    Iterator iterator = arrayContainer.getIterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}

```

132. Proxy

Ответ

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

Пример реализации шаблона:

```

interface Car {
    public void drive();
}

```

```

class Reno implements Car {
    @Override
    public void drive() {
        System.out.println("Reno driving ...");
    }
}

public static void main(String[] args) {
    Car car = new CarProxy(new Reno());
    car.drive();
}

class CarProxy implements Car {
    private Car car;

    public CarProxy(Car car) {
        this.car = car;
    }

    @Override
    public void drive() {
        System.out.println("CarProxy driving ...");
        this.car.drive();
    }
}

```

133. Observer (Listener-s)

Ответ

Наблюдатель (Observer) — это поведенческий шаблон проектирования. Он определяет связь между объектами: *наблюдаемыми* и *наблюдателями*. *Наблюдаемый* — это объект, который уведомляет *наблюдателей* об изменениях своего состояния.

Пример реализации шаблона:

Например, информационное агентство может уведомлять каналы о получении новостей. Получение новостей — это то, что изменяет состояние информационного агентства и заставляет каналы получать уведомления.

Наблюдаемый

Сначала мы определим класс NewsAgency – observable (Наблюдаемый):

```

class NewsAgency {
    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObserver(Channel channel) {
        this.channels.add(channel);
    }

    public void removeObserver(Channel channel) {
        this.channels.remove(channel);
    }

    public void setNews(String news) {
        this.news = news;
        notifyChannels();
    }

    private void notifyChannels() { // кому надо сделать update?
        for (Channel channel : this.channels) {
            channel.update(this.news);
        }
    }
}

```

NewsAgency является наблюдаемым, и когда `news` обновятся, состояние **NewsAgency** изменится. Когда происходит изменение, **NewsAgency** уведомляет об этом наблюдателей, вызывая их метод `update()`.

Чтобы сделать это, наблюдаемый объект должен хранить ссылки на наблюдателей. В нашем случае это переменная `channels`.

Наблюдатель

Теперь давайте посмотрим, как может выглядеть наблюдатель, класс **Channel**. Он должен иметь метод `update()`, который вызывается при изменении состояния **NewsAgency**:

```

class NewsChannel implements Channel {
    private String news;

    @Override
    public void update(Object news) { // что нужно сделать?
        System.out.println((String) news);
        this.setNews((String) news);
    }

    public String getNews() {
        return news;
    }

    public void setNews(String news) {
        this.news = news;
    }
}

```

Интерфейс **Channel** имеет только один метод:

```

interface Channel {
    public void update(Object o);
}

```

Теперь, если мы добавим экземпляр **NewsChannel** в список **наблюдателей** (то есть в поле **channels** объекта класса **NewsAgency**) и изменим состояние **NewsAgency** (то есть поменяем значение поля **news**), экземпляр **NewsChannel** будет обновлен (то есть вызовется метод **update**):

```

public static void main(String[] args) {
    NewsAgency observable = new NewsAgency();
    NewsChannel observer = new NewsChannel();
    observable.addObserver(observer);

    observable.notifyChannels("news1");
    observable.notifyChannels("news2");
}

```

115. Wrapper ???

Ответ???

???

116. Immutable

Ответ

Immutable класс – это не изменяемый класс.

Чтобы создать неизменяемый класс в Java, необходимо следовать этим общим принципам:

1. Объявите **класс** таким, **final** чтобы его нельзя было расширить.
2. Сделайте все **поля private** так, чтобы прямой доступ был запрещен.
3. **Не предоставляйте методы** установки для переменных.
4. Сделайте все изменяемые **поля final**, чтобы значение поля можно было назначить только один раз.
5. **Инициализируйте все поля** с помощью метода **конструктора**, выполняющего глубокое копирование.
6. Выполните **клонирование объектов** в методах получения, чтобы вернуть копию, а не вернуть реальную ссылку на объект.

117. MVC (Model-View-Controller)

Ответ

MVC — это не паттерн проектирования. MVC — это именно **набор архитектурных идей и принципов** для построения сложных систем с пользовательским интерфейсом.

Следуя паттерну MVC нужно разделять систему на три составные части. Их, в свою очередь, можно называть модулями или компонентами. У каждой составной компоненты свое предназначение.

Три составные MVC:

- **Model**

Первая компонента/модуль — так называемая модель. Она содержит всю бизнес-логику приложения.

Модель — самая независимая часть системы. Настолько независимая, что она не должна ничего знать о модулях Вид и Контроллер. Модель настолько независима, что ее разработчики могут практически ничего не знать о Виде и Контроллере.

- **View**

Вторая часть системы — вид. Данный модуль отвечает за отображение данных пользователю. Все, что видит пользователь, генерируется видом.

Основное предназначение Вида — предоставлять информацию из Модели в удобном для восприятия пользователя формате. Основное ограничение Вида — он никак не должен изменять модель.

- **Controller**

Третьим звеном данной цепи является контроллер. В нем хранится код, который отвечает за обработку действий пользователя (любое действие пользователя в системе обрабатывается в контроллере).

Основное предназначение Контроллера — обрабатывать действия пользователя. Именно через Контроллер пользователь вносит изменения в модель. Точнее в данные, которые хранятся в модели.

Пример реализации MVC:

```
interface ModelLayer {
    public Student getStudent();
}

class DBLayer implements ModelLayer {
    @Override
    public Student getStudent() {
        return new Student();
    }
}

class FileSystemLayer implements ModelLayer {
    @Override
    public Student getStudent() {
        return new Student();
    }
}

class Student {
    private String name;
    private int age;

    public Student() {
    }

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

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        StringBuilder builder =
            new StringBuilder();
        builder.append("Student [name=");
        builder.append(this.name);
        builder.append(", age=");
        builder.append(this.age);
        builder.append("]");
        return builder.toString();
    }
}

interface View {
    public void showStudent(Student student);
}

class ConsoleViewStudent implements View {
    @Override
    public void showStudent(Student student) {
        System.out.println(student.toString());
    }
}

class XMLViewStudent implements View {
    @Override
    public void showStudent(Student student) {
        System.out.println(student.toString());
    }
}

class Controller {
    private ModelLayer modelLayer = new DBLayer();
    private View view = new ConsoleViewStudent();

    public void execute() {
        Student student =
            this.modelLayer.getStudent();
        this.view.showStudent(student);
    }
}
```

```
public static void main(String[] args) {
    Controller controller = new Controller();
    controller.execute();
}
```

118. DAO vs. Repository

Ответ

DAO

Описание

DAO (Data Access Object) — шаблон DAO используется для отделения API или операций доступа к данным низкого уровня от бизнес-сервисов высокого уровня. Ниже приведены участники шаблона объекта доступа к данным.

- **Интерфейс объекта доступа к данным** — этот интерфейс определяет стандартные операции, которые должны выполняться над объектом (объектами) модели – (interface DAO).
- **Конкретный класс объекта доступа к данным** — этот класс реализует вышеуказанный интерфейс. Этот класс отвечает за получение данных из источника данных, которым может быть база данных, xml или любой другой механизм хранения.
- **Объект модели или объект значения** — этот объект представляет собой простой POJO, содержащий методы get/set для хранения данных, полученных с использованием класса DAO — (сущность).

Пример реализации шаблона

```
interface Data {
    public String getData();
}

class Db implements Data {
    @Override
    public String getData() {
        return "data form data base";
    }
}

class FileSystem implements Data {
    @Override
    public String getData() {
        return "data form file system";
    }
}

public class Main {
    private static Data data = new Db();

    public static void main(String[] args) {
        System.out.println(data.getData());
    }
}
```

Repository

Описание

Репозиторий (Repository) — это слой абстракции, инкапсулирующий в себе всё, что относится к способу хранения данных.

Репозиторий также имеет дело с данными и скрывает запросы, подобные DAO, но он находится на более высоком уровне, ближе к бизнес-логике приложения.

Следовательно, репозиторий может использовать DAO для извлечения данных из базы данных и заполнения объекта домена. Или он может подготовить данные из объекта домена и отправить их в систему хранения, используя DAO для сохранения.

Пример реализации шаблона

```
class User {
    private Long id;
    private String userName;
    private String firstName;
    private String email;

    public Long getId() {
        return id;
    }
}

class Tweet {
    private Long id;
    private String email;
    private String tweetText;
    private Date dateCreated;

    public Long getId() {
        return id;
    }
}
```

```

    public void setId(Long id) {
        this.id = id;
    }

    public String getUsername() {
        return userName;
    }

    public void setUsername(String userName) {
        this.userName = userName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName){
        this.firstName = firstName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

interface UserDao {
    public void create(User user);

    public User read(Long id);

    public void update(User user);

    public void delete(String userName);
}

interface UserRepository {
    public User get(Long id);

    public void add(User user);

    public void update(User user);

    public void remove(User user);
}

    public void setId(Long id) {
        this.id = id;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getTweetText() {
        return tweetText;
    }

    public void setTweetText(String tweetText){
        this.tweetText = tweetText;
    }

    public Date getDateCreated() {
        return dateCreated;
    }

    public void setDateCreated(Date dateCreated) {
        this.dateCreated = dateCreated;
    }
}

class UserDaoImpl implements UserDao {
    @Override
    public void create(User user) {
    }

    @Override
    public User read(Long id) {
        return null;
    }

    @Override
    public void update(User user) {
    }

    @Override
    public void delete(String userName) {
    }
}

class UserRepositoryImpl implements UserRepository {
    private UserDaoImpl userDaoImpl;

    @Override
    public User get(Long id) {
        User user = userDaoImpl.read(id);
        return user;
    }

    @Override
    public void add(User user) {
        userDaoImpl.create(user);
    }

    @Override
    public void update(User user) {
        userDaoImpl.update(user);
    }

    @Override
    public void remove(User user) {
        System.out.println("remove()");
    }
}

```



```
interface TweetDao {
    public List<Tweet> fetchTweets(String
        email);
}
```

```
class UserSocialMedia extends User {
    private List<Tweet> tweets;

    public List<Tweet> getTweets() {
        return tweets;
    }

    public void setTweets(List<Tweet> tweets) {
        this.tweets = tweets;
    }
}
```

```
class TweetDaoImpl implements TweetDao {
    @Override
    public List<Tweet> fetchTweets(String email) {
        List<Tweet> tweets = new
        ArrayList<Tweet>();
        return tweets;
    }
}
```

```
class UserTweetRepositoryImpl
implements UserRepository {
    private static UserDao userDao =
        new UserDaoImpl();
    private static TweetDao tweetDao =
        new TweetDaoImpl();

    @Override
    public User get(Long id) {
        UserSocialMedia user =
            (UserSocialMedia)
                userDao.read(id);
        List<Tweet> tweets =
            tweetDao.
                fetchTweets(user.getEmail());
        user.setTweets(tweets);
        return user;
    }

    @Override
    public void add(User user) {
    }

    @Override
    public void update(User user) {
    }

    @Override
    public void remove(User user) {
    }
}
```

DAO vs. Repository

DAO

- a) DAO — это абстракция сохраняемости данных.
- b) DAO будет считаться ближе к базе данных, часто ориентированной на таблицы.
- c) DAO не может быть реализован с использованием Репозитория.
- d) DAO обычно имеет более широкий интерфейс. Такой метод, как Update, подходит для DAO, но не для репозитория — при использовании репозитория изменения в сущностях обычно отслеживаются отдельным UnitOfWork.

Repository

- a) Репозиторий — это абстракция набора объектов.
- b) Репозиторий будет считаться ближе к бизнес-логике.
- c) Репозиторий может быть реализован с использованием DAO.
- d) Репозиторий обычно имеет более узкий интерфейс. Это должен быть просто набор объектов с Get(id), Find(ISpecification), Add(Entity).

JEE

119. Что входит в JEE?

Ответ

Java EE — что это?

Java EE — это платформа, построенная на основе Java SE, которая предоставляет API и среду времени выполнения для разработки и запуска крупномасштабных, многоуровневых, масштабируемых, надежных и безопасных сетевых приложений.

Подобные приложения называют корпоративными (Enterprise applications), так как они решают проблемы, с которыми сталкиваются большие бизнесы.

Архитектура Java EE приложений

У Java EE приложений есть структура, которая обладает двумя ключевыми качествами:

- **многоуровневость** – Java EE приложения — многоуровневые.
- **вложенность** – Есть Java EE сервер (или сервер приложений), внутри него располагаются контейнеры компонентов. В данных контейнерах размещаются компоненты.

Уровни приложений

Многоуровневые приложения — это приложения, которые разделены по функциональному принципу на изолированные модули (уровни, слои).

Обычно (в том числе в контексте Java EE разработки) корпоративные приложения делят на три уровня:

- **клиентский уровень;**

Клиентский уровень представляет из себя некоторое приложение, которое запрашивает данные у Java EE сервера (среднего уровня). Сервер, в свою очередь, обрабатывает запрос клиента и возвращает ему ответ. Клиентским приложением может быть браузер, отдельное приложение (мобильное либо десктопное) или другие серверные приложения без графического интерфейса.

- **средний уровень;**

Средний уровень подразделяется, в свою очередь, на:

➤ ***web-уровень;***

На web-уровне используются такие технологии Java EE:

- JavaServer Faces technology (JSF);
- Java Server Pages (JSP);
- Expression Language (EL);
- Servlets;
- Contexts and Dependency Injection for Java EE (CDI).

➤ ***уровень бизнес-логики;***

Уровень бизнес-логики состоит из компонент, в которых реализована вся бизнес-логика приложения. Бизнес-логика — это код, который обеспечивает функциональность, покрывающую нужды некоторой конкретной бизнес сферы (финансовая индустрия, банковское дело, электронная коммерция). Данный уровень можно считать ядром всей системы.

Технологии, которые задействованы на данном уровне:

- Enterprise JavaBeans (EJB);
- JAX-RS RESTful web services;
- Java Persistence API entities;
- Java Message Service.

- **уровень доступа к данным.**

Уровень доступа к данным. Данный уровень иногда называют уровнем корпоративных информационных систем (Enterprise Information Systems, сокращенно — EIS). EIS состоит из различных серверов баз данных, ERP (англ. Enterprise Resource Planning) систем планирования ресурсов предприятия и прочих источников данных. К этому уровню за данными обращается уровень бизнес-логики.

В данном уровне можно встретить такие технологии, как:

- Java Database Connectivity API (JDBC);
- Java Persistence API;
- Java EE Connector Architecture;
- Java Transaction API (JTA).

120. Что такое сервер приложений? Что такое веб-сервер, в чём его отличие от сервера приложений? Привести примеры веб-сервера и сервера приложений.

Ответ

Сервер — это центральное место, где информация и программы хранятся и доступны приложениям по сети.

Сервер приложений

Сервер приложений — это виртуальная машина Java™ (JVM), на которой выполняются пользовательские приложения. Сервер приложений взаимодействует с веб-сервером, чтобы вернуть динамический, настраиваемый ответ на запрос клиента.

Веб-сервер

Веб-сервер — это программное обеспечение, которое может обрабатывать запросы клиентов и отправлять ответ обратно. Например, Apache — один из наиболее широко используемых веб-серверов. Он работает на физической машине и прослушивает запросы клиентов на определенном порту.

Веб-сервер — это сервер, который принимает запрос данных и отправляет соответствующий документ в ответ, тогда как **сервер приложений** также содержит компонент контейнера EJB для запуска корпоративных приложений.

Отличие веб-сервера от сервера приложений

Фактор	Веб сервер	Сервер приложений
Цель	Веб-сервер содержит только веб-контейнер.	Сервер приложений содержит веб-контейнер и контейнер EJB.
Полезный	Веб-сервер хорош в случае статического содержимого, такого как статические html-страницы.	Сервер приложений актуален в случае динамического содержимого, такого как веб-сайты банков.
Потребление ресурсов	Веб-сервер потребляет меньше ресурсов ЦП и памяти по сравнению с сервером приложений.	Сервер приложений использует больше ресурсов.
Целевая среда	Веб-сервер обеспечивает среду выполнения для веб-приложений.	Серверы приложений обеспечивают среду выполнения для корпоративных приложений.
Поддержка многопоточности	Многопоточность не поддерживается.	Поддерживается многопоточность.
Поддерживаемые протоколы	Веб-серверы поддерживают протокол HTTP.	Серверы приложений поддерживают протоколы HTTP, а также протоколы RPC/RMI.
Пример	Веб-сервер Apache.	Веблогик, JBoss.

121. Что такое контейнер сервлетов? Что такое сервлет?

Ответ

Контейнер сервлетов — это программа, которая запускается на сервере и умеет взаимодействовать с созданными нами сервлетами. Иными словами, если мы хотим запустить наше веб-приложение на сервере, мы сначала разворачиваем контейнер сервлетов, а потом помещаем в него сервлеты.

Схема работы проста: когда клиент обращается на сервер, контейнер обрабатывает его запрос, определяет, какой именно сервлет должен его обработать и передает его.

Сервлет — это класс, который умеет получать запросы от клиента и возвращать ему ответы. Сервлеты в Java — именно те элементы, с помощью которых строится клиент-серверная архитектура.

122. Методы сервлета. Жизненный цикл сервлета.

Ответ

Методы сервлета

Для обработки запроса в HttpServlet определен ряд методов, которые мы можем переопределить в классе сервлета:

- **doGet:** обрабатывает запросы GET (получение данных)
- **doPost:** обрабатывает запросы POST (отправка данных)

- **doPut**: обрабатывает запросы PUT (отправка данных для изменения)
- **doDelete**: обрабатывает запросы DELETE (удаление данных)
- **doHead**: обрабатывает запросы HEAD

Каждый метод обрабатывает определенный тип запросов HTTP, и мы можем определить все эти методы, но, зачастую, работа идет в основном с методами **doGet** и **doPost**.

Все методы в качестве параметра принимают два объекта: **HttpServletRequest** - хранит информацию о запросе и **HttpServletResponse** - управляет ответом на запрос.

Жизненный цикл сервлета

Для каждого сервлета движок сервлетов создает только одну копию. Вне зависимости от того, сколько запросов будет отправлено сервлету, все запросы будут обрабатываться только одной копией сервлета. Объект сервлета создается либо при запуске движка сервлетов, либо, когда сервлет получает первый запрос. Затем для каждого запроса запускается поток, который обращается к объекту сервлета.

При работе с сервлетом движок сервлетов вызывает у класса сервлета ряд методов, которые определены в родительском абстрактном классе **HttpServlet**.

- a) Когда движок сервлетов создает объект сервлета, у сервлета вызывается метод **init()**.

```
public void init(ServletConfig config) throws ServletException {}
```

Этот метод вызывается только один раз - при создании сервлета. Мы можем переопределить этот метод, чтобы определить в нем некоторую логику инициализации.

- b) Когда к сервлету приходит запрос, движок сервлетов вызывает метод **service()** сервлета. А этот метод, исходя из типа запроса (GET, POST, PUT и т.д.) решает, какому методу сервлета (**doGet**, **doPost** и т.д.) обрабатывать этот запрос.

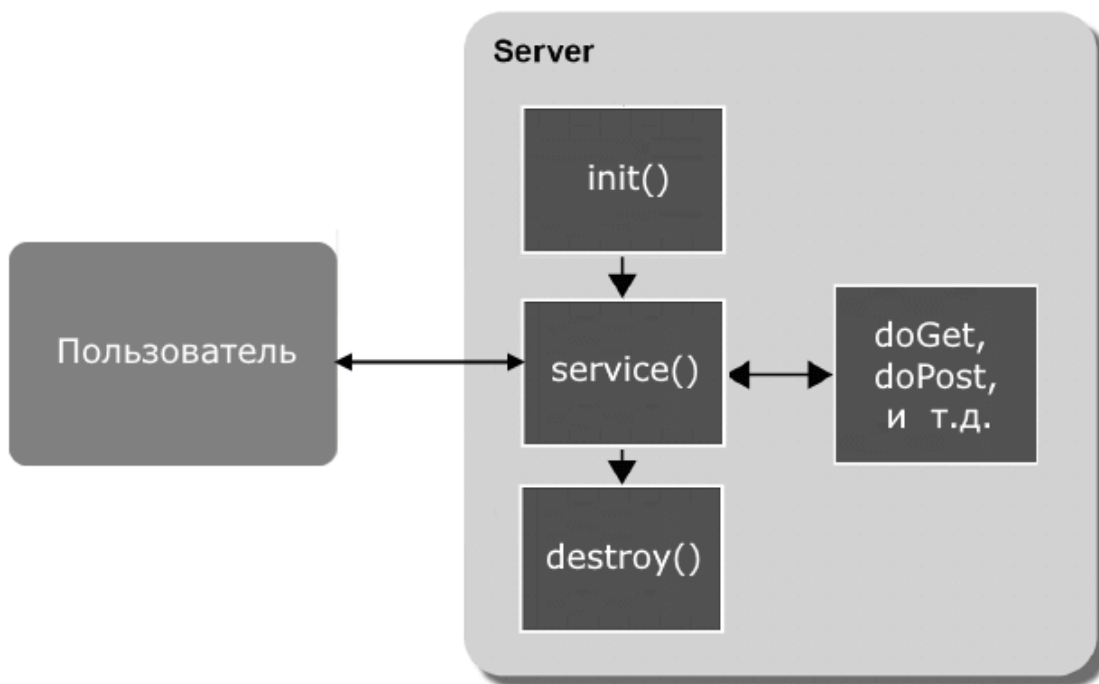
```
public void service(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {}
```

Этот метод также можно переопределить, однако в этом нет смысла. В реальности для обработки запроса переопределяются методы **doGet**, **doPost** и т.д., которые обрабатывают конкретные типы запросов.

- c) Если объект сервлета долгое время не используется (к нему нет никаких запросов), или если происходит завершение работы движка сервлетов, то движок сервлетов выгружает из памяти все созданные экземпляры сервлетов. Однако до выгрузки сервлета из памяти у сервлета вызывается метод **destroy()**.

```
public void destroy() {}
```

При необходимости мы также можем его переопределить, например, определить в нем логику логгирования или что иное. В то же время следует учитывать, что, если сервер вдруг упадет по какой-то причине, например, отключится электричество и т.д., тогда данный метод естественно не будет вызван и его логика не сработает.



Поскольку для обработки всех запросов создается один экземпляр сервлета, и все обращения к нему идут в отдельных потоках, то не рекомендуется в классе сервлета объявлять и использовать глобальные переменные, так как они не будут потокобезопасными.

123. Что такое jsp. Жизненный цикл.

Ответ

Что такое JSP

Java Server Pages (JSP) — это технология, которая используется для разработки веб-страниц путем вставки Java-кода в HTML-страницы путем создания специальных тегов JSP.

Жизненный цикл JSP

Жизненный цикл JSP определяется как перевод Страницы JSP в сервлет, так как Страница JSP должна быть сначала преобразована в сервлет для обработки запросов на обслуживание.

Жизненный цикл начинается с создания JSP и заканчивается распадом этого.

Фазы жизненного цикла JSP

Когда браузер запрашивает JSP, механизм JSP сначала проверяет, нужно ли ему компилировать страницу. Если JSP последний раз компилируется или в JSP выполняется последняя модификация, то JSP-движок компилирует страницу.

Процесс компиляции JSP-страницы включает три этапа:

- Разбор JSP
- Превращение JSP в сервлет
- Компиляция сервлета

Жизненный цикл JSP:

- Перевод страницы JSP;**
 - В Java сервлет файл создается из исходного файла JSP. Это первый шаг жизненного цикла JSP. На этапе перевода контейнер проверяет синтаксическую правильность страницы JSP и файлов тегов.
- Компиляция страницы JSP (Компиляция страницы JSP в `_jsp.java`);**
 - Сгенерированный файл сервлета Java скомпилирован в класс сервлета Java
- Загрузка классов (`_jsp.java` преобразуется в файл класса `_jsp.class`);**
 - Класс сервлета, который был загружен из источника JSP, теперь загружен в контейнер.
- Instantiation (Объект сгенерированного сервлета создан);**
- Инициализация (метод `_jspinit()` вызывается контейнером);**
 - Метод `_jspinit()` иницирует экземпляр сервлета, который был сгенерирован из JSP, и будет вызываться контейнером на этом этапе.
 - Как только экземпляр будет создан, метод `init` будет вызван сразу после этого.
 - Он вызывается только один раз в течение жизненного цикла JSP, метод инициализации объявлен, как показано выше.
- Обработка запросов (метод `_jspservice()` вызывается контейнером);**
 - Метод `_jspservice()` вызывается контейнером для всех запросов, вызванных страницей JSP в течение ее жизненного цикла.
 - Для этого этапа он должен пройти все вышеуказанные этапы, и тогда может быть вызван только метод обслуживания.
 - Он передает объекты запроса и ответа.
 - Этот метод не может быть переопределен
 - Метод показан выше: он отвечает за генерацию всех HTTP-методов GET, POST и т. д.
- Destroy (метод `_jspDestroy()`, вызываемый контейнером);**
 - Метод `_jspdestroy()` также вызывается контейнером
 - Этот метод вызывается, когда контейнер решает, что ему больше не нужен экземпляр сервлета для обслуживания запросов.
 - Когда вызывается метод уничтожения, сервлет готов к сборке мусора.
 - Это конец жизненного цикла.

- Мы можем переопределить метод `_jspdestroy()`, когда выполняем любую очистку, такую как освобождение соединений с базой данных или закрытие открытых файлов.

124. Что такое сессия? Жизненный цикл.

Ответ

Что такое сессия

Сессия — это сеанс между клиентом и сервером, устанавливаемая на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и веб-сервером в момент получения первого запроса к веб-приложению. Каждый клиент устанавливает с сервером свой собственный сеанс, который сохраняется до окончания работы с приложением.

Сессия позволяет сохранять некоторую информацию на время сеанса. Когда клиент обращается к сервлету или странице JSP, то движок сервлетов проверяет, определен ли в запросе параметр ID сессии. Если такой параметр не определен (например, клиент первый раз обращается к приложению), тогда движок сервлетов создает уникальное значение ID и связанный с ним объект сессии. Объект сессии сохраняется на сервере, а ID отправляется в ответе клиенту и по умолчанию сохраняется на клиенте в куках. Затем, когда приходит новый запрос от того же клиента, то движок сервлетов опять же может получить ID и сопоставить его с объектом сессии на веб-сервере.

По умолчанию ID сессии хранится в куках, но возможна ситуация, когда куки отключены на клиенте. Для решения этой проблемы есть ряд техник, в частности, добавление ID в адрес.

Для получения объекта сессии в сервлете у объекта `HttpServletRequest` определен метод `getSession()`. Он возвращает объект `HttpSession`.

```
HttpSession session = request.getSession();
```

Для управления сессией объект `HttpSession` предоставляет ряд методов:

- `setAttribute(String name, Object o)`: сохраняет объект в сессии под ключом `name`.
- `getAttribute(String name)`: возвращает из сессии объект с ключом `name`. Если ключа `name` в сессии неопределено, то возвращается **null**.
- `removeAttribute(String name)`: удаляет из сессии объект с ключом `name`.
- `getAttributeNames()`: возвращает объект `java.util.Enumeration`, который содержит все ключи имеющих в сессии объектов.
- `getId()`: возвращает идентификатор сессии в виде строки.
- `isNew()`: возвращает `true`, если для клиента еще не установлена сессия (клиент сделал первый запрос или на клиенте отключены куки).
- `setMaxInactiveInterval(int seconds)`: устанавливает интервал неактивности в секундах. И если в течение этого интервала клиент был неактивен, то данные сессии удаляются. По умолчанию максимальный интервал неактивности 1800 секунд. Значение `-1` указывает, что сессия удаляется только тогда, когда пользователь закрыл вкладку в браузере.
- `invalidate()`: удаляет из сессии все объекты.

Жизненный цикл сессии

Жизненный цикл сессии начинается при первом запросе пользователя к серверу. Завершается он по истечению времени бездействия пользователя или при вызове метода **`invalidate()`**.

125. Что такое request и response? Из чего состоит? Жизненный цикл.

Ответ

Что такое request и response?

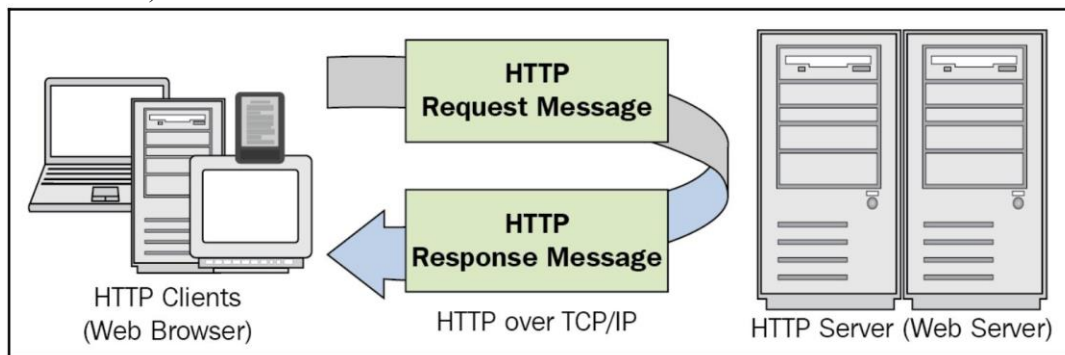
Данные между клиентом и сервером в рамках работы протокола передаются с помощью HTTP-сообщений. Они бывают двух видов:

- **Запросы (HTTP Requests)** — сообщения, которые отправляются клиентом на сервер, чтобы вызвать выполнение некоторых действий. Зачастую для получения доступа к определенному ресурсу. Основой запроса является HTTP-заголовок.
- **Ответы (HTTP Responses)** — сообщения, которые сервер отправляет в ответ на клиентский запрос.

Из чего состоит request?

Протокол связи HTTP (HyperText Transfer Protocol) — был разработан специально для передачи данных и просмотра интернет-страниц. Каждое взаимодействие по HTTP выглядит как общение между пользователем и сервером: человек с помощью клиента делает запрос (HTTP Requests) к серверу, а сервер даёт ответ (HTTP Response) — допустим, отображает сайт в браузере.

Фактически запросы представляют собой некие информационные сообщения, которые клиент передаёт на сервер с целью выполнения какого-либо действия. У каждого HTTP Requests должен быть задан URL-адрес и метод, а основой выступает хедер (HTTP-заголовок).



HTTP-запросы наполнены несколькими текстовыми строками, которые закодированы в ASCII. Версии HTTP/1.1 и ранее предполагали использование данных в виде обычного текста. В более совершенной версии HTTP/2 текстовую информацию стали разделять на фреймы.

HTTP-запросы могут выглядеть следующим образом: *GET https://www.cloud4y.ru/ HTTP/1.1*

Здесь можно видеть, что в качестве метода используется GET, обращение идёт к ресурсу <https://www.cloud4y.ru>, а в качестве протокола используется версия HTTP/1.1.

Для большей наглядности изобразим это так:



Традиционно запросы обладают структурой следующего вида:

а) Стартовая строка

Необходима для описания запроса или статуса. В ней сообщается версия протокола и другие сведения. Это может быть запрашиваемый ресурс или код ответа (например, ошибки). Длина — ровно одна строка.

Стартовая строка HTTP-запроса включает в себя три основных компонента:

- **метод запроса**

Метод — короткое слово, которое указывает, что требуется сделать с запрашиваемым ресурсом. Чаще всего встречаются методы глагольной формы (GET, PUT или POST) или существительные (HEAD, OPTIONS). GET, например, даёт серверу понять, что пользователю требуются некие данные, а POST означает, что пользователь отправляет данные на сервер.

Описание методов подробнее:

Метод	Описание
GET	Позволяет запросить некоторый конкретный ресурс. Дополнительные данные могут быть переданы через строку запроса (Query String) в составе URL (например ?param=value).
POST	Позволяет отправить данные на сервер. Поддерживает отправку различных типов файлов, среди которых текст, PDF-документы и другие типы данных в двоичном виде. Обычно метод POST используется при отправке информации (например, заполненной формы логина) и загрузке данных на веб-сайт, таких как изображения и документы.
HEAD	Здесь придется забежать немного вперед и сказать, что обычно сервер в ответ на запрос возвращает заголовок и тело, в котором содержится запрашиваемый ресурс. Данный метод при использовании его в запросе позволит получить только заголовки, которые сервер бы вернул при получении GET-запроса к тому же ресурсу. Запрос с использованием данного метода обычно производится для того, чтобы узнать размер запрашиваемого ресурса перед его загрузкой.
PUT	Используется для создания (размещения) новых ресурсов на сервере. Если на сервере данный метод разрешен без надлежащего контроля, то это может привести к серьезным проблемам безопасности.
DELETE	Позволяет удалить существующие ресурсы на сервере. Если использование данного метода настроено некорректно, то это может привести к атаке типа «Отказ в обслуживании» (Denial of Service, DoS) из-за удаления критически важных файлов сервера.
OPTIONS	Позволяет запросить информацию о сервере, в том числе информацию о допускаемых к использованию на сервере HTTP-методах.
PATCH	Позволяет внести частичные изменения в указанный ресурс по указанному расположению.

- **его цель**

Цель запроса фактически является указателем нужного пользователю URL. Состоит из протокола, URL или IP-адреса, пути к хранящемуся на сервере ресурсу. Также в состав цели запроса могут входить указание порта, параметры запроса.

- **версию протокола передачи данных.**

Версия используемого протокола передачи данных. Сейчас распространены версии HTTP/1.1 и HTTP/2. От протокола зависит структура данных, идущих вслед за стартовой строкой.

б) HTTP-заголовок.

HTTP-заголовок представляет собой строку формата «Имя-Заголовок:Значение», с двоеточием(:) в качестве разделителя. Название заголовка не учитывает регистр, то есть между Host и host, с точки зрения HTTP, нет никакой разницы. Однако в названиях заголовков принято начинать каждое новое слово с заглавной буквы. Структура значения зависит от конкретного заголовка. Несмотря на то, что заголовок вместе со значениями может быть достаточно длинным, занимает он всего одну строку.

В запросах может передаваться большое число различных заголовков, но все их можно разделить на три категории:

- **Общего назначения**, которые применяются ко всему сообщению целиком.
- **Заголовки запроса** уточняют некоторую информацию о запросе, сообщая дополнительный контекст или ограничивая его некоторыми логическими условиями.
- **Заголовки представления**, которые описывают формат данных сообщения и используемую кодировку. Добавляются к запросу только в тех случаях, когда с ним передается некоторое тело.

Самые частые заголовки запроса:

Заголовок	Описание
Host	Используется для указания того, с какого конкретно хоста запрашивается ресурс. В качестве возможных значений могут использоваться как доменные имена, так и IP-адреса. На одном HTTP-сервере может быть размещено несколько различных веб-сайтов. Для обращения к какому-то конкретному требуется данный заголовок.
User-Agent	Заголовок используется для описания клиента, который запрашивает ресурс. Он содержит достаточно много информации о пользовательском окружении. Например, может указать, какой браузер используется в качестве клиента, его версию, а также операционную систему, на которой этот клиент работает.
Refer	Используется для указания того, откуда поступил текущий запрос. Например, если вы решите перейти по какой-нибудь ссылке в этой статье, то вероятнее всего к запросу будет добавлен заголовок Refer: https://selectel.ru
Accept	Позволяет указать, какой тип медиафайлов принимает клиент. В данном заголовке могут быть указаны несколько типов, перечисленные через запятую (‘ , ’). А для указания того, что клиент принимает любые типы, используется следующая последовательность — */*.
Cookie	Данный заголовок может содержать в себе одну или несколько пар «Куки-Значение» в формате cookie=value. Куки представляют собой небольшие фрагменты данных, которые хранятся как на стороне клиента, так и на сервере, и выступают в качестве идентификатора. Куки передаются вместе с запросом для поддержания доступа клиента к ресурсу. Помимо этого, куки могут использоваться и для других целей, таких как хранение пользовательских предпочтений на сайте и отслеживание клиентской сессии. Несколько кук в одном заголовке могут быть перечислены с помощью символа точка с запятой (‘ ; ’), который используется как разделитель.
Authorization	Используется в качестве еще одного метода идентификации клиента на сервере. После успешной идентификации сервер возвращает токен, уникальный для каждого конкретного клиента. В отличие от куки, данный токен хранится исключительно на стороне клиента и отправляется клиентом только по запросу сервера. Существует несколько типов аутентификации, конкретный метод определяется тем веб-сервером или веб-приложением, к которому клиент обращается за ресурсом.

с) Пустая строка.

Она необходима, чтобы сообщить об успешной отправке метаданных конкретного запроса.

На самом деле эта строка не пуста, она содержит параметр CRLF, обозначающий конец заголовка.

d) Тело запроса.

В нём хранится информация о запросе или документ, который отправляется в ответе на запрос.

Тело HTTP-запроса может быть не у всех запросов. Например, для методов GET, HEAD, DELETE, OPTIONS тело, как правило, не нужно. А вот для методов типа POST оно необходимо — чтобы отправлять на сервер информацию для обновления, например.

Тело запроса условно делится на две категории:

- **одноресурсное**

К данной категории относятся тела запроса, состоящие из одного обособленного файла с двумя заголовками: Content-Type и Content-Length.

- **многоресурсное**

Многоресурсные же тела состоят из нескольких частей, каждая из которых содержит свой бит информации. Эти части связаны с HTML-формами.

Из чего состоит response?

HTTP-ответ является сообщением, которое сервер отправляет клиенту в ответ на его запрос. Его структура равна структуре HTTP-запроса:

а) Строка статуса (Status line)

Стартовая строка HTTP-ответа называется **строкой статуса** (status line). На ней располагаются следующие элементы:

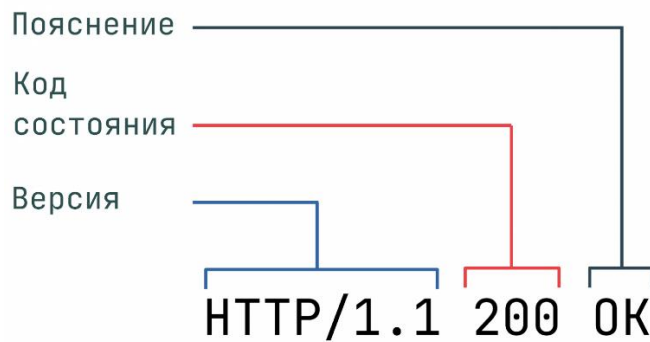
- Уже известная нам по стартовой строке запроса **версия протокола (HTTP/2 или HTTP/1.1)**.
- **Код состояния**, который указывает, насколько успешно завершилась обработка запроса.
Коды состояния HTTP используются для того, чтобы сообщить клиенту статус их запроса. HTTP-сервер может вернуть код, принадлежащий одной из пяти категорий кодов состояния:

Категория	Описание
1xx	Коды из данной категории носят исключительно информативный характер и никак не влияют на обработку запроса.
2xx	Коды состояния из этой категории возвращаются в случае успешной обработки клиентского запроса.
3xx	Эта категория содержит коды, которые возвращаются, если серверу нужно перенаправить клиента.
4xx	Коды данной категории означают, что на стороне клиента был отправлен некорректный запрос. Например, клиент в запросе указал не поддерживаемый метод или обратился к ресурсу, к которому у него нет доступа.
5xx	Ответ с кодами из этой категории приходит, если на стороне сервера возникла ошибка.

Полный список кодов состояния доступен в спецификации к протоколу, ниже приведены только самые распространенные коды ответов:

Категория	Описание
200 OK	Возвращается в случае успешной обработки запроса, при этом тело ответа обычно содержит запрошенный ресурс.
302 Found	Перенаправляет клиента на другой URL. Например, данный код может прийти, если клиент успешно прошел процедуру аутентификации и теперь может перейти на страницу своей учетной записи.
400 Bad Request	Данный код можно увидеть, если запрос был сформирован с ошибками. Например, в нем отсутствовали символы завершения строки.
403 Forbidden	Означает, что клиент не обладает достаточными правами доступа к запрошенному ресурсу. Также данный код можно встретить, если сервер обнаружил вредоносные данные, отправленные клиентом в запросе.
404 Not Found	Каждый из нас, так или иначе, сталкивался с этим кодом ошибки. Данный код можно увидеть, если запросить у сервера ресурс, которого не существует на сервере.
500 Internal Error	Данный код возвращается сервером, когда он не может по определенным причинам обработать запрос.

- **Пояснение** — короткое текстовое описание к коду состояния. Используется исключительно для того, чтобы упростить понимание и восприятие человека при просмотре ответа.



б) Заголовки ответа

Response Headers, или заголовки ответа, используются для того, чтобы уточнить ответ, и никак не влияют на содержимое тела. Они существуют в том же формате, что и остальные заголовки, а именно «Имя-Значение» с двоеточием (:) в качестве разделителя.

Ниже приведены наиболее часто встречаемые в ответах заголовки:

Категория	Пример	Описание
Server	Server: nginx	Содержит информацию о сервере, который обработал запрос.
Set-Cookie	Set-Cookie:PHPSSID=bf42938f	Содержит куки, требуемые для идентификации клиента. Браузер парсит куки и сохраняет их в своем хранилище для дальнейших запросов.
WWW-Authenticate	WWW-Authenticate: BASIC realm=»localhost»	Уведомляет клиента о типе аутентификации, который необходим для доступа к запрашиваемому ресурсу.

с) Тело ответа

Последней частью ответа является его тело. Несмотря на то, что у большинства ответов тело присутствует, оно не является обязательным. Например, у кодов «201 Created» или «204 No Content» тело отсутствует, так как достаточную информацию для ответа на запрос они передают в заголовке.

Жизненный цикл от отправки запроса до получения ответа браузером?

Жизненный цикл HTTP-запроса представляет собой последовательность этапов, которые происходят при отправке HTTP-запроса от браузера к серверу и получении ответа.

Этапы жизненного цикла:

а) Разрешение DNS:

- Браузер извлекает доменное имя из введенного URL-адреса.
- Браузер проверяет свое внутреннее хранилище (кэш) для получения соответствующего IP-адреса. Если IP-адрес не найден в кэше, браузер отправляет запрос на разрешение доменного имени (DNS resolution).
- Запрос на разрешение доменного имени отправляется к DNS-серверу, чтобы получить соответствующий IP-адрес, связанный с доменным именем.
- Если DNS-сервер имеет запись для запрашиваемого доменного имени, он отправляет IP-адрес обратно в браузер.

б) Установление соединения:

- Браузер использует полученный IP-адрес для установления TCP-соединения с сервером.
- Происходит трехэтапное рукопожатие (TCP three-way handshake) между браузером и сервером:
 - Браузер отправляет пакет синхронизации (SYN) серверу.
 - Сервер отправляет пакет подтверждения синхронизации (SYN-ACK) браузеру.
 - Браузер отправляет пакет подтверждения (ACK) серверу.
- После завершения рукопожатия устанавливается стабильное TCP-соединение между браузером и сервером, которое будет использоваться для передачи данных.

с) Отправка HTTP-запроса:

- Браузер формирует HTTP-запрос, который содержит метод (GET, POST, PUT и другие), путь к запрашиваемому ресурсу (URI), версию протокола HTTP и другие заголовки.
 - Запрос может также содержать тело, в случае POST-запроса, когда данные отправляются на сервер.
 - Браузер отправляет сформированный HTTP-запрос по установленному TCP-соединению на сервер.
- d) **Обработка запроса на сервере:**
- Сервер получает HTTP-запрос от браузера.
 - Сервер анализирует метод запроса, путь к запрашиваемому ресурсу и другие заголовки, чтобы определить, какой обработчик должен обрабатывать запрос.
 - Обработчик сервера выполняет необходимые операции, такие как поиск или генерация данных, взаимодействие с базой данных, выполнение бизнес-логики и т.д.
- e) **Формирование и отправка HTTP-ответа:**
- Сервер формирует HTTP-ответ, который содержит код состояния (например, 200 OK для успешного запроса), заголовки (Content-Type, Content-Length и др.) и тело ответа (если есть).
 - Тело ответа может содержать HTML-код, изображения, JSON-данные и другие данные, в зависимости от запроса.
 - Сформированный HTTP-ответ отправляется обратно в браузер через установленное TCP-соединение.
- f) **Получение и обработка HTTP-ответа:**
- Браузер получает HTTP-ответ от сервера.
 - Браузер проверяет код состояния ответа для определения успешности запроса или наличия ошибок (например, 404 Not Found).
 - Браузер анализирует загл. Рендеринг и отображение ответа:
 - Если ответ содержит HTML-код, браузер начинает процесс рендеринга страницы.
 - Браузер интерпретирует HTML-код и создает DOM (Document Object Model) - дерево объектов, представляющее структуру страницы.
 - CSS-стили, указанные в ответе или связанные с документом, применяются к элементам DOM, что позволяет задать внешний вид и расположение элементов на странице.
 - JavaScript-код, если есть, выполняется, обеспечивая дополнительную интерактивность и функциональность на странице.
 - Браузер отображает контент на экране пользователя, включая текст, изображения, формы, ссылки и другие элементы, соответствующие заданным стилям.
- g) **Заккрытие соединения:**
- После завершения обработки и отображения ответа, браузер может закрыть TCP-соединение с сервером.
 - Однако, соединение может быть оставлено открытым для повторного использования, если на странице присутствуют дополнительные ресурсы, такие как изображения или скрипты, которые должны быть загружены.

126. Cookies. Как можно достать Cookies, а если Cookies удалить, можно ли достать сессию?

Ответ

Что такое Cookie?

Cookie – это текстовый файлы, которые хранятся на локальной машине клиента, содержащий различную информацию. Она используется для отслеживания различных данных.

Куки представляют простейший способ хранения данных приложения. Куки хранятся в браузере пользователя в виде пары ключ-значение: с каждым уникальным ключом сопоставляется определенное значение. По ключу мы можем получить сохраненное в куках значение.

Приложение на сервере может устанавливать куки и отправлять в ответе пользователю, после чего куки сохраняются в браузере. Когда клиент отправляет запрос к приложению, то в запросе также отправляются и те куки, которые установлены данным приложением.

Куки могут быть двух типов:

- **Одни куки хранятся только в течении сеанса;**

То есть когда пользователь закрывает вкладку браузера и прекращает работать с приложением, то куки сеанса уничтожаются.

- **Второй тип куков - постоянные куки;**

Хранятся в течение продолжительного времени (до 3 лет).

Работа с Cookie

Для работы с куками сервлеты могут использовать класс `javax.servlet.http.Cookie` / `jakarta.servlet.http.Cookie`.

Для создания куки надо создать объект этого класса с помощью конструктора `Cookie(String name, String value)`, где `name` - ключ, а `value` - значение, которое сохраняется в куках. Стоит отметить, что мы можем сохранить в куках только строки.

- **Метод добавление Cookie**

Чтобы добавить куки в ответ клиенту у объекта `HttpServletResponse` применяется метод **`addCookie(Cookie cookie)`**.

```
response.addCookie(new Cookie("name", "value"));
```

- **Методы установки и получения параметров;**

При создании куки можно использовать ряд методов объекта `Cookie` для установки и получения отдельных параметров:

```
Cookie cookie = new Cookie("name", "value");
```

Метод	Пример	Описание
<code>public void setMaxAge(int expiry)</code>	<code>cookie.setMaxAge(60)</code>	Устанавливает время в секундах, в течение которого будут существовать куки. Специальное значение <code>-1</code> указывает, что куки будут существовать только в течение сессии и после закрытия браузера будут удалены.
<code>public int getMaxAge()</code>	<code>cookie.getMaxAge()</code>	Возвращает время хранения кук.
<code>public void setValue(String newValue)</code>	<code>cookie.setValue("newValue")</code>	Устанавливает хранимое значение.
<code>public String getValue()</code>	<code>cookie.getValue()</code>	Возвращает значение кук.
<code>public String getName()</code>	<code>cookie.getName()</code>	Возвращает имя cookie.
<code>public void setDomain(String pattern)</code>	<code>cookie.setDomain("Domain")</code>	Устанавливает домен запроса, например, <code>proselyte.net/</code>
<code>public String getDomain()</code>	<code>cookie.getDomain()</code>	Возвращает домен запроса, например, <code>proselyte.net</code>
<code>public void setPath(String uri)</code>	<code>cookie.setPath("uri")</code>	Устанавливает путь cookie.
<code>public String getPath()</code>	<code>cookie.getPath()</code>	Возвращает путь cookie.
<code>public void setSecure(boolean flag)</code>	<code>cookie.setSecure(true)</code>	Указывает, должны ли cookie передаваться только по защищённому соединению.

- **Получение всех Cookies**

Чтобы получить куки, которые приходят в запросе от клиента, применяется метод **`getCookies()`** класса `HttpServletRequest`.

```
Cookie[] cookies = request.getCookies();
```

- **Получение Cookie по названию**

Получение куки по имени немного громоздко, поскольку нам надо перебрать набор полученных кук и сравнить их с нужным ключом. Поэтому при частном использовании, как правило, определять вспомогательные методы, которые инкапсулируют подобную функциональность.

- **Удаление Cookie**

Cookie существуют по не истечёт время действия или пока работает браузер / сессия. Чтобы удалить Cookie можно задать методом `setMaxAge(int expiry)` задать 0 секунд, и он удалится.

```
cookie.setMaxAge(0);
```

Cookie Vs. сессия

Cookie	Session (сессия)
<ul style="list-style-type: none"> • Cookies – это файлы на стороне клиента, которые содержат информацию о пользователе 	<ul style="list-style-type: none"> • Сессии – это файлы на стороне сервера, которые содержат информацию о пользователе
<ul style="list-style-type: none"> • Cookie-файл заканчивается в зависимости от срока, установленного для него 	<ul style="list-style-type: none"> • Сессия заканчивается, когда пользователь закрывает свой браузер
<ul style="list-style-type: none"> • Официальный максимальный размер файла cookie составляет 4 КБ. 	<ul style="list-style-type: none"> • В течение сеанса вы можете хранить столько данных, сколько захотите. Единственное ограничение, которое вы можете достичь, – это максимальный объем памяти, который скрипт может использовать за один раз, по умолчанию 128 МБ.
<ul style="list-style-type: none"> • Файл cookie не зависит от сеанса 	<ul style="list-style-type: none"> • Сеанс зависит от Cookie

127. Что нужно написать в браузерной строке, чтобы обратиться к сервлету? Можно ли из браузерной строки напрямую вызвать метод сервлета?

Ответ

`http://localhost:8080/web_app_market/ServletTest`

128. Чем отличается методы POST и GET. Если не указать напрямую, какой из этих методов выполнится по умолчанию?

Ответ

Отличия GET и POST запросов

Запрос GET передает данные в URL в виде пар "имя-значение" (другими словами, через ссылку), а **запрос POST передает данные в теле запроса** (подробно показано в примерах ниже). Это различие определяет свойства методов и ситуации, подходящие для использования того или иного HTTP метода.

Страница, созданная методом GET, может быть открыта повторно множество раз. Такая страница может быть кэширована браузерами, проиндексирована поисковыми системами и добавлена в закладки пользователем. Из этого следует, что метод GET следует использовать для получения данных от сервера и не желательно в запросах, предполагающих внесения изменений в ресурс.

Запрос, выполненный методом POST, напротив следует использовать в случаях, когда нужно вносить изменение в ресурс (выполнить авторизацию, отправить форму оформления заказа, форму обратной связи, форму онлайн заявки). Повторный переход по конечной ссылке не вызовет повторную обработку запроса, так как не будет содержать переданных ранее параметров. Метод POST имеет большую степень защиты данных, чем GET: параметры запроса не видны пользователю без использования специального ПО, что дает методу преимущество при пересылке конфиденциальных данных, например, в формах авторизации.

HTTP метод POST поддерживает тип кодирования данных *multipart/form-data*, что позволяет передавать файлы.

По умолчанию производится запрос GET.

Сравнительная таблица HTTP методов GET и POST

Свойство	GET	POST
Способ передачи данных	Через URL	В теле HTTP запроса
Защита данных	Данные видны всем в адресной строке браузера, истории браузера и т.п.	Данные можно увидеть только с помощью инструментов разработчика, расширений браузера, специализированных программ.
Длина запроса	Не более 2048 символов	Не ограничена <i>Примечание: ограничения могут быть установлены сервером.</i>
Сохранение в закладки	Страница с параметрами может быть добавлена в закладки	Страница с параметрами не может быть добавлена в закладки.
Кэширование	Страница с параметрами может быть кэширована	Страница с параметрами не может быть кэширована
Индексирование поисковыми системами	Страница с параметрами может быть индексируется	Страница с параметрами не может быть индексируется
Возможность отправки файлов	Не поддерживается	Поддерживается
Поддерживаемые типы кодирования	application/x-www-form-urlencoded	application/x-www-form-urlencoded multipart/form-data text/plain
Использование в гиперссылках <a>	Да	Нет
Использование в HTML формах	Да	Да
Использование в AJAX запросах	Да	Да

129. Как сделать redirect незаметно для пользователя?

Ответ

Никак, пользователь увидит изменение URL в строке. /делать редирект на ту же страницу

130. Какие scopes (области видимости) переменных существуют в JSP?

Ответ

Область видимости объектов определяется тем контекстом, в который помещается данный объект. В зависимости от той или иной области действия так же определяется время существования объекта.

В JSP предусмотрены следующие области действия переменных (объектов):

- **request** — область действия запроса;

При использовании области действия запроса **request** объект будет доступен на текущей JSP странице, странице пересылки (при использовании `jsp:forward`) или на включаемой странице (при использовании `jsp:include`).

- **session** — область действия сессии;

В случае области действия сессии **session**, объект будет помещен в сеанс пользователя. Бин будет доступен на всех JSP страницах и будет существовать пока существует сессия пользователя, или он не будет из нее принудительно удален.

- **application** — область действия приложения;

Бин с областью действия **application** доступен для всех пользователей на всех JSP страницах и существует на протяжении всей работы приложения или пока не будет удален принудительно и контекста приложения.

- **page** — область действия страницы.

При использовании области действия **page** объект доступен только на той странице, где он определен. На включаемых (`jsp:include`) и переадресуемых (`jsp:forward`) страницах данный объект уже не доступен.

Таким образом, чтобы бин был доступен всем JSP страницам, необходимо указать область видимости **application** или **session**, в зависимости от того требуется ли доступ к объекту всем пользователям или только текущему. Для указания требуемой области действия при определении объекта на JSP странице используется атрибут **scope** тега **jsp:useBean**:

```
<jsp:useBean id="myBean" class="ru.javacore.MyBean" scope="session"/>
```

Отметит, что если не указывать атрибут **scope**, то по умолчанию задается область видимости страницы **page**.

131. В чём различие forward и redirect?

Ответ

Иногда первоначальный обработчик HTTP-запросов в нашем сервлете Java должен делегировать запрос другому ресурсу. В этих случаях мы можем либо перенаправить запрос дальше (**forward**), либо перенаправить его на другой ресурс (**redirect**).

Перенаправление (forward) запроса

Метод **forward()** класса **RequestDispatcher** позволяет перенаправить запрос из сервлета на другой сервлет, html-страницу или страницу jsp.

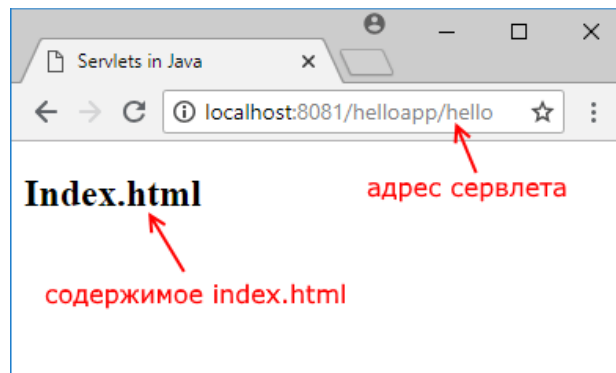
Для того, чтобы выполнить перенаправление запроса надо с помощью его метода `getRequestDispatcher()` получаем объект **RequestDispatcher**. Путь к ресурсу, на который надо выполнить перенаправление, передается в качестве параметра в `getRequestDispatcher`.

```
String path = "/index.jsp";
RequestDispatcher requestDispatcher = request.getRequestDispatcher(path);
```

Затем у объекта **RequestDispatcher** вызывается метод `forward()`, в который передаются объекты **HttpServletRequest** и **HttpServletResponse**.

```
requestDispatcher.forward(request, response);
```

И если мы обратимся к сервлету (который перенаправляет запрос), то фактически мы получим содержимое страницы `index.html`, который будет перенаправлен запрос.



Подобным образом можно выполнять перенаправление на страницы jsp и другие сервлеты.

Переадресация (redirect) запроса

Для переадресации применяется метод **sendRedirect()** объекта **HttpServletResponse**. В качестве параметра данный метод принимает адрес переадресации. Адрес может быть локальным, внутренним, а может быть и внешним.

```
response.sendRedirect("https://metanit.com/");
```

Объяснение метафорой redirect и forward

Вы обращаетесь к другу с вопросом. Друг не знает ответа на вопрос, но знаком с человеком, который знает.

Forward

Друг скажет вам: «Ок, я знаю ответ на твой вопрос!». После этого он САМ позвонил человеку, который знает ответ на вопрос, получит от него информацию и передаст эту информацию вам.

Redirect

Друг говорит вам, что не знает ответа на вопрос, но знаком с человеком, кто знает. После этого он даёт вам телефон человека, который знает ответ на вопрос. Вы совершаете звонок на этот номер.

Различие forward и redirect

Redirect	Forward
<ul style="list-style-type: none">• Redirect происходит на клиенте (браузер совершает новый запрос)	<ul style="list-style-type: none">• Forward происходит на сервере (клиент не знает про это)
<ul style="list-style-type: none">• При Redirect URL в браузере меняется	<ul style="list-style-type: none">• При Forward URL в браузере не меняется (перенаправление случается целиком на сервере, клиент об этом не знает)
<ul style="list-style-type: none">• Redirect медленнее Forward (необходимо совершить больше операций)	<ul style="list-style-type: none">• Forward быстрее Redirect
<ul style="list-style-type: none">• Адрес может быть локальным, внутренним, а может быть и внешним.	<ul style="list-style-type: none">• Адрес может быть на другой сервер, html-страницу или страницу jsp локально.

132. Отличия `getAttribute()` от `getParameter()` в сервлете.

Ответ

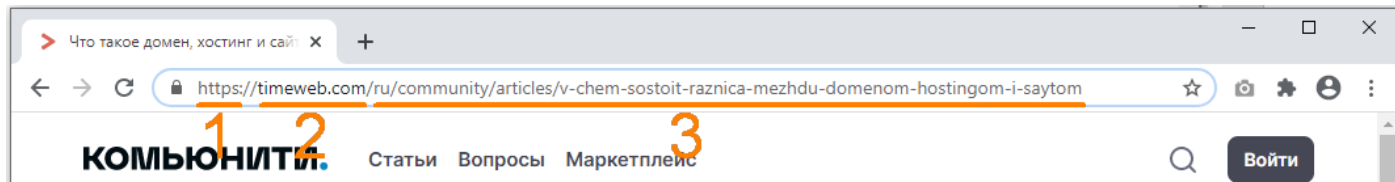
`getParameter()` – возвращает параметры `http`-запроса. Они передаются от клиента к серверу. Может только возвращать `String`.

`getAttribute()` – предназначен только для использования на стороне сервера - вы заполняете запрос атрибутами, которые вы можете использовать в рамках одного запроса. Может использоваться для любого объекта, а не только для строки.

133. Из чего состоит url?

Ответ

Структура URL адреса



а) В начале адреса (1) всегда указан протокол (в некоторых браузерах по умолчанию он может быть скрыт и становится виден при щелчке по адресной строке). Если мы просматриваем веб-страницу, это будет протокол передачи данных «`http`» или его форма «`https`» с поддержкой шифрования для установки безопасного соединения. Однако URL может начинаться с других обозначений, например:

- «`ftp`» — в этом случае браузер откроет файловый сервер. После протокола «`ftp`» может указываться логин и пароль для входа в учетную запись. Выглядеть это может следующим образом: `ftp://name:password@example.com`;
- «`mailto`» — браузер выполнит команду отправки письма на указанный адрес;
- «`file`» — в браузере будет открыт файл с компьютера.

б) После протокола (2) следует доменное имя сайта (хост) или в редких случаях его IP-адрес. Также в некоторых случаях URL-адрес может содержать номер порта, например, его можно увидеть в сетевых приложениях (выглядит это так: `//example.com:8080`).

в) Затем указывается путь к странице (3), состоящий из каталогов и подкаталогов, который, в свою очередь, включает в себя ее название.

URL также может включать параметры, которые указываются после знака «`?`» и разделяются символом «`&`». Пример адреса страницы с результатами поиска по слову «`url`» в поисковой системе Google:

https://www.google.ru/search?newwindow=1&sxsrf=ALeKk02BP8tO_kCAffUrYqQOwhLV3p_jdw%3A1605124767263&source=hp&ei=n0KsX6mcDO-grgT7tpeYDA&q=url&oq=url&gs_lcp=CgZwc3ktYWIQAzIFCAAQsQMMyBQgAELEDMgIIIjICCAAyAggAMgIIADICCAAyAggAMgIIADICCABQjghYjghgkRpoAHAAeACAAUOIAUOSAQExmAEOAECoAEBqgEHZ3dzLXdpeg&scient=psy-ab&ved=0ahUKEwjp58bco_vsAhVvkIsKHXvbBcMQ4dUDCAs&uact=5

d) Конечный компонент URL, который пользователь может увидеть в документах большого объема, состоящих из нескольких разделов, — это якорь, которому предшествует знак решетки «#». Часть адреса после этого знака ссылается на определенный абзац внутри страницы сайта. Пример: если на странице Википедии со статьей «URL» перейти по ссылке «Структура URL» в блоке «Содержание», унифицированный указатель ресурса в адресной строке браузера примет такой вид:

https://ru.wikipedia.org/wiki/URL#Структура_URL

134. Отличие `jsp:include` от директивы `include`.

Ответ

<@include> - Тег директивы инструктирует компилятор JSP объединить содержимое включенного файла в JSP, прежде чем создавать сгенерированный код сервлета. Во время выполнения выполняется только один сервлет.

<jsp:include> - Тег JSP Action, с другой стороны, указывает контейнеру приостановить выполнение этой страницы, запустить запущенную страницу и объединить вывод с этой страницы в вывод с этой страницы. Каждая включенная страница выполняется как отдельный сервлет во время выполнения.

Разница между директивой JSP `include` (**<@include>**) и `jsp:include` (**<jsp:include>**) action заключается в том, что для директивы `include`, контент для другого ресурса будет добавлен в созданный сервлет на этапе трансляции JSP (фаза Translation), в то время как `jsp:include` action работает в рантайме.

Другое отличие JSP `include` action в том, что мы можем передать параметры для вложения с помощью команды `jsp:params`, в то время как директива `jsp:include` не имеет возможности передавать параметры. Использовать директиву JSP `include` необходимо для статических ресурсов вроде header, footer, image file для повышения производительности. Если же нам необходима динамика, передача параметров для обработки, то необходимо использовать тег `jsp:include` action.

135. Применение классов `HttpServletRequestWrapper` и `HttpServletResponseWrapper`.

Ответ

В Servlet HTTP API предоставляются два класса обертки - `HttpServletRequestWrapper` и `HttpServletResponseWrapper`. Они помогают разработчикам реализовывать собственные реализации типов `request` и `response` сервлета. Мы можем расширить эти классы и переопределить только необходимые методы для реализации собственных типов объектов ответов и запросов. Эти классы не используются в стандартном программировании сервлетов.

136. Что делает `RequestDispatcher.include()`?

Ответ

Если необходимо вызывать сервлет из того же приложения, то необходимо использовать механизм внутренней коммуникации сервлетов (inter-servlet communication mechanisms). Мы можем вызвать другой сервлет с помощью `RequestDispatcher` **`forward()`** и **`include()`** методов для доступа к дополнительным атрибутам в запросе для использования в другом сервлете. Метод **`forward()`** используется для передачи обработки запроса в другой сервлет. Метод **`include()`** используется, если мы хотим вложить результат работы другого сервлета в возвращаемый ответ.

137. В какой последовательности выполняются сервлет фильтры?

Ответ

Цепочка, определяемая `url-pattern`, выстраивается в том порядке, в котором встречаются соответствующие описания фильтров в `web.xml`

138. Как обрабатывается тег с телом?

Ответ

Обработчик тега для тега с телом реализуется по-разному, в зависимости от того, должен ли обработчик тега взаимодействовать с телом или нет. Под словом взаимодействовать понимается, может ли обработчик тега изменять содержимое тела.

Обработчик тега не взаимодействует с телом

Если обработчику тега не нужно взаимодействовать с телом, он должен реализовывать интерфейс `Tag` (или порождаться от `TagSupport`). Если тело тега нужно вычислять, метод `doStartTag` должен возвращать `EVAL_BODY_INCLUDE`; в противном случае, он должен возвращать `SKIP_BODY`.

Если обработчик тега должен последовательно вычислять тело, он должен реализовывать интерфейс `IterationTag` или порождаться из `TagSupport`. Он должен возвращать `EVAL_BODY_AGAIN` из методов `doStartTag` и `doAfterBody`, если определит, что тело должно вычисляться повторно.

Обработчик тега взаимодействует с телом

Если обработчику тега нужно взаимодействовать с телом, обработчик тега должен реализовывать `BodyTag` (или порождаться из `BodyTagSupport`). Такие обработчики обычно реализуют методы `doInitBody` и `doAfterBody`. Эти методы взаимодействуют с содержимым тела, переданного в обработчик тега сервлетом JSP-страницы.

Содержимое тела поддерживает несколько методов чтения и записи своего содержимого. Обработчик тега может использовать методы содержимого тела `getString` или `getReader` для извлечения информации из тела, и метод `writeOut(out)` для записи содержимого тела в выходной поток. Объект `out`, предоставленный методу `writeOut`, берется при помощи метода обработчика тега `getPreviousOut`. Этот метод используется для проверки того, что результаты обработчика тега доступны внешнему обработчику тега.

Если тело тега должно быть вычислено, метод `doStartTag` должен вернуть `EVAL_BODY_BUFFERED`; в противном случае, он должен вернуть `SKIP_BODY`.

- **Метод `doInitBody`**

Метод `doInitBody` вызывается после установки содержимого тела, но перед тем как оно будет вычисляться. Этот метод обычно используется для выполнения любой инициализации, зависящей от содержимого тела.

- **Метод `doAfterBody`**

Метод `doAfterBody` вызывается после вычисления содержимого тела. Аналогично методу `doStartTag` метод `doAfterBody` должен вернуть признак, продолжать ли вычисление тела. То есть, если тело должно быть вычислено снова, например, в случае реализации итерационного тега, `doAfterBody` должен вернуть `EVAL_BODY_BUFFERED`; в противном случае, `doAfterBody` должен вернуть `SKIP_BODY`.

- **Метод `release`**

В методе `release` обработчик тега должен сбросить свое состояние и освободить все собственные ресурсы.

139. JSTL.

Ответ

Что такое JSTL

JSP по умолчанию позволяет встраивать код на java в разметку html. Однако иногда использование стандартных способов для ряда операций, например, для вывода на страницу элементов из списка и т.д., может быть несколько громоздким. Чтобы упростить встраивание кода java в JSP была разработана специальная библиотека - **JSTL**. **JSTL** (JSP Standard Tag Library) предоставляет теги для базовых задач JSP (цикл, условные выражения и т.д.).

Эта библиотека не является частью инфраструктуры Java EE, поэтому ее необходимо добавлять в проект самостоятельно.

Несмотря на то, что JSTL часто называется библиотекой, на самом деле она содержит ряд библиотек:

- **Core**: содержит основные теги для наиболее распространенных задач. Использует префикс "c" и uri "http://java.sun.com/jsp/jstl/core"
- **Formatting**: предоставляет теги для форматирования чисел, дат, времени. Использует префикс "fmt" и uri "http://java.sun.com/jsp/jstl/fmt"
- **SQL**: предоставляет теги для работы с sql-запросами и источниками данных. Использует префикс "sql" и uri "http://java.sun.com/jsp/jstl/sql"
- **XML**: предоставляет теги для работы с xml. Использует префикс "x" и uri "http://java.sun.com/jsp/jstl/xml"

- **Functions:** предоставляет функции для работы со строками.

Использует префикс "fn" и uri "http://java.sun.com/jsp/jstl/functions"

Для подключения функционала этих библиотек на страницу jsp применяется директива **taglib**. Например, подключения основной библиотеки:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Основные возможности JSTL

Основные возможности JSTL из библиотеки Core:

- **Цикл**

Вывод в цикле элементов массива или коллекции:

```
<c:forEach var="user" items="${users}">
    <p>${user}</p>
</c:forEach>
```

В данном случае параметр "items" указывает на коллекцию, элементы которой выводятся. А параметр "var" задает переменную, через которую доступен текущий перебираемый элемент.

- **Условные выражения**

Выражение if:

```
<c:if test="${isVisible == true}">
    <p>Visible</p>
</c:if>
```

В данном случае если атрибут isVisible равен true, то выводится код, который расположен между тегами <c:if> и </c:if>.

Если надо задать альтернативную логику, то можно добавить тег c:if, который проверяет противоположное условие:

```
<c:if test="${isVisible == true}">
    <p>Visible</p>
</c:if>
<c:if test="${isVisible == false}">
    <p>Invisible</p>
</c:if>
```

- **Тег choose**

Тег c:choose подобно конструкции switch...case в Java проверяет объект на соответствие одному из значений.

```
<c:choose>
    <c:when test="${val == 1}">
        <p>Equals to 1</p>
    </c:when>
    <c:when test="${val == 2}">
        <p>Equals to 2</p>
    </c:when>
    <c:otherwise>
        <p>Undefined</p>
    </c:otherwise>
</c:choose>
```

В данном случае тег c:choose проверяет значение атрибута "val". Для проверки применяются вложенные теги c:when, которые аналогичны блокам case в конструкции switch...case. С помощью их параметра test значение атрибута сравнивается с определенным значением. И если выражения сравнения истинно, то выводится код, который размещен внутри данного элемента c:when. Таким образом мы можем определить несколько блоков c:when. Дополнительный тег <c:otherwise> выполняется, если условия проверки значения во всех тегах c:when ложно.

- **Тег url**

Тег <c:url> позволяет создать адрес относительно корня приложения. Этот тег может применяться, например, при создании ссылок.

```
<a href='<c:url value="/edit" />'> <c:url value="/edit" /> </a>
```

Параметр value содержит часть адреса, которая добавляется к корню приложения.

- **Редирект**

С помощью тега `redirect` можно установить редирект на другой адрес. Например, в случае если атрибут `val` не определен, то делаем редирект на страницу `"notfound.jsp"`:

```
<c:if test="${val == null}">
  <c:redirect url="/notfound.jsp" />
</c:if>
```

140. Что нужно написать в строке браузера, чтобы обратиться к хосту, на котором установлен tomcat, развёрнуто приложение, в котором есть несколько сервлетов? Как обратиться к конкретному сервлету? Что такое `www`? Где нужно указывать порт?

Ответ

WWW

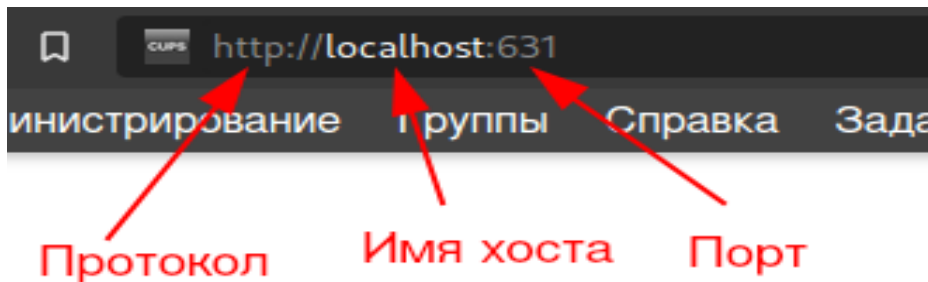
Префикс `www` – это аббревиатура, которая означает `world wide web` – или информационная система, предоставляющая доступ к текстовой информации по протоколу `http`, по-русски – всемирная паутина.

Всемирная сеть — сокращённо: `WWW`, `W3`, или `Web`; Сеть, паутина или веб — всемирная система публичных веб-страниц в сети Интернет. Сеть не является Интернетом: Сеть лишь использует Интернет как среду передачи информации и данных.

Порт

Сетевой порт – это некое виртуальное расширение, дополнение к сетевому адресу (`IP`-адресу). Для большего понимания можно сказать, что без сетевого порта информация с другого устройства вряд ли дойдет до нашего компьютера, так как она будет идти только по `IP`. Компьютер просто не поймёт, как обработать её, с помощью какого приложения.

Порт в `URL`-адресе указывается после двоеточия.



141. Можно ли в `web.xml` определить сервлет без указания `url` паттерна и как к нему обратиться?

Ответ

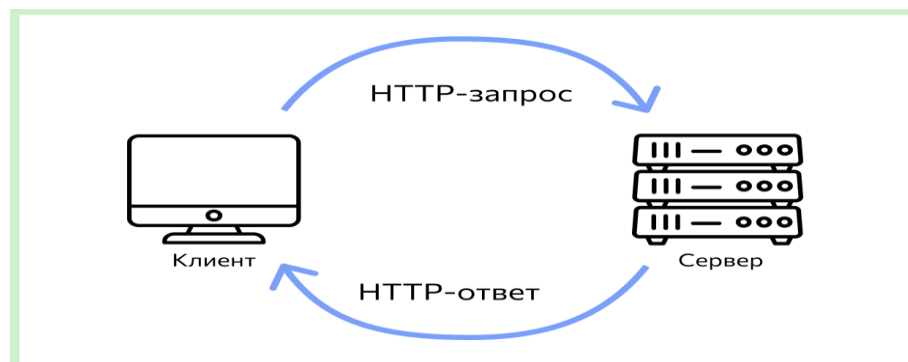
Определить можно, но при запуске программы вылетит 404 ошибка. К такому сервлету нельзя обратиться.

142. Что такое `HTTP`? Отличия `HTTP 1.0` и `HTTP 2`.

Ответ

Что такое `HTTP`

HTTP – это протокол передачи информации в интернете, который расшифровывается как «протокол передачи гипертекста» (`HyperText Transfer Protocol`). Например, браузер отправляет единичный запрос на сервер, который в свою очередь обрабатывает его, формирует ответ и делится с браузером этим ответом – ресурсами в виде данных.



Отличия HTTP 1.0 и HTTP 2

HTTP/2 стал первым бинарным протоколом. Если сравнивать его с прошлой версией протокола, то здесь разработчики поменяли методы распределения данных на фрагменты и их отправку от сервера к пользователю и наоборот. Новая версия протокола позволяет серверам доставлять информацию, которую клиент пока что не запросил. Это было внедрено с той целью, чтобы сервер сразу же отправлял браузеру для отображения документов дополнительные файлы и избавлял его от необходимости анализировать страницу и самостоятельно запрашивать недостающие файлы.

Еще одно отличие http 2.0 от версии 1.1 – мультиплексирование запросов и ответов для решения проблемы блокировки начала строки, присущей HTTP 1.1. Еще в новом протоколе можно сжимать HTTP заголовки и вводить приоритеты для запросов.

143. Сохраняет ли http протокол своё состояние.

Ответ

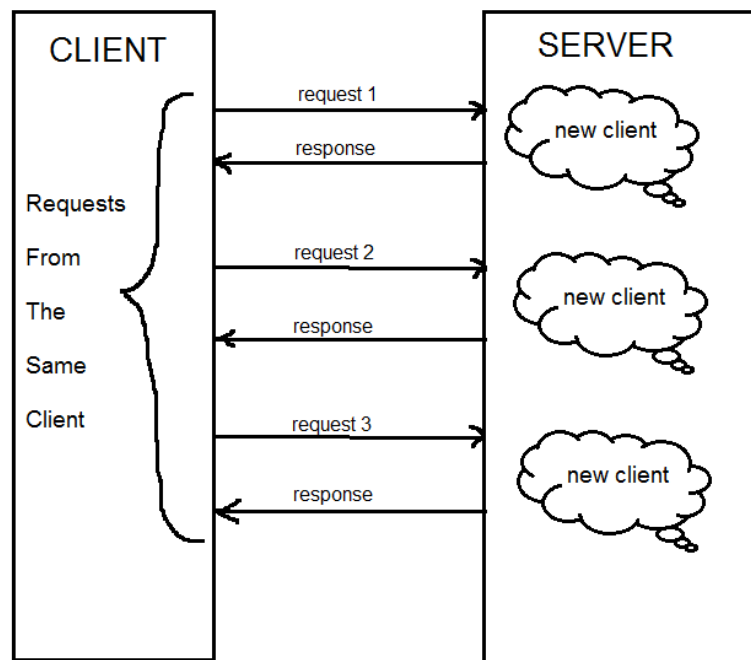
HTTP - это протокол без сохранения состояния, то есть сервер не сохраняет никаких данных (состояние) между двумя парами "запрос-ответ". Несмотря на то, что HTTP основан на TCP/IP, он также может использовать любой другой протокол транспортного уровня с гарантированной доставкой.

144. Как сервер понимает, что для пользователя создана сессия и не нужно её создавать?

Ответ

Зачем нужна сессия

Одной из основных особенностей протокола HTTP является то, что он не обязывает сервер сохранять информацию о клиенте между запросами, то есть идентифицировать клиента. Это так называемый stateless-протокол. Связь между клиентом и сервером заканчивается, как только завершается обработка текущего запроса. Каждый новый запрос к серверу подразумевается, как абсолютно уникальный и независимый, даже если он был отправлен повторно от одного и того же источника.



Один клиент отправляет запросы. Сервер думает, что это разные клиенты

Можно ли обойтись без сессии

Что, если оставить stateless-природу протокола HTTP и не идентифицировать пользователя? Без состояний сеанса можно легко обойтись, если на вашем сайте представлена статичная (обезличенная) информация, например, новостная статья, состоящая из текста и изображений. В таком контексте совершенно необязательно ассоциировать несколько запросов с одним пользователем. Ведь содержание статьи никак не изменится, будь то десять запросов с одного устройства, либо десять запросов от разных людей с разных устройств.

Но как только мы собираемся передать персональную информацию на сервер, нам необходимо каким-то образом сделать так, чтобы сервер ассоциировал все наши запросы именно с нами, и в будущем верно определял все исходящие от нас запросы. Если этого не сделать, то с каждым новым запросом мы будем вынуждены повторно передавать необходимые

персональные данные. Например, логин для входа в личный кабинет на сайте, или такую информацию как имя, адрес доставки, при совершении покупки в интернет-магазине.

Что такое сессия

Сессия (session) – это некоторый отрезок во времени, в пределах которого веб-приложение может определять все запросы от одного клиента.

Когда клиент впервые передает персональные данные в запросе, на сервере создается новая сессия для этого клиента. В период времени жизни сессии все запросы от этого клиента будут однозначно распознаны и связаны с ним. По истечении этого времени связь с клиентом будет потеряна, и очередной запрос от него будет обрабатываться как абсолютно уникальный, никак не связанный с предыдущими.

145. Отличие авторизации от аутентификации.

Ответ

Определения

- **Идентификация** – процедура, в результате выполнения которой для субъекта выявляется его уникальный признак, однозначно определяющий его в информационной системе.

Идентификация без аутентификации не работает. Потому что мало ли кто ввел существующий в системе логин! Системе обязательно надо удостовериться, что этот кто-то знает еще и пароль. Но пароль могли подсмотреть или подобрать, поэтому лучше подстраховаться и спросить что-то дополнительное, известное только данному пользователю: например, одноразовый код для подтверждения входа.

- **Аутентификация** – процедура проверки подлинности, например, проверка подлинности пользователя путем сравнения введенного им пароля с паролем, сохраненным в базе данных.

Аутентификация без предварительной идентификации лишена смысла – пока система не поймет, подлинность чего же надо проверять, совершенно бессмысленно начинать проверку. Для начала надо представиться.

- **Авторизация** – предоставление определенному лицу прав на выполнение определенных действий.

А вот **авторизация** без идентификации и аутентификации очень даже возможна. Например, в Google Документах можно публиковать документы так, чтобы они были доступны всем. В этом случае вы как владелец файла увидите сверху надпись, гласящую, что его читает неопознанный субъект. Несмотря на это, система его все же авторизовала – то есть выдала право прочитать этот документ.

Но если вы открыли этот документ для чтения только определенным пользователям, то им в таком случае сперва пришлось бы идентифицироваться (ввести свой логин), потом аутентифицироваться (ввести пароль и одноразовый код) и только потом получить право на чтение документа – авторизоваться.

Если же речь идет о содержимом вашего почтового ящика, то Google никогда и ни за что не авторизует неопознанного субъекта на чтение вашей переписки.

Примеры

Пользователь хочет войти в свой аккаунт Google (Google подходит лучше всего, потому что там процедура входа явным образом разбита на несколько простейших этапов). Вот что при этом происходит:

- Сначала система запрашивает логин, пользователь его указывает, система распознает его как существующий — это **идентификация**.

- После этого Google просит ввести пароль, пользователь его вводит, и система соглашается, что пользователь действительно настоящий, ведь пароль совпал, — это **аутентификация**.

- Возможно, Google дополнительно спросит еще и одноразовый код из SMS или приложения. Если пользователь и его правильно введет, то система окончательно согласится с тем, что он настоящий владелец аккаунта, — это **двухфакторная аутентификация**.

- После этого система предоставит пользователю право читать письма в его почтовом ящике и все остальное — это **авторизация**.

146. Сервер приложений. Веб-сервер. Отличия.

Ответ

Вопрос 120.

147. Пошагово рассказать, что происходит, когда пользователь нажимает на кнопку. (с формы поля сетаются в request, потом вызывается контейнер сервлетов, потом он как-то по request понимает,

куда нужно идти дальше (в дескриптор развертывания, а их может быть несколько, нужно как-то понимать в какой).

Ответ

С формы поля добавляются в request -> вызывается контейнер сервлетов -> оттуда вызывается нужный сервлет (на который отправился запрос – по имени сервлета) -> у меня в проекте -> из request берётся значение параметра «command» -> в enum по названию команды достаётся объект с логикой команды -> у объекта вызывается метод execute и передаются в метод request и response -> выполняется логика метода -> возвращается объект router в котором указывается путь по которому нужно перейти и тип (forward / redirect) с помощью которого нужно перейти по данному пути -> переходим по url.

148. Какие бывают Webservice?

Ответ

Что такое веб-сервисы?

Веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.

С появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

Протоколы веб-сервисов

На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

- **SOAP** (Simple Object Access Protocol) — по сути это тройка стандартов SOAP/WSDL/UDDI
 - Он содержит такие направления как XML, WSDL, SOAP – это спецификации на основе которых строятся веб сервисы.
 - Можно взаимодействовать с методами, предоставленными с веб сервисом.
 - Есть поддержка транзакций, уровней безопасностей и прочего.
 - Присутствует очень много спецификаций.
 - Может использоваться разные транспортные уровни. Можно обращаться к веб сервисам по HTTP запросам, по JMS (Java Message Service) и другие.
 - Данный веб сервис разрабатывается сложнее.
- **REST** (Representational State Transfer)
- **XML-RPC** (XML Remote Procedure Call)

SOAP произошел от XML-RPC и является следующей ступенью его развития. В то время как REST — это концепция, в основе которой лежит скорее архитектурный стиль, нежели новая технология, основанный на теории манипуляции объектами CRUD (Create Read Update Delete) в контексте концепций WWW.

Существуют и иные протоколы, но, поскольку они не получили широкого распространения.

SOAP против REST

SOAP более применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD, а вот в тех приложениях, которые не покидают рамки данной теории, вполне применимым может оказаться именно REST ввиду своей простоты и прозрачности.

Действительно, если любым объектам вашего сервиса не нужны более сложные взаимоотношения, кроме: «Создать», «Прочитать», «Изменить», «Удалить» (как правило — в 99% случаев этого достаточно), возможно, именно REST станет правильным выбором. Кроме того, REST по сравнению с SOAP, может оказаться и более производительным, так как не требует затрат на разбор сложных XML команд на сервере (выполняются обычные HTTP запросы — PUT, GET, POST, DELETE). Хотя SOAP, в свою очередь, более надежен и безопасен.

149. RESTful сервис. Принципы работы.

Ответ

Что такое JAX-RS?

Java API for RESTful Web Services (JAX-RS) – это спецификация, предоставляющая поддержку для создания веб-сервисов в стиле REST (Representational State Transfer) с использованием языка программирования Java.

Что такое RESTful Web Services?

RESTful Web Services – это легковесный и масштабируемый подход к созданию веб-сервисов, основанный на принципах архитектурного стиля REST. Он использует стандартные протоколы и конвенции, такие как HTTP, и поддерживает различные форматы данных, такие как XML, JSON и др.

REST расшифровывается как REpresentational State Transfer SQL.

Отличительной особенностью сервисов REST является то, что они позволяют наилучшим образом использовать протокол HTTP.

Основные компоненты JAX-RS

JAX-RS предоставляет набор аннотаций и интерфейсов, которые упрощают разработку RESTful Web Services. Вот некоторые из основных компонентов:

- **Resource classes:** классы, представляющие ресурсы веб-сервиса и содержащие методы для обработки HTTP-запросов.
- **Resource methods:** методы, аннотированные с помощью HTTP-методов (`@GET`, `@POST`, `@PUT`, `@DELETE`), которые обрабатывают соответствующие запросы.
- **Path annotations:** аннотации, определяющие путь к ресурсам и методам (`@Path`).
- **Parameter annotations:** аннотации, позволяющие получать параметры из запросов (`@PathParam`, `@QueryParam`, `@HeaderParam`, `@FormParam`).

Как работает REST API: 6 принципов архитектуры

Всего в REST есть шесть требований к проектированию API. Пять из них обязательные, одно — опциональное:

- Клиент-серверная модель (client-server model).
- Отсутствие состояния (statelessness).
- Кэширование (cacheability).
- Единообразие интерфейса (uniform interface).
- Многоуровневая система (layered system).
- Код по требованию (code on demand) — необязательно.

Чтобы разобраться в них подробнее, нужно понимать, что в вебе называют ресурсами. **Ресурсы** — это любые данные: текст, изображение, видео, аудио, целая программа. Например, HTML веб-страницы, на которой вы сейчас находитесь, — тоже ресурс.

SQL

150. Что такое нормализация.

Ответ

РЕЛЯЦИОННАЯ БАЗА ДАННЫХ

Под базой данных можно понимать любой набор информации, которую можно найти в этой базе данных и воспользоваться ей, однако если говорить в контексте SQL, то речь будет идти, конечно, о реляционных базах данных, а что же это такое?

Реляционная база данных — это упорядоченная информация, связанная между собой определёнными отношениями. Логически такая база данных представлена в виде таблиц, в которых и лежит вся эта информация.

НОРМАЛИЗАЦИЯ БАЗ ДАННЫХ

Нормализация — это процесс удаления избыточных данных.

Нормализация — это метод проектирования базы данных, который позволяет привести базу данных к минимальной избыточности.

Избыточность устраняется, как правило, за счёт декомпозиции отношений (таблиц), т.е. разбиения одной таблицы на несколько.

Зачем нормализовать базу данных?

Избыточность данных создает предпосылки для появления различных аномалий, снижает производительность, и делает управление данными не гибким и не очень удобным. Отсюда можно сделать вывод, что нормализация нужна для:

- Устранения аномалий
- Повышения производительности
- Повышения удобства управления данными

Избыточность данных

Избыточность данных – это когда одни и те же данные хранятся в базе в нескольких местах, именно это и приводит к аномалиям.

Если существует избыточность данных необходимо добавлять, изменять или удалять одни и те же данные в нескольких местах. Например, если не выполнить операцию в каком-нибудь одном месте, то возникает ситуация, когда одни данные не соответствуют вроде как точно таким же данным в другом месте.

Пример

Допустим, у нас есть следующая таблица, она хранит информацию о предметах мебели, в частности наименование предмета и материал, из которого изготовлен этот предмет.

Идентификатор предмета	Наименование предмета	Материал
1	Стул	Металл
2	Стол	Массив дерева
3	Кровать	ЛДСП
4	Шкаф	Массив дерева
5	Комод	ЛДСП

А теперь допустим, что у нас возникла необходимость подкорректировать название материала, вместо «*Массив дерева*» нужно написать «*Натуральное дерево*», и чтобы это сделать нам необходимо внести изменения сразу в несколько строк, так как предметов, изготовленных из массива дерева, несколько, а именно 2: стол и шкаф.

А теперь представьте, что по каким-то причинам мы внесли изменения только в одну строку, в итоге в нашей таблице будет и «*Массив дерева*», и «*Натуральное дерево*».

Идентификатор предмета	Наименование предмета	Материал
1	Стул	Металл
2	Стол	Натуральное дерево
3	Кровать	ЛДСП
4	Шкаф	Массив дерева
5	Комод	ЛДСП

Какое из этих названий будет правильным? А если представить, что мы можем внести еще какое-то новое значение при добавлении новых записей, например, просто «*Дерево*».

В этом случае в нашей таблице в скором времени будет и «*Массив дерева*», и «*Натуральное дерево*», и просто «*Дерево*», и вообще, что угодно, ведь это просто текст.

Идентификатор предмета	Наименование предмета	Материал
1	Стул	Металл
2	Стол	Натуральное дерево
3	Кровать	ЛДСП
4	Шкаф	Массив дерева
5	Комод	ЛДСП
6	Тумба	Дерево

Однако по своей сути это один и тот же материал, мы просто решили или подкорректировать его название, или ошиблись при добавлении новой записи. Это и есть аномалия, когда одни данные в одном месте не соответствуют вроде как

точно таким же данным в другом месте. Это всего лишь один вид аномалии, однако в процессе добавления, изменения и удаления данных может возникать много других противоречивых ситуаций, т.е. аномалий.

Именно поэтому мы должны устранять избыточность данных в базе, т.е. проводить так называемую нормализацию базы данных.

В данном конкретном случае мы должны название материала, из которого изготовлены предметы мебели, вынести в отдельную таблицу, а в таблице с предметами сделать всего лишь ссылку на нужный материал, тем самым, соотнеся эту ссылку с исходной записью, мы будем понимать, из какого материала сделан тот или иной предмет.

Предметы мебели.

Идентификатор предмета	Наименование предмета	Идентификатор материала
1	Стул	2
2	Стол	1
3	Кровать	3
4	Шкаф	1
5	Комод	3

Материалы, из которых изготовлены предметы мебели.

Идентификатор материала	Материал
1	Массив дерева
2	Металл
3	ЛДСП

В этом случае, когда нам потребуется изменить название материала, мы будем вносить изменение только в одном месте, т.е. править только одну строку.

Таким образом, представляя материалы в виде отдельной сущности и создавая для нее отдельную таблицу, мы устраняем описанную выше аномалию.

Другими словами, каждая сущность должна храниться отдельно, а в случае необходимости использования этой сущности в другой таблице на нее делается всего лишь ссылка, т.е. выстраивается связь.

Нормальные формы базы данных

Нормальная форма базы данных – это набор правил и критериев, которым должна отвечать база данных.

Каждая следующая нормальная форма содержит более строгие правила и критерии, тем самым приводя базу данных к определённой нормальной форме мы устраняем определённый набор аномалий.

Процесс нормализации – это последовательный процесс приведения базы данных к эталонному виду, т.е. переход от одной нормальной формы к следующей.

Процесс перехода от одной нормальной формы к следующей – это **усовершенствование базы данных**. Если база данных находится в какой-то определённой нормальной форме – это означает, что в базе данных отсутствует определённый вид аномалий.

Существует 5 основных нормальных форм базы данных:

- Первая нормальная форма (1NF)
- Вторая нормальная форма (2NF)
- Третья нормальная форма (3NF)
- Четвертая нормальная форма (4NF)
- Пятая нормальная форма (5NF)

Однако выделяют еще дополнительные нормальные формы:

- Ненормализованная форма или нулевая нормальная форма (UNF)
- Нормальная форма Бойса-Кодда (BCNF)
- Доменно-ключевая нормальная форма (DKNF)
- Шестая нормальная форма (6NF)

Если объединить оба этих списка и упорядочить нормальные формы от менее нормализованной до самой нормализованной, т.е. начиная с формы, при которой база данных по своей сути не является нормализованной, и заканчивая самой строгой нормальной формой, то мы получим следующий перечень:

а) Ненормализованная форма или нулевая нормальная форма (UNF)

Требования ненормализованной формы или нулевой нормальной формы (UNF)

По реляционной теории строки в таблицах не должны быть пронумерованы, т.е. порядок строк не имеет значения, так же как не имеет значения порядок столбцов. Например, если мы поменяем порядок столбцов, или порядок строк, ничего измениться не должно, это не должно ни на что повлиять. Таким образом по реляционной теории мы не можем обратиться к определённой строке или столбцу по ее номеру.

Пример приведения таблицы к ненормализованной форме или нулевой нормальной форме (UNF)

Достаточно часто в Excel можно встретить таблицы следующего вида.

Порядковый номер строки	А	В
1	Иван	Иванов
2	Сергей	Сергеев
3	John	Smith
4	Иван	Иванов

Однако, к сожалению, подобные таблицы нельзя назвать реляционными, так как это пронумерованные двумерные массивы данных. И если мы поменяем местами строки, то наша нумерация просто нарушится.

По реляционной теории данные в таблицах никак не упорядочены, и мы не можем сказать, *что нам нужно получить данные из строки с порядковым номером 2* или из второго столбца.

Поэтому чтобы приступить к нормализации нашей таблицы, нам необходимо как минимум удалить столбец с порядковым номером и не учитывать порядок столбцов, например, задав им более корректные имена.

first_name	last_name
Иван	Иванов
Сергей	Сергеев
John	Smith
Иван	Иванов

б) Первая нормальная форма (1NF)

Требования первой нормальной формы (1NF)

Таблицы должны соответствовать реляционной модели данных и соблюдать определённые реляционные принципы. Реляционные принципы:

- В таблице не должно быть дублирующих строк
- В каждой ячейке таблицы хранится атомарное значение (одно не составное значение)
- В столбце хранятся данные одного типа
- Отсутствуют массивы и списки в любом виде

Главное правило первой нормальной формы

Строки, столбцы и ячейки в таблицах необходимо использовать строго по назначению.

- Назначение строк – хранить данные
- Назначение столбцов – хранить структурную информацию
- Назначение ячеек – хранить атомарное значение

Т.е. если ячейка таблицы по реляционной теории должна хранить одно атомарное значение, не нужно записывать в ячейку какой-то список значений или составное значение. Также не нужно создавать строки, которые уже есть в таблице и хранить в столбце значения разных типов данных.

Пример приведения таблицы к первой нормальной форме

Следующая таблица не находится в первой нормальной форме, так как:

- в таблице есть дублирующие строки (John Smith) – это нарушен первый принцип (в таблице не должно быть дублирующих строк);
- в некоторых ячейках хранятся списки значений (*каждый номер телефона — это одно значение*) – это нарушен четвёртый принцип (Отсутствуют массивы и списки в любом виде).

Таблица сотрудников в ненормализованном виде.

Сотрудник	Контакт
Иванов И.И.	123-456-789, 987-654-321
Сергеев С.С.	Рабочий телефон 555-666-777, Домашний телефон 777-888-999
John Smith	123-456-789
John Smith	123-456-789

Решение

Чтобы привести эту таблицу к первой нормальной форме, необходимо удалить дублирующие строки, в ячейках хранить один номер телефона, а не список, а тип телефона (*домашний или рабочий*) вынести в отдельный столбец, так как столбцы хранят структурную информацию.

Таблица сотрудников в первой нормальной форме.

Сотрудник	Телефон	Тип телефона
Иванов И.И.	123-456-789	
Иванов И.И.	987-654-321	
Сергеев С.С.	555-666-777	Рабочий телефон
Сергеев С.С.	777-888-999	Домашний телефон
John Smith	123-456-789	

с) Вторая нормальная форма (2NF)

Требования второй нормальной формы (2NF)

Таблицы в базе данных должны удовлетворять следующим требованиям:

- Таблица должна находиться в первой нормальной форме
- Таблица должна иметь ключ
- Все не ключевые столбцы таблицы должны зависеть от полного ключа (*в случае если он составной*)

Ключ – это столбец или набор столбцов, по которым гарантировано можно отличить строки друг от друга, т.е. ключ идентифицирует каждую строку таблицы. По ключу мы можем обратиться к конкретной строке данных в таблице.

Если ключ составной, т.е. состоит из нескольких столбцов, то все остальные неключевые столбцы должны зависеть от всего ключа, т.е. от всех столбцов в этом ключе. Если какой-то атрибут (столбец) зависит только от одного столбца в ключе, значит, база данных не находится во второй нормальной форме.

Иными словами, в таблице не должно быть данных, которые можно получить, зная только половину ключа, т.е. только один столбец из составного ключа.

Главное правило второй нормальной формы (2NF)

Таблица должна иметь правильный ключ, по которому можно идентифицировать каждую строку.

Пример приведения таблицы ко второй нормальной форме

Представим, что нам нужно хранить список сотрудников организации, и для этого мы создали следующую таблицу.

Таблица сотрудников в первой нормальной форме.

ФИО	Должность	Подразделение	Описание подразделения
Иванов И.И.	Программист	Отдел разработки	Разработка и сопровождение приложений и сайтов
Сергеев С.С.	Бухгалтер	Бухгалтерия	Ведение бухгалтерского и налогового учета финансово-хозяйственной деятельности
John Smith	Продавец	Отдел реализации	Организация сбыта продукции

Мы видим, что она удовлетворяет условиям первой нормальной формы, т.е. в ней нет дублирующих строк и все значения атомарны.

Теперь мы можем начать процесс нормализации этой таблицы до второй нормальной формы.

Нам нужно внедрить первичный ключ.

Поработав немного с предметной областью, мы выясняем, что в этой организации каждому сотруднику присваивается уникальный табельный номер, который никогда не будет изменен.

Поэтому очевидно, что для таблицы, которая будет хранить список сотрудников, первичным ключом может выступать табельный номер, зная который мы можем четко идентифицировать каждого сотрудника, т.е. каждую строку нашей таблицы. Если бы такого табельного номера у нас не было или в рамках организации он мог повторяться (*например, сотрудник уволился, и спустя время его номер присвоили новому сотруднику*), то для первичного ключа мы могли бы создать искусственный ключ с целочисленным типом данных, который автоматически увеличивался бы в случае добавления новых записей в таблицу. Тем самым мы бы точно также четко идентифицировали каждую строку в таблице.

Таким образом, чтобы привести эту таблицу ко второй нормальной форме, мы должны добавить в нее еще один атрибут, т.е. столбец с табельным номером.

Таблица сотрудников во второй нормальной форме с простым первичным ключом.

Табельный номер	ФИО	Должность	Подразделение	Описание подразделения
1	Иванов И.И.	Программист	Отдел разработки	Разработка и сопровождение приложений и сайтов
2	Сергеев С.С.	Бухгалтер	Бухгалтерия	Ведение бухгалтерского и налогового учета финансово-хозяйственной деятельности
3	John Smith	Продавец	Отдел реализации	Организация сбыта продукции

В результате, так как наш первичный ключ является простым, а не составным, наша таблица автоматически переходит во вторую нормальную форму.

Пример приведения таблицы ко второй нормальной форме (первичный ключ составной)

Представим, что наша организация выполняет несколько проектов, в которых может быть задействовано несколько участников, и нам необходимо хранить информацию об этих проектах. В частности, мы хотим знать, кто участвует в каждом из проектов, продолжительность этого проекта, ну и возможно какие-то другие сведения. При этом мы понимаем, что отдельно взятый сотрудник может участвовать в нескольких проектах.

Для хранения таких данных мы создали следующую таблицу.

Таблица проектов организации в первой нормальной форме.

Название проекта	Участник	Должность	Срок проекта (мес.)
Внедрение приложения	Иванов И.И.	Программист	8
Внедрение приложения	Сергеев С.С.	Бухгалтер	8
Внедрение приложения	John Smith	Менеджер	8
Открытие нового магазина	Сергеев С.С.	Бухгалтер	12
Открытие нового магазина	John Smith	Менеджер	12

Данная таблица в первой нормальной форме, значит, надо приводить ее ко второй нормальной форме.

Чтобы привести таблицу ко второй нормальной форме, необходимо определить для нее первичный ключ.

Посмотрев на эту таблицу, мы понимаем, что четко идентифицировать каждую строку мы можем только с помощью комбинации столбцов, например, «Название проекта» + «Участник», иными словами, зная «Название проекта» и «Участника», мы можем четко определить конкретную запись в таблице, т.е. каждое сочетание значений этих столбцов является уникальным.

Таким образом, мы определили первичный ключ и он у нас составной, т.е. состоящий из двух столбцов.

Название проекта	Участник	Должность	Срок проекта (мес.)
Внедрение приложения	Иванов И.И.	Программист	8
Внедрение приложения	Сергеев С.С.	Бухгалтер	8
Внедрение приложения	John Smith	Менеджер	8
Открытие нового магазина	Сергеев С.С.	Бухгалтер	12
Открытие нового магазина	John Smith	Менеджер	12

Так как первичный ключ составной, необходимо проверить еще и второе требование, которое гласит, что «Все не ключевые столбцы таблицы должны зависеть от полного ключа».

Другими словами, остальные столбцы, которые не входят в первичный ключ, должны зависеть от всего первичного ключа, т.е. от всех столбцов, а не от какого-то одного.

Чтобы это проверить, мы можем задать себе несколько вопросов.

Можем ли мы определить «Должность», зная только название проекта? Нет. Для этого нам необходимо знать и участника. Значит, пока все хорошо, по этой части ключа мы не можем четко определить значение неключевого столбца. Идем дальше и проверяем другую часть ключа.

Можем ли мы определить «Должность» зная только участника? Да, можем. Значит наш **первичный ключ плохой**, и требование второй нормальной формы не выполняется.

Что делать в этом случае?

В этом случае мы будем выполнять действие, которое выполняется, наверное, в 99% случаев на протяжении всего процесса нормализации базы данных – это декомпозиция.

Декомпозиция – это процесс разбиения одного отношения (таблицы) на несколько.

Чтобы декомпонировать нашу таблицу и привести базу данных к нормализованной форме, мы должны создать следующие таблицы.

Проекты.

Идентификатор проекта	Название проекта	Срок проекта (мес.)
1	Внедрение приложения	8
2	Открытие нового магазина	12

Участники.

Идентификатор участника	Участник	Должность
1	Иванов И.И.	Программист
2	Сергеев С.С.	Бухгалтер
3	John Smith	Менеджер

Связь проектов и участников этих проектов.

Идентификатор проекта	Идентификатор участника
1	1
1	2
1	3
2	2
2	3

Мы создали 3 таблицы:

1. Проекты, в нее мы добавили искусственный первичный ключ
2. Участники, в нее мы также добавили искусственный первичный ключ
3. Связь между проектами и участниками, она нужна для реализации связи «Многие ко многим», так как между этими таблицами связь именно такая

d) Третья нормальная форма (3NF)

Требования третьей нормальной формы (3NF)

В таблицах базы данных должна отсутствовать транзитивная зависимость.

Транзитивная зависимость – это когда не ключевые столбцы зависят от значений других не ключевых столбцов.

Чтобы нормализовать базу данных до третьей нормальной формы, необходимо сделать так, чтобы в таблицах отсутствовали не ключевые столбцы, которые зависят от других не ключевых столбцов.

Главное правило третьей нормальной форме (3NF)

Таблица должна содержать правильные не ключевые столбцы

Пример приведения таблиц базы данных к третьей нормальной форме

Таблица сотрудников во второй нормальной форме.

Табельный номер	ФИО	Должность	Подразделение	Описание подразделения
1	Иванов И.И.	Программист	Отдел разработки	Разработка и сопровождение приложений и сайтов
2	Сергеев С.С.	Бухгалтер	Бухгалтерия	Ведение бухгалтерского и налогового учета финансово-хозяйственной деятельности
3	John Smith	Продавец	Отдел реализации	Организация сбыта продукции

Чтобы определить, находится ли эта таблица в третьей нормальной форме, мы должны проверить все не ключевые столбцы, каждый из них должен зависеть только от первичного ключа, и никаким образом к другим не ключевым столбцам он не должен относиться.

Однако, в результате проверки мы выясняем, что столбец «Описание подразделения» не зависит напрямую от первичного ключа. Мы это выяснили, когда задали себе один вопрос «Каким образом описание подразделения связано с

сотрудником?». И наш ответ звучит следующим образом: «Атрибут описание подразделения содержит детальные сведения того подразделения, в котором работает сотрудник».

Отсюда следует, что столбец «Описание подразделения» не связан напрямую с сотрудником, он связан напрямую со столбцом «Подразделение», который напрямую связан с сотрудником, ведь сотрудник работает в каком-то конкретном подразделении. Это и есть транзитивная зависимость, когда один не ключевой столбец связан с первичным ключом через другой не ключевой столбец.

Чтобы привести эту таблицу к третьей нормальной форме, мы должны сделать что? Правильно, декомпозицию! Мы должны эту таблицу разбить на две: в первой хранить сотрудников, а во второй подразделения. А для реализации связи в таблице сотрудников создать ссылку на таблицу подразделений, т.е. добавить внешний ключ.

Таблица сотрудников в третьей нормальной форме.

Табельный номер	ФИО	Должность	Подразделение
1	Иванов И.И.	Программист	1
2	Сергеев С.С.	Бухгалтер	2
3	John Smith	Продавец	3

Таблица подразделений в третьей нормальной форме.

Идентификатор подразделения	Подразделение	Описание подразделения
1	Отдел разработки	Разработка и сопровождение приложений и сайтов
2	Бухгалтерия	Ведение бухгалтерского и налогового учета финансово-хозяйственной деятельности
3	Отдел реализации	Организация сбыта продукции

е) Нормальная форма Бойса-Кодда (BCNF)
Между 3 и 4 нормальной формой есть еще и промежуточная нормальная форма, она называется – Нормальная форма Бойса-Кодда (BCNF). Иногда ее еще называют «Усиленная третья нормальная форма». Промежуточной или усиленной третьей нормальной формой ее называют потому, что ситуации, в которых могут предъявляться требования нормальной формы Бойса-Кодда, возникают не всегда, т.е. это некий частный случай, именно поэтому данная форма не включена в основную градацию.

Требования нормальной формы Бойса-Кодда

- Требования нормальной формы Бойса-Кодда следующие:
- Таблица должна находиться в третьей нормальной форме.
- Здесь все как обычно, т.е. как и у всех остальных нормальных форм, первое требование заключается в том, чтобы таблица находилась в предыдущей нормальной форме (то есть соответствует предыдущим нормальным формам), в данном случае в третьей нормальной форме;
- Ключевые атрибуты составного ключа не должны зависеть от не ключевых атрибутов.

Главное правило нормальной формы Бойса-Кодда (BCNF)
Часть составного первичного ключа не должна зависеть от не ключевого столбца.

Пример приведения таблиц базы данных к нормальной форме Бойса-Кодда

Представим, что у нас есть организация, которая реализует множество различных проектов. При этом в каждом проекте работа ведётся по нескольким функциональным направлениям, в каждом из которых есть свой куратор. Сотрудник может быть куратором только того направления, на котором он специализируется, т.е. если сотрудник программист, он не может курировать в проекте направление, связанное с бухгалтерией.

Допустим, что нам нужно хранить информацию о кураторах всех проектов по каждому направлению.

В итоге мы реализуем следующую таблицу, в которой первичный ключ составной «Проект + Направление», так как в каждом проекте есть несколько направлений работы и поэтому, зная только проект, мы не можем определить куратора

направления, так же как зная только направление, мы не сможем определить куратора, нам нужно знать и проект, и направление, чтобы определить куратора этого направления в этом проекте.

Таблица проектов и кураторов.

Проект	Направление	Куратор
1	Разработка	Иванов И.И.
1	Бухгалтерия	Сергеев С.С.
2	Разработка	Иванов И.И.
2	Бухгалтерия	Петров П.П.
2	Реализация	John Smith
3	Разработка	Андреев А.А.

Наша таблица находится в третьей нормальной форме, так как у нас есть первичный ключ, а не ключевой столбец зависит от всего ключа, а не от какой-то его части.

Но в данном случае таблица не находится в нормальной форме Бойса-Кодда, дело в том, что, зная куратора, мы можем четко определить, какое направление он курирует, иными словами, часть составного ключа, т.е. «*Направление*», зависит от неключевого атрибута, т.е. «*Куратора*».

Чтобы привести данную таблицу к нормальной форме Бойса-Кодда, необходимо, как всегда сделать декомпозицию данного отношения, т.е. разбить эту таблицу на несколько таблиц.

Таблица кураторов.

Идентификатор куратора	ФИО	Направление
1	Иванов И.И.	Разработка
2	Сергеев С.С.	Бухгалтерия
3	Петров П.П.	Бухгалтерия
4	John Smith	Реализация
5	Андреев А.А.	Разработка

Таблица связи кураторов и проектов.

Проект	Идентификатор куратора
1	1
1	2
2	1
2	3
2	4
3	5

Таким образом, в таблице кураторов у нас хранится список кураторов и их специализация, т.е. направление, которое они могут курировать, а в таблице связи кураторов и проектов отражается связь проектов и кураторов.

f) Четвертая нормальная форма (4NF)

Требования четвертой нормальной формы (4NF)

Требование четвертой нормальной формы (4NF) заключается в том, чтобы в таблицах отсутствовали нетривиальные многозначные зависимости.

В таблицах многозначная зависимость выглядит следующим образом.

Начнем с того, что таблица должна иметь как минимум три столбца, допустим А, В и С, при этом В и С между собой никак не связаны и не зависят друг от друга, но по отдельности зависят от А, и для каждого значения А есть множество значений В, а также множество значений С.

В данном случае многозначная зависимость обозначается вот так:

A \twoheadrightarrow B

A \twoheadrightarrow C

Если подобная многозначная зависимость есть в таблице, то она не соответствует четвертой нормальной форме.

Главное правило четвертой нормальной формы:

В таблице не должно быть многозначных зависимостей

Пример приведения таблиц базы данных к четвертой нормальной форме

Представим, что мы работаем в каком-то учебном заведении, где есть курсы, которые изучают студенты, преподаватели, которые читают эти курсы, и аудитории, в которых преподаватели проводят занятия по курсам.

Курсы.

Идентификатор курса	Название курса
1	SQL
2	Python
3	JavaScript

Преподаватели.

Идентификатор преподавателя	ФИО
1	Иванов И.И.
2	Сергеев С.С.
3	John Smith

Аудитории.

Идентификатор аудитории	Название аудитории
1	101
2	203
3	305
4	407
5	502

При этом мы понимаем, что один и тот же курс могут преподавать разные преподаватели, и необязательно в какой-то одной аудитории, один раз курс может читаться в одной аудитории, а в другой раз совсем в другой аудитории, например, на курс записалось гораздо меньше студентов, и чтобы не занимать аудиторию большого размера, под этот поток могут выделить аудиторию меньшего размера.

Также стоит отметить, что под каждый курс подходит только определенный набор аудиторий, например, те, которые оснащены необходимым оборудованием, или те, которые имеют соответствующую вместимость для конкретно этого курса.

В учебном заведении, конечно же, постоянно возникают вопросы с составлением расписания, однако для того чтобы его составлять, необходимо предварительно знать возможности этого учебного заведения. Иными словами, какие преподаватели могут преподавать тот или иной курс, а также в каких аудиториях тот или иной курс может читаться.

Для этого нам необходимо соединить эти три сущности в одной таблице. В итоге у нас получается следующая таблица (для наглядности здесь представлены текстовые значения, а не идентификаторы).

Таблица связей курсов, преподавателей и аудиторий.

Курс	Преподаватель	Аудитория
SQL	Иванов И.И.	101
SQL	Иванов И.И.	203
SQL	Сергеев С.С.	305
SQL	Сергеев С.С.	407
Python	John Smith	502
Python	John Smith	305

В данном случае первичный ключ здесь состоит из всех трех столбцов, поэтому эта таблица автоматически находится в третьей нормальной форме и нормальной форме Бойса-Кодда. Однако она не находится в четвертой нормальной форме, так как здесь есть многозначная зависимость

Курс ->-> Преподаватель

Курс ->-> Аудитория

Т.е. для каждого курса в этой таблице может быть несколько преподавателей, а также несколько аудиторий.

Два атрибута «Преподаватель» и «Аудитория» никак не зависят друг от друга, но они оба по отдельности зависят от курса.

Но что же плохого в этой таблице и в этой многозначной зависимости? Вы можете спросить.

Чтобы ответить на этот вопрос, мы можем задать себе несколько других вопросов.

Что будет если, например, преподаватель «Иванов И.И.» уволился? Нам нужно будет удалить две строки из этой таблицы, но удалив эти строки, мы удалим всю информацию и о аудиториях 101 и 203. Но они на самом-то деле есть и должны участвовать в планировании расписания. Это аномалия, и это плохо.

Или другая ситуация, что будет, если курсу назначен преподаватель, но аудитория еще не определена? Или наоборот, с аудиторией уже определились, а вот преподаватель еще не известен.

Мы должны создать записи либо с NULL либо со значениями по умолчанию, и это также является аномалией.

Многозначные зависимости плохи как раз тем, что их нельзя независимо друг от друга редактировать. Иными словами, чтобы внести изменения в одну зависимость, мы неизбежно должны затронуть другую зависимость.

Решение в данном случае как всегда – **декомпозиция**.

Мы должны вынести каждую многозначную зависимость в отдельную таблицу, т.е. разнести независимые друг от друга атрибуты, в нашем случае «Преподаватель» и «Аудитория», по разным таблицам.

Связь курсов и преподавателей.

Курс	Преподаватель
SQL	Иванов И.И.
SQL	Сергеев С.С.
Python	John Smith

Связь курсов и аудиторий.

Курс	Аудитория
SQL	101
SQL	203
SQL	305

SQL	407
Python	502
Python	305

Классический пример приведения таблиц базы данных к четвертой нормальной форме

Чтобы стало еще понятней, давайте закрепим знания и рассмотрим классический пример, который обычно используется в литературе для пояснения четвертой нормальной формы.

Таблица связей студентов, курсов и хобби.

Студент	Курс	Хобби
Иванов И.И.	SQL	Футбол
Иванов И.И.	Java	Хоккей
Сергеев С.С.	SQL	Волейбол
Сергеев С.С.	SQL	Теннис
John Smith	Python	Футбол
John Smith	Java	Теннис

Данная таблица хранит информацию о студентах, в частности здесь хранятся курсы, которые посещает студент, и увлечения этого студента, т.е. хобби.

Отсюда следует, что каждый студент может посещать несколько курсов и иметь несколько увлечений.

Первичный ключ здесь также составной и состоит он из всех трех столбцов.

При этом мы можем заметить, что курс и хобби никак не связаны и не зависят друг от друга, но по отдельности зависят от студента.

Таким образом, мы можем наблюдать в этой таблице нетривиальную многозначную зависимость

Студент ->-> Курс

Студент ->-> Хобби

Поэтому эта таблица не находится в четвертой нормальной форме.

Кроме всех тех аномалий, связанных с редактированием данных, которые мы уже рассмотрели на предыдущем примере, в данном случае еще продемонстрирована проблема неоднозначной выборки данных.

Допустим, нам необходимо получить информацию о хобби студентов, которые посещают курс по SQL. Очевидным действием станет выборка с условием Курс = SQL, в результате мы получим 3 хобби: футбол, волейбол и теннис.

Результат выборки. Хобби студентов, которые посещают курс по SQL.

Студент	Курс	Хобби
Иванов И.И.	SQL	Футбол
Сергеев С.С.	SQL	Волейбол
Сергеев С.С.	SQL	Теннис

Однако, если мы заглянем в исходную таблицу, то мы четко увидим, что «Иванов И.И.» посещает курс по SQL и имеет хобби «Хоккей», но в нашей выборке этого хобби нет.

Чтобы нормализовать эту таблицу, мы должны точно так же, как и в предыдущем примере, разбить ее на две.

Связь студентов и курсов.

Студент	Курс
Иванов И.И.	SQL

Иванов И.И.	Java
Сергеев С.С.	SQL
John Smith	Python
John Smith	Java

Связь студентов и хобби.

Студент	Хобби
Иванов И.И.	Футбол
Иванов И.И.	Хоккей
Сергеев С.С.	Волейбол
Сергеев С.С.	Теннис
John Smith	Футбол
John Smith	Теннис

Бывают ли базы данные с нормализацией выше 4-ой формы

Однако в реальности такую ситуацию и такую таблицу вряд ли можно встретить, так как следуя здравому смыслу такие абсолютно не связанные друг с другом данные никто не будет хранить в одной таблице. Поэтому этот пример чисто теоретический и приводится для демонстрации принципов четвертой нормальной формы.

И если говорить о реальных данных, то нормализация до четвертой нормальной формы, как и до всех последующих, в современном мире практически не встречается. Если четвертую нормальную форму еще как-то можно представить и даже встретить данные, нормализованные до этой формы, то встретить данные, нормализованные до 5 или 6 нормальной формы, практически невозможно.

Вы можете спросить, *а почему не нормализуют данные до 5 или 6 нормальной формы?* Ведь каждая нормальная форма устраняет определенные аномалии, и если сделать полностью нормализованную базу данных, то по сути она будет идеальной, не содержащая ни одной аномалии, это же хорошо.

Да, совершенно верно, база данных не будет содержать аномалий, но давайте вспомним, какие преимущества нам дает нормализация.

Обычно во всех источниках приводится два основных глобальных преимущества:

- Устранение аномалий
- Повышение производительности

Если с устранением аномалий все ясно, т.е. в полностью нормализованной базе данных их не будет и это хорошо, то с повышением производительности не все так однозначно.

Да, нормализация повышает производительность, но только где-то до 3 нормальной формы. Начиная с 4 нормальной формы, производительность увеличиваться не будет, более того, с каждой новой формой производительность будет значительно снижаться, не говоря уже о том, что с нормализованной базой данных до 5 или 6 нормальной формы будет крайне сложно и неудобно работать и сопровождать ее, ведь с каждой новой формой мы значительно увеличиваем количество таблиц в базе данных.

Поэтому процесс нормализации не является строго обязательным, т.е. не нужно нормализовать базу данных, только для того чтобы она была нормализована.

В процессе проектирования базы данных необходимо следовать здравому смыслу и найти баланс между отсутствием аномалий и приемлемой производительностью.

Полностью нормализованная база данных – это **плохая база данных**.

Хорошая база данных – это база, которая достаточно нормализована, чтобы не создавать аномалии для пользователей этой базы данных, и в то же время она имеет хорошую производительность.

g) Пятая нормальная форма (5NF)

Требования пятой нормальной формы (5NF)

Переменная отношения находится в пятой нормальной форме (иначе – в проекционно-соединительной нормальной форме) тогда и только тогда, когда каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.

На основе этого определения мы можем сделать следующий вывод:

Требование пятой нормальной формы (5NF) заключается в том, чтобы в таблице каждая **нетривиальная зависимость соединения** определялась потенциальным ключом этой таблицы.

существуют таблицы, которые не получится декомпозировать на две таблицы без потери данных, т.е. какие-то данные мы потеряем при соединении двух итоговых, полученных после декомпозиции, таблиц. Но, если декомпозировать такую таблицу не на две, а на три таблицы, то потери данных можно избежать.

И таблица будет находиться в пятой нормальной форме, если при соединении (JOIN) этих трех таблиц, которые были получены в результате декомпозиции, будут формироваться ровно те же самые данные, что и в исходной таблице до декомпозиции. Однако если этого происходить не будет, т.е. данные будут отличаться, например, какие-то строки были потеряны, или созданы новые, то в этом случае возникает так называемая **зависимость соединения**, т.е. часть данных одного столбца зависит от части данных другого столбца.

Таким образом, таблица будет находиться в пятой нормальной форме, если она не будет содержать зависимости соединения.

Декомпозиция без потерь – процесс разбиения одной таблицы на несколько, при условии, что в случае соединения таблиц, которые были получены в результате декомпозиции, будет формироваться ровно та же самая информация, что и в исходной таблице до декомпозиции.

Иными словами, чтобы выполнить требование пятой нормальной формы, необходимо осуществить декомпозицию таблицы без потери данных.

Допустим, существует таблица **T (C1, C2, C3)** где C1, C2, C3 – столбцы и вместе они являются составным первичным ключом. Таблица находится в четвертой нормальной форме. В соответствии с требованиями предметной области у нас проявляется зависимость соединения:

{C1, C2}, {C1, C3}, {C2, C3}

Чтобы привести данную таблицу к пятой нормальной форме, необходимо декомпозировать ее на следующие три таблицы:

T1 (C1, C2)

T2 (C1, C3)

T3 (C2, C3)

При этом, если мы соединим (JOIN) эти три новые таблицы (T1, T2, T3) и получим исходную таблицу (T), то это будет означать, что декомпозицию мы выполнили без потерь.

Пример приведения таблиц базы данных к пятой нормальной форме

Представим, что у нас есть таблица, которая хранит информацию о связи сотрудников с проектами и направлениями работы сотрудников в этих проектах.

Сразу хочется отметить, если Вас когда-то попросят определить, находится та или иная таблица в 5 нормальной форме, то Вы смело можете отвечать *«неизвестно, так как все зависит от требований предметной области»*.

В случае нашей таблицы мы также не можем сказать, находится ли она в 5NF или нет, так как нам сначала необходимо разобраться в предметной области и определить ограничения.

Связь сотрудников с проектами и направлениями работы в проектах.

Сотрудник	Проект	Направление
Иванов И.И.	Интернет магазин	Разработка
Сергеев С.С.	Интернет магазин	Бухгалтерия
Сергеев С.С.	Новый офис	Реализация
John Smith	Личный кабинет	Бухгалтерия
Иванов И.И.	Личный кабинет	Разработка
Иванов И.И.	Информационная система	Разработка

Поработав с предметной областью, мы выясняем, что:

- Иванов И.И. может работать только в направлении *«Разработка»*
- Сергеев С.С. может работать в любом направлении, за исключением *«Разработка»*

- Иванов И.И. может участвовать в большом количестве проектов
- John Smith может участвовать только в одном проекте

Если придерживаться этих требований, то в нашу таблицу можно очень легко внести некорректные данные, и у нас точно так же, как и в случае с четвертой нормальной формой, будут возникать аномалии при добавлении, изменении и удалении данных.

Наша таблица находится в четвертой нормальной форме, так как у нас отсутствует многозначная зависимость, ведь у нас нет таких атрибутов, которые зависели бы от другого атрибута.

Однако принимая во внимание наши требования, мы понимаем, что часть данных каждого из столбцов зависит от части данных другого столбца, т.е. существуют некие зависимости, и эти зависимости определяются не целым потенциальным ключом, а только его частью.

Поэтому, чтобы устранить возможность внесения некорректных данных, мы можем попытаться выполнить декомпозицию без потерь, и тем самым привести таблицу к пятой нормальной форме.

Чтобы выполнить декомпозицию без потерь, нам нужно разбить данную таблицу на три проекции

{Сотрудник, Проект}, {Сотрудник, Направление}, {Проект, Направление}

с условием, что в случае обратного соединения, мы получим те же самые данные, что у нас были и до декомпозиции.

Если это нам удастся сделать, то мы устраним нетривиальные зависимости соединения и нормализуем наши таблицы до пятой нормальной формы.

Связь сотрудников и проектов.

Сотрудник	Проект
Иванов И.И.	Интернет магазин
Сергеев С.С.	Интернет магазин
Сергеев С.С.	Новый офис
John Smith	Личный кабинет
Иванов И.И.	Личный кабинет
Иванов И.И.	Информационная система

Связь сотрудников и направлений.

Сотрудник	Направление
Иванов И.И.	Разработка
Сергеев С.С.	Бухгалтерия
Сергеев С.С.	Реализация
John Smith	Бухгалтерия

Связь проектов и направлений.

Проект	Направление
Интернет магазин	Разработка
Интернет магазин	Бухгалтерия
Новый офис	Реализация
Личный кабинет	Бухгалтерия
Личный кабинет	Разработка
Информационная система	Разработка

Таблицы созданы, теперь если мы выполним следующий запрос, который соединяет эти три таблицы, и он вернет нам точно такие же данные, что и в исходной таблице, то зависимости соединения у нас нет, и наши таблицы находятся в 5NF.

Стоит отметить, что пятая нормальная форма является окончательной нормальной формой по отношению к операциям разбиения таблиц на проекции и их соединения, именно поэтому ее альтернативное название – проекционно-соединительная нормальная форма. Таким образом, если таблица находится в 5NF, то гарантируется, что она не содержит аномалий, которые могут быть исключены посредством ее разбиения на проекции.

Также стоит отметить, что таблицы, которые необходимо нормализовать до пятой нормальной формы, встречаются крайне редко, т.е. это очень частный случай. Более того, такие таблицы являются не совсем удачными с точки зрения проектирования. Кроме всего прочего, чтобы привести таблицу к пятой нормальной форме, Вы должны очень хорошо разбираться в предметной области, чтобы определить зависимости соединения, ведь это действительно очень сложно. Иными словами, если Вам удастся определить эти зависимости соединения, то только в этом случае Вы сможете привести таблицу к пятой нормальной форме.

h) Доменно-ключевая нормальная форма (DKNF)

Пятая нормальная форма является окончательной нормальной формой по отношению к операциям разбиения таблиц на проекции и их соединения.

Однако существуют и другие нормальные формы, например, доменно-ключевая нормальная форма (DKNF), которая, в отличие от предыдущих нормальных форм, не определяется в терминах функциональных зависимостей, многозначных зависимостей или зависимостей соединения. Вместо этого в фокусе внимания в этой нормальной форме стоят ограничения доменов и ограничения ключей.

Требования доменно-ключевой нормальной формы (DKNF)

Ограничение домена – это ограничение, предписывающее использование для определенного атрибута значений только из некоторого заданного домена (набора значений).

Ограничение ключа – это ограничение, утверждающее, что некоторый атрибут или комбинация атрибутов представляет собой потенциальный ключ.

Таким образом, требование доменно-ключевой нормальной формы заключается в том, чтобы каждое наложенное ограничение на таблицу являлось логическим следствием ограничений доменов и ограничений ключей, которые накладываются на данную таблицу.

Таблица, находящаяся в доменно-ключевой нормальной форме, обязательно находится в 5NF, и соответственно, в 4NF и т.д. Однако, стоит отметить, что не всегда возможно привести таблицу к доменно-ключевой нормальной форме, более того, не всегда возможно получить ответ на вопрос о том, когда может быть выполнено такое приведение.

i) Шестая нормальная форма (6NF)

Описание шестой нормальной формы (6NF)

Шестая нормальная форма (6NF) была введена при работе с хронологическими базами данных.

Хронологическая база данных – это база, которая может хранить не только текущие данные, но и исторические данные, т.е. данные, относящиеся к прошлым периодам времени. Однако такая база может хранить и данные, относящиеся к будущим периодам времени.

В процессе проектирования хронологических баз данных возникают некоторые особые проблемы, решить которые можно с помощью: горизонтальной декомпозиции и вертикальной декомпозиции.

В данном случае нас интересует вертикальная декомпозиция, процесс которой очень сильно напоминает нашу классическую нормализацию, которую мы рассматривали до пятой нормальной формы включительно.

Иными словами, декомпозиция таблиц, которую мы использовали для приведения этих таблиц к той или иной нормальной форме, по факту и является вертикальной декомпозицией.

В процессе изучения хронологических баз данных исследователи выдвигали доводы в пользу максимально возможной вертикальной декомпозиции таблиц, а не просто их декомпозиции до какой-то определенной степени, которую требует классическая теория нормализации. Общая идея состояла в том, что таблицы должны быть приведены к неприводимым компонентам, под этим подразумеваются такие компоненты, для которых дальнейшая декомпозиция без потерь становится невозможной.

Теперь стоит напомнить, что пятая нормальная форма основана на так называемых зависимостях соединения.

А поскольку вертикальная декомпозиция, которая используется в хронологических базах данных, представляет собой классическое разделение таблиц на проекции, была сформулирована новая нормальная форма, основанная на обобщенном понятии зависимости соединения, поэтому новую форму называли «Шестая нормальная форма».

Требование шестой нормальной формы заключается в том, что таблица должна удовлетворять всем нетривиальным зависимостям соединения.

Из этого определения следует, что таблица находится в 6NF, когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Стоит отметить, что таблица, которая находится в 6NF, также находится и в 5NF, и во всех предыдущих.

Шестая нормальная форма вводит такое понятие как «Декомпозиция до конца», т.е. максимально возможная декомпозиция таблиц.

Однако, если в хронологических базах данных такая нормализация может быть полезна, так как она позволяет бороться с избыточностью, то в нехронологических базах данных нормализация таблиц до шестой нормальной формы приведёт к значительному снижению производительности. Кроме этого такая нормализация сделает работу с базой данных очень сложной за счет многократного увеличения количества таблиц.

Поэтому шестую нормальную форму в реальном мире не используют, более того, трудно даже представить себе ситуацию, при которой возникала бы необходимость нормализовать базу данных до шестой нормальной формы. Практического применения шестой нормальной формы, наверное, просто нет.

Здесь снова давайте вспомним, что нет никакой необходимости приводить базу данных до какой-то определенной нормальной формы.

Процесс проектирования правильной базы данных – это не процесс приведения ее к самой высокой нормальной форме, это компромисс между отсутствием аномалий и приемлемой производительностью.

Поэтому в процессе нормализации базы данных необходимо руководствоваться в первую очередь требованиями к разрабатываемой системе и требованиями предметной области. Вы должны подумать о том, какие именно операции (действия) будут выполняться над данными. Так все ошибки нормализации станут очевидными, и Вы сможете увидеть, какие аномалии могут возникнуть в тех или иных случаях, и принимать решения о нормализации, иными словами, Вы должны руководствоваться здравым смыслом.

151. Какие есть типы связей в базе данных. Привести пример.

Ответ

Связи делятся на:

а) Многие ко многим

Представим, что нам нужно написать БД, которая будет хранить работников ИТ-компании. При этом существует некий стандартный набор должностей. При этом:

- Работник может иметь одну и более должностей. Например, некий работник может быть и админом, и программистом.
- Должность может «владеть» одним и более работников. Например, админами является определенный набор работников. Другими словами, к админам относятся некие работники.

Работников представляет таблица «employee» (id, имя, возраст), должности представляет таблица «Position» (id и название должности). Как видно, обе эти таблицы связаны между собой по правилу многие ко многим: каждому работнику соответствует одна и больше должностей (многие должности), каждой должности соответствует один и больше работников (многие работники).

Как построить такие таблицы?

Мы уже имеем две таблицы, описывающие работника и профессию. Теперь нам нужно установить между ними связь многие ко многим. Для реализации такой связи нам нужен некий посредник между таблицами «Employee» и «Positions». В нашем случае это будет некая таблица «employee_has_positions» (работники и должности). Эта таблица-посредник связывает между собой работника и должность следующим образом:

employee_id	positions_id
1	1
1	2
2	3
3	3

Слева указаны работники (их id), справа — должности (их id). Работники и должности на этой таблице указываются с помощью id'шников.

На эту таблицу можно посмотреть с двух сторон:

- Таким образом, мы говорим, что работник с id 1 находится на должность с id 1. При этом обратите внимание на то, что в этой таблице работник с id 1 имеет две должности: 1 и 2. Т.е., каждому работнику слева соответствует некая должность справа.
- Мы также можем сказать, что должности с id 3 принадлежат пользователи с id 2 и 3. Т.е., каждой роли справа принадлежит некий работник слева.

Реализация

```

CREATE TABLE IF NOT EXISTS employee (
  id BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  name VARCHAR(255),
  age BIGINT
) ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS positions (
  id BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  name VARCHAR(255)
) ENGINE=INNODB;

CREATE TABLE IF NOT EXISTS employee_has_positions (
  employee_id BIGINT,
  INDEX employee_id_ind (employee_id),
  FOREIGN KEY (employee_id)
    REFERENCES employee (id),
  positions_id BIGINT,
  INDEX positions_id_ind (positions_id),
  FOREIGN KEY (positions_id)
    REFERENCES positions (id)
) ENGINE=INNODB;

```

Вывод

Для реализации связи многие ко многим нам нужен некий посредник между двумя рассматриваемыми таблицами. Он должен хранить два внешних ключа, первый из которых ссылается на первую таблицу, а второй — на вторую.

Обязательные и необязательные связи

- Любая связь многие ко многим является необязательной.
- Например, Человек может инвестировать в акции разных компаний (многих). Инвесторами какой-то компании являются определенные люди (многие).
- Человек может вообще не инвестировать свои деньги в акции.
 - Акции компании мог никто не купить.

б) Один ко многим

- Предположим, нам нужно реализовать некую БД, которая ведет учет данных о пользователях. У пользователя есть: имя, фамилия, возраст, номера телефонов. При этом у каждого пользователя может быть от одного и больше номеров телефонов (многие номера телефонов).
- В этом случае мы наблюдаем следующее: пользователь может иметь многие номера телефонов, но нельзя сказать, что номеру телефона принадлежит определенный пользователь.
- Другими словами, телефон принадлежит только одному пользователю. А пользователю могут принадлежать 1 и более телефонов (многие).
- Как мы видим, это отношение один ко многим.

Как построить такие таблицы?

Пользователей будет представлять некая таблица «persons» (id, имя, фамилия, возраст), номера телефонов будет представлять таблица «phones». Она будет выглядеть так:

PhoneId	PersonId	PhoneNumber
1	5	11 091-10
2	5	19 124-66
3	17	21 972-02

Данная таблица представляет три номера телефона. При этом номера телефона с id 1 и 2 принадлежат пользователю с id 5. А вот номер с id 3 принадлежит пользователю с id 17.

Если бы у таблицы «Phones» было бы больше атрибутов, то мы смело бы их добавляли в эту таблицу.

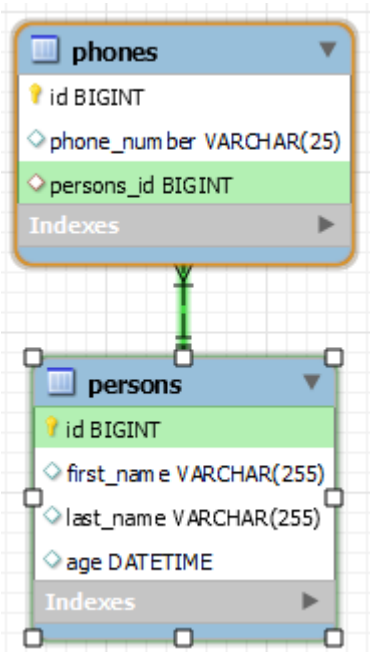
Почему мы не делаем тут таблицу-посредника?

Таблица-посредник нужна только в том случае, если мы имеем связь многие-ко-многим. По той простой причине, что мы можем рассматривать ее с двух сторон. Как, например, таблицу `employee_has_positions` ранее:

- Каждому работнику принадлежат несколько должностей (многие).
- Каждой должности принадлежит несколько работников (многие).

Но в нашем случае мы не можем сказать, что каждому телефону принадлежат несколько пользователей — номеру телефона может принадлежать только один пользователь.

Реализация



```
CREATE TABLE IF NOT EXISTS persons (  
    id BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,  
    first_name VARCHAR(255) DEFAULT NULL,  
    last_name VARCHAR(255) DEFAULT NULL,  
    age DATETIME DEFAULT NOW()  
) ENGINE=INNODB;
```

```
CREATE TABLE IF NOT EXISTS phones (  
    id BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,  
    phone_number VARCHAR(25) DEFAULT 'don\'t have phone number',  
    persons_id BIGINT,  
    INDEX persons_id_ind (persons_id),  
    FOREIGN KEY (persons_id)  
        REFERENCES persons (id)  
) ENGINE=INNODB;
```

Обязательные и необязательные связи

с обязательной связью	с необязательной связью
К одному полку относятся многие бойцы. Один боец относится только к одному полку. Обратите внимание, что любой солдат обязательно принадлежит к одному полку, а полк не может существовать без солдат.	На планете Земля живут все люди. Каждый человек живет только на Земле. При этом планета может существовать и без человечества. Соответственно, нахождение нас на Земле не является обязательным

Одну и ту же связь можно рассматривать как обязательную и как необязательную. Рассмотрим вот такой пример: У одной биологической матери может быть много детей. У ребенка есть только одна биологическая мать.

- У женщины необязательно есть свои дети. Соответственно, связь необязательна.
- У ребенка обязательно есть только одна биологическая мать – в таком случае, связь обязательна.

с) Один к одному

Представим, что на работе вам дали задание написать БД для учета всех работников для HR. Начальник уверял, что компании нужно знать только об имени, возрасте и телефоне работника. Вы разработали такую БД и поместили в нее всю 1000 работников компании. И тут начальник говорит, что им зачем-то нужно знать о том, является ли работник инвалидом или нет. Наиболее простое, что приходит в голову — это добавить новый столбец типа `bool` в вашу таблицу. Но это слишком долго вписывать 1000 значений и ведь `true` вы будете вписывать намного реже, чем `false` (2% будут `true`, например).

Более простым решением будет создать новую таблицу, назовем ее «DisabledEmployee». Она будет выглядеть так:

DisabledPersonId	EmployeeId
1	159
2	722
3	937

Но это еще не связь один к одному. Дело в том, что в такую таблицу работник может быть вписан более одного раза, соответственно, мы получили отношение один ко многим: работник может быть несколько раз инвалидом. Нужно сделать так, чтобы работник мог быть вписан в таблицу только один раз, соответственно, мог быть инвалидом только один раз. Для этого нам нужно указать, что столбец EmployeeId может хранить только уникальные значения. Нам нужно просто наложить на столбец EmployeeId ограничение unique. Это ограничение сообщает, что атрибут может принимать только уникальные значения.

Выполнив это, мы получили связь один к одному.

Обратите внимание на то, что мы могли также наложить на атрибут EmployeeId ограничение primary key. Оно отличается от ограничения unique лишь тем, что не может принимать значения null.

Вывод

Можно сказать, что отношение один к одному — это разделение одной и той же таблицы на две.

Обязательные и необязательные связи

с обязательной связью	с необязательной связью
У одного гражданина определенной страны обязательно есть только один паспорт этой страны. У одного паспорта есть только один владелец.	У одной страны может быть только одна конституция. Одна конституция принадлежит только одной стране. Но конституция не является обязательной. У страны она может быть, а может и не быть, как, например, у Израиля и Великобритании.

Одну и ту же связь можно рассматривать как обязательную и как необязательную:

У одного человека может быть только один загранпаспорт. У одного загранпаспорта есть только один владелец.

- Наличие загранпаспорта необязательно – его может и не быть у гражданина. Это необязательная связь.
- У загранпаспорта обязательно есть только один владелец. В этом случае, это уже обязательная связь.

Итоги

1. Связи бывают:

- Многие ко многим.
- Один ко многим.
 - с обязательной связью;
 - с необязательной связью.
- Один к одному.
 - с обязательной связью;
 - с необязательной связью.

2. Связи организуются с помощью внешних ключей.

3. Foreign key (внешний ключ) — это атрибут или набор атрибутов, которые ссылаются на primary key или unique другой таблицы. Другими словами, это что-то вроде указателя на строку другой таблицы.

152. Что такое primary key (первичный ключ)?

Ответ

Атрибут **PRIMARY KEY** задает первичный ключ таблицы.

```
CREATE TABLE IF NOT EXISTS persons (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255)
) ENGINE=INNODB;
```

Первичный ключ уникально идентифицирует строку в таблице. В качестве первичного ключа необязательно должны выступать столбцы с типом `int`, они могут представлять любой другой тип.

Установка первичного ключа на уровне таблицы:

```
CREATE TABLE IF NOT EXISTS persons (  
    id BIGINT AUTO_INCREMENT,  
    first_name VARCHAR(255),  
    last_name VARCHAR(255),  
    PRIMARY KEY (id)  
) ENGINE=INNODB;
```

Первичный ключ может быть составным. Такой ключ использовать сразу несколько столбцов, чтобы уникально идентифицировать строку в таблице. Например:

```
CREATE TABLE OrderLines (  
    order_id INT,  
    product_id INT,  
    quantity INT,  
    PRIMARY KEY (order_id , product_id)  
) ENGINE=INNODB;
```

Здесь поля `OrderId` и `ProductId` вместе выступают как составной первичный ключ. То есть в таблице `OrderLines` не может быть двух строк, где для обоих из этих полей одновременно были бы одни и те же значения.

153. Что такое foreign key (внешний ключ)?

Ответ

FOREIGN KEY

Внешние ключи позволяют установить связи между таблицами. Внешний ключ устанавливается для столбцов из зависимой, подчиненной таблицы, и указывает на один из столбцов из главной таблицы. Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы.

Для создания ограничения внешнего ключа после **FOREIGN KEY** указывается столбец таблицы, который будет представляет внешний ключ. А после ключевого слова **REFERENCES** указывается имя связанной таблицы, а затем в скобках имя связанного столбца, на который будет указывать внешний ключ. После выражения **REFERENCES** идут выражения **ON DELETE** и **ON UPDATE**, которые задают действие при удалении и обновлении строки из главной таблицы соответственно.

Например, определим две таблицы и свяжем их посредством внешнего ключа:

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    age INT,  
    first_name VARCHAR(20) NOT NULL,  
    last_name VARCHAR(20) NOT NULL,  
    phone VARCHAR(20) NOT NULL UNIQUE  
) ENGINE=INNODB;  
  
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    customer_id INT,  
    created_at DATE,  
    FOREIGN KEY (customer_id)  
        REFERENCES customers (id)  
) ENGINE=INNODB;
```

В данном случае определены таблицы `customers` и `orders`. Таблица `customers` является главной и представляет клиента. Таблица `orders` является зависимой и представляет заказ, сделанный клиентом. Таблица `orders` через столбец `customer_id` связана с таблицей `customers` и ее столбцом `id`. То есть столбец `customer_id` является внешним ключом, который указывает на столбец `id` из таблицы `customers`.

С помощью оператора **CONSTRAINT** можно задать имя для ограничения внешнего ключа:

```
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    customer_id INT,  
    created_at DATE,  
    CONSTRAINT orders_customers_fk FOREIGN KEY (customer_id)
```

```
REFERENCES customers (id)
) ENGINE=INNODB;
```

ON DELETE и ON UPDATE

С помощью выражений **ON DELETE** и **ON UPDATE** можно установить действия, которые выполняются соответственно при удалении и изменении связанной строки из главной таблицы. В качестве действия могут использоваться следующие опции:

- **CASCADE**

CASCADE – автоматически удаляет или изменяет строки из зависимой таблицы при удалении или изменении связанных строк в главной таблице.

- **SET NULL**

SET NULL – при удалении или обновлении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение NULL. (В этом случае столбец внешнего ключа должен поддерживать установку NULL)

- **RESTRICT**

RESTRICT – отклоняет удаление или изменение строк в главной таблице при наличии связанных строк в зависимой таблице.

- **NO ACTION**

NO ACTION – то же самое, что и **RESTRICT**.

- **SET DEFAULT**

SET DEFAULT – при удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение по умолчанию, которое задается с помощью атрибута DEFAULT. Несмотря на то, что данная опция в принципе доступна, однако движок InnoDB не поддерживает данное выражение.

Каскадное удаление

Каскадное удаление позволяет при удалении строки из главной таблицы автоматически удалить все связанные строки из зависимой таблицы. Для этого применяется опция **CASCADE**:

```
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    created_at DATE,
    FOREIGN KEY (customer_id)
        REFERENCES customers (id)
        ON DELETE CASCADE
) ENGINE=INNODB;
```

Подобным образом работает и выражение **ON UPDATE CASCADE**. При изменении значения первичного ключа автоматически изменится значение связанного с ним внешнего ключа. Однако поскольку первичные ключи изменяются очень редко, да и в принципе не рекомендуется использовать в качестве первичных ключей столбцы с изменяемыми значениями, то на практике выражение **ON UPDATE** используется редко.

Установка NULL

При установке для внешнего ключа опции **SET NULL** необходимо, чтобы столбец внешнего ключа допускал значение NULL:

```
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    created_at DATE,
    FOREIGN KEY (customer_id)
        REFERENCES customers (id)
        ON DELETE SET NULL
) ENGINE=INNODB;
```

154. Что такое индексы в базе данных? Для чего их используют? Чем они хороши и чем плохи?

Ответ

Что такое индексы в базе данных?

Индексы – это специальные таблицы, которые могут быть использованы поисковым двигателем базы данных, для ускорения получения данных. Необходимо просто добавить указатель индекса в таблицу.

Индекс в БД крайне схож с индексом в конце книги.

Допустим, мы хотим иметь ссылку на все страницы книги, которые касаются определённой темы (например, Наследование в книге по программированию на языке Java). Для этого, мы в первую очередь ссылаемся на индекс, который указан в конце книги и переходим на любую из страниц, которая относится к необходимой теме.

Для чего их используют?

Индекс помогает ускорить запросы на получение данных (`SELECT [WHERE]`), но замедляет процесс добавления и изменения записей (`INSERT, UPDATE`). Индексы могут быть добавлены или удалены без влияния на сами данные.

Для того, чтобы добавить индекс, нам необходимо использовать команду **CREATE INDEX**, что позволит нам указать имя индекса и определить таблицу и колонку или индекс колонки и определить используется ли индекс по возрастанию или по убыванию.

Индексы также могут быть уникальными, так же, как и констрейнт **UNIQUE**. В этом случае индекс предотвращает добавление повторяющихся данных в колонку или комбинацию колонок, на которые указывает индекс.

Добавление индекса

Команда добавления индекса имеет следующий вид:

```
CREATE INDEX имя_индекса ON имя_таблицы;
```

Индекс может относиться, как к одной колонке:

```
CREATE INDEX имя_индекса  
ON имя_таблицы (имя_колонки);
```

Так и к нескольким:

```
CREATE INDEX имя_индекса  
ON имя_таблицы (колонка1, колонка2);
```

Выбор типа индекса (одноколоночный или многоколоночный) зависит от того, что именно мы чаще всего будем использовать в нашем условном операторе **WHERE**.

Удаление индекса

Для того, чтобы удалить индекс, мы должны использовать следующую команду:

```
DROP INDEX имя_индекса ON имя_таблицы;
```

Или

```
ALTER TABLE имя_таблицы  
DROP INDEX имя_индекса;
```

Чем они хороши и чем плохи?

Индекс помогает ускорить запросы на получение данных (`SELECT [WHERE]`), но замедляет процесс добавления и изменения записей (`INSERT, UPDATE`).

Когда следует использовать индексы

Индексы используются для увеличения производительности БД, но есть случаи, когда нам стоит избегать их использования:

- Не стоит использовать индексы для небольших таблиц.
- Не стоит использовать индексы для таблиц, в которых, как предполагается, будут часто добавляться новые данные, либо эти данные будут изменяться.
- Не стоит использовать индекс для колонок, с которыми будут производиться частые манипуляции.
- Не стоит использовать индексы для колонок, которые имеют много значений **NULL**.

155. Какие есть типы **JOIN**'ов. Кратко опишите каждый из типов.

Ответ

Исходные данные

Таблица с наименованием товаров (**nomenclature**), будет хранить номер товара (**id**) и краткое название (**name**).
Содержание таблицы товары (**nomenclature**):

<pre>CREATE TABLE IF NOT EXISTS nomenclature (id BIGINT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255)) ENGINE=INNODB;</pre>	<pre>CREATE TABLE IF NOT EXISTS descriptions (id BIGINT AUTO_INCREMENT PRIMARY KEY, description VARCHAR(255)) ENGINE=INNODB;</pre>																
<pre>insert into nomenclature(id, name) values(1, 'книга'), (2, 'табуретка'), (3, 'карандаш');</pre>	<pre>insert into descriptions(id, description) values(1, 'замечательная книга'), (3, 'красный карандаш'), (5, 'зелёная машинка');</pre>																
<pre>SELECT * FROM nomenclature;</pre>	<pre>SELECT * FROM descriptions;</pre>																
Содержание таблицы товары (<i>nomenclature</i>):	Содержание таблицы с описанием товаров (<i>descriptions</i>):																
<table border="1"> <thead> <tr><th>id</th><th>name</th></tr> </thead> <tbody> <tr><td>1</td><td>книга</td></tr> <tr><td>2</td><td>табуретка</td></tr> <tr><td>3</td><td>карандаш</td></tr> </tbody> </table>	id	name	1	книга	2	табуретка	3	карандаш	<table border="1"> <thead> <tr><th>id</th><th>description</th></tr> </thead> <tbody> <tr><td>1</td><td>замечательная книга</td></tr> <tr><td>3</td><td>красный карандаш</td></tr> <tr><td>5</td><td>зелёная машинка</td></tr> </tbody> </table>	id	description	1	замечательная книга	3	красный карандаш	5	зелёная машинка
id	name																
1	книга																
2	табуретка																
3	карандаш																
id	description																
1	замечательная книга																
3	красный карандаш																
5	зелёная машинка																

Таблица *nomenclature* содержит перечень **всех** товаров, которые есть в базе. Таблица описаний *descriptions*, напротив, содержит лишь **неполный перечень** описаний для товаров, которые необязательно присутствуют в базе. Чтобы однозначно привязать описание к товару, в таблицах присутствует столбец *id*, который содержит уникальный номер товара. В обеих таблицах *id* является первичным ключом, что соответствует связи один-к-одному.

Выборка

В зависимости от требований к результату, MySQL позволяет производить три разных типа объединения:

- *INNER JOIN* (*CROSS JOIN*) - внутреннее (перекрёстное) объединение

Этот тип объединения позволяет извлекать строки, которые обязательно присутствуют во всех объединяемых таблицах.

В простейшем случае (без указания условий отбора), выборка вернёт так называемое декартово произведение, в котором каждая строка одной таблицы будет сопоставлена с каждой строкой другой таблицы:

Запрос	Результат запроса																																								
<pre>SELECT * FROM nomenclature INNER JOIN descriptions;</pre>	<table><tr><th>id</th><th>name</th><th>nomendature_id</th><th>description</th></tr><tr><td>3</td><td>карандаш</td><td>1</td><td>замечательная книга</td></tr><tr><td>2</td><td>табуретка</td><td>1</td><td>замечательная книга</td></tr><tr><td>1</td><td>книга</td><td>1</td><td>замечательная книга</td></tr><tr><td>3</td><td>карандаш</td><td>3</td><td>красный карандаш</td></tr><tr><td>2</td><td>табуретка</td><td>3</td><td>красный карандаш</td></tr><tr><td>1</td><td>книга</td><td>3</td><td>красный карандаш</td></tr><tr><td>3</td><td>карандаш</td><td>5</td><td>зелёная машинка</td></tr><tr><td>2</td><td>табуретка</td><td>5</td><td>зелёная машинка</td></tr><tr><td>1</td><td>книга</td><td>5</td><td>зелёная машинка</td></tr></table>	id	name	nomendature_id	description	3	карандаш	1	замечательная книга	2	табуретка	1	замечательная книга	1	книга	1	замечательная книга	3	карандаш	3	красный карандаш	2	табуретка	3	красный карандаш	1	книга	3	красный карандаш	3	карандаш	5	зелёная машинка	2	табуретка	5	зелёная машинка	1	книга	5	зелёная машинка
id	name	nomendature_id	description																																						
3	карандаш	1	замечательная книга																																						
2	табуретка	1	замечательная книга																																						
1	книга	1	замечательная книга																																						
3	карандаш	3	красный карандаш																																						
2	табуретка	3	красный карандаш																																						
1	книга	3	красный карандаш																																						
3	карандаш	5	зелёная машинка																																						
2	табуретка	5	зелёная машинка																																						
1	книга	5	зелёная машинка																																						

Как правило, декартово произведение таблиц требуется нечасто, чаще требуется выбрать только те записи, которые сопоставлены друг другу. Сделать это можно, если задать условие отбора, используя *ON* или *USING*.

Запрос	Результат запроса									
<pre>SELECT * FROM nomenclature INNER JOIN descriptions USING(id);</pre>	<table><tr><th>id</th><th>name</th><th>description</th></tr><tr><td>1</td><td>книга</td><td>замечательная книга</td></tr><tr><td>3</td><td>карандаш</td><td>красный карандаш</td></tr></table>	id	name	description	1	книга	замечательная книга	3	карандаш	красный карандаш
id	name	description								
1	книга	замечательная книга								
3	карандаш	красный карандаш								

Запрос вернул только две записи, поскольку именно столько строк имеют одинаковые идентификаторы в обеих таблицах.

Использование **USING** обусловлено тем, что в таблицах ключевой столбец имеет одно и тоже имя - **id**. В противном случае, надо было бы использовать **ON**.

Помимо конструкции **INNER JOIN** внутреннее объединение можно объявить так же через **CROSS JOIN**, **JOIN** и запятую в объявлении **FROM**. Следующие четыре запроса вернут одинаковый результат:

Запрос	Результат запроса																																								
<pre>SELECT * FROM nomenclature INNER JOIN descriptions;</pre>	<table><tr><th>id</th><th>name</th><th>id</th><th>description</th></tr><tr><td>3</td><td>карандаш</td><td>1</td><td>замечательная книга</td></tr><tr><td>2</td><td>табуретка</td><td>1</td><td>замечательная книга</td></tr><tr><td>1</td><td>книга</td><td>1</td><td>замечательная книга</td></tr><tr><td>3</td><td>карандаш</td><td>3</td><td>красный карандаш</td></tr><tr><td>2</td><td>табуретка</td><td>3</td><td>красный карандаш</td></tr><tr><td>1</td><td>книга</td><td>3</td><td>красный карандаш</td></tr><tr><td>3</td><td>карандаш</td><td>5</td><td>зелёная машинка</td></tr><tr><td>2</td><td>табуретка</td><td>5</td><td>зелёная машинка</td></tr><tr><td>1</td><td>книга</td><td>5</td><td>зелёная машинка</td></tr></table>	id	name	id	description	3	карандаш	1	замечательная книга	2	табуретка	1	замечательная книга	1	книга	1	замечательная книга	3	карандаш	3	красный карандаш	2	табуретка	3	красный карандаш	1	книга	3	красный карандаш	3	карандаш	5	зелёная машинка	2	табуретка	5	зелёная машинка	1	книга	5	зелёная машинка
id	name	id	description																																						
3	карандаш	1	замечательная книга																																						
2	табуретка	1	замечательная книга																																						
1	книга	1	замечательная книга																																						
3	карандаш	3	красный карандаш																																						
2	табуретка	3	красный карандаш																																						
1	книга	3	красный карандаш																																						
3	карандаш	5	зелёная машинка																																						
2	табуретка	5	зелёная машинка																																						
1	книга	5	зелёная машинка																																						
<pre>SELECT * FROM nomenclature CROSS JOIN descriptions;</pre>																																									
<pre>SELECT * FROM nomenclature JOIN descriptions;</pre>																																									
<pre>SELECT * FROM nomenclature, descriptions;</pre>																																									

Если объединять таблицы через запятую, то нельзя использовать конструкции **ON** и **USING**, поэтому условие может быть задано только в конструкции **WHERE**. Например, это может выглядеть так:

Запрос	Результат запроса												
<pre>SELECT * FROM nomenclature, descriptions WHERE nomenclature.id = descriptions.id;</pre>	<table><tr><th>id</th><th>name</th><th>id</th><th>description</th></tr><tr><td>1</td><td>книга</td><td>1</td><td>замечательная книга</td></tr><tr><td>3</td><td>карандаш</td><td>3</td><td>красный карандаш</td></tr></table>	id	name	id	description	1	книга	1	замечательная книга	3	карандаш	3	красный карандаш
id	name	id	description										
1	книга	1	замечательная книга										
3	карандаш	3	красный карандаш										

Поскольку поле **id** не является однозначным, приходится до уточнять в каком контексте оно используется через указание имени таблицы.

Внутреннее объединение можно задать следующими способами:

```
SELECT * FROM Таблица1, Таблица2[, Таблица3, ...] [WHERE Условие1 [Условие2 ...]]
SELECT * FROM Таблица1 [INNER | CROSS] JOIN Таблица2 [(ON Условие1 [Условие2 ...]) | (USING (Поле))]
```

Результатом будет декартово произведение всех таблиц, на которое можно накладывать условия выборки, используя **ON**, **USING** и **WHERE**.

- **LEFT JOIN** - левостороннее внешнее объединение

Левосторонние объединения позволяют извлекать данные из таблицы, дополняя их по возможности данными из другой таблицы.

К примеру, чтобы получить полный список наименований товаров вместе с их описанием, нужно выполнить следующий запрос:

Запрос	Результат запроса												
<pre>SELECT * FROM nomenclature LEFT JOIN descriptions USING(id);</pre>	<table><tr><th>id</th><th>name</th><th>description</th></tr><tr><td>1</td><td>книга</td><td>замечательная книга</td></tr><tr><td>2</td><td>табуретка</td><td>NULL</td></tr><tr><td>3</td><td>карандаш</td><td>красный карандаш</td></tr></table>	id	name	description	1	книга	замечательная книга	2	табуретка	NULL	3	карандаш	красный карандаш
id	name	description											
1	книга	замечательная книга											
2	табуретка	NULL											
3	карандаш	красный карандаш											

Поскольку для наименования **табуретка** в таблице описаний нет подходящей записи, то в поле **descriptions** подставился **NULL**. Это справедливо для всех записей, у которых нет подходящей пары.

Если дополнить предыдущий запрос условием на проверку не существования описания, то можно получить список записей, которые не имеют пары в таблице описаний:

Запрос	Результат запроса				
<pre>SELECT id, name FROM nomenclature LEFT JOIN descriptions USING(id) WHERE description IS NULL;</pre>	<table> <tr> <th>id</th><th>name</th></tr> <tr> <td>2</td><td>табуретка</td></tr> </table>	id	name	2	табуретка
id	name				
2	табуретка				

По сути это и есть основное назначение внешних запросов - показывать расхождение данных двух таблиц.

Кроме того, при таком объединении обязательным является условие, которое задаётся через **ON** или **USING**. Без него запрос будет выдавать ошибку.

- **RIGHT JOIN** - правостороннее внешнее объединение

Этот вид объединений практически ничем не отличается от левостороннего объединения, за тем исключением, что данные берутся из второй таблицы, которая находится **справа** от конструкции **JOIN**, и сравниваются с данными, которые находятся в таблице, указанной перед конструкцией.

Запрос	Результат запроса												
<pre>SELECT * FROM nomenclature RIGHT JOIN descriptions USING(id);</pre>	<table><tr><th>id</th><th>description</th><th>name</th></tr><tr><td>1</td><td>замечательная книга</td><td>книга</td></tr><tr><td>3</td><td>красный карандаш</td><td>карандаш</td></tr><tr><td>5</td><td>зелёная машинка</td><td>NULL</td></tr></table>	id	description	name	1	замечательная книга	книга	3	красный карандаш	карандаш	5	зелёная машинка	NULL
id	description	name											
1	замечательная книга	книга											
3	красный карандаш	карандаш											
5	зелёная машинка	NULL											

Как видно, теперь уже поле **name** содержит нулевые значения. Также поменялся и порядок расположения столбцов.

Однако, во всех случаях использования правосторонних объединений, запрос можно переписать, используя левостороннее объединение, просто поменяв таблицы местами, и наоборот. Следующие два запроса равнозначны:

Запрос	Результат запроса												
<pre>SELECT * FROM nomenclature LEFT JOIN descriptions USING(id);</pre>	<table><tr><th>id</th><th>name</th><th>description</th></tr><tr><td>1</td><td>книга</td><td>замечательная книга</td></tr><tr><td>2</td><td>табуретка</td><td>NULL</td></tr><tr><td>3</td><td>карандаш</td><td>красный карандаш</td></tr></table>	id	name	description	1	книга	замечательная книга	2	табуретка	NULL	3	карандаш	красный карандаш
id	name	description											
1	книга	замечательная книга											
2	табуретка	NULL											
3	карандаш	красный карандаш											
<pre>SELECT * FROM descriptions RIGHT JOIN nomenclature USING(id);</pre>													

Многотабличные запросы

Используя JOIN, можно объединять не только две таблицы, но и гораздо больше. В MySQL можно объединить до 61 таблицы. Помимо объединений разных таблиц, MySQL позволяет объединять таблицу саму с собой. Однако необходимо следить за именами столбцов и таблиц, если они будут неоднозначны, то запрос не будет выполнен.

Если таблицу просто объединить саму на себя, то возникнет конфликт имён и запрос не выполнится.

Запрос	Результат запроса
<pre>SELECT * FROM nomenclature JOIN nomenclature;</pre>	Error Code: 1066. Not unique table/alias: 'nomenclature'

Обойти конфликт имён позволяет использование синонимов (**alias**) для имён таблиц и столбцов. В следующем примере внутреннее объединение будет работать успешнее:

Запрос	Результат запроса																																								
<pre>SELECT * FROM nomenclature JOIN nomenclature AS t2;</pre>	<table><tr><th>id</th><th>name</th><th>id</th><th>name</th></tr><tr><td>3</td><td>карандаш</td><td>1</td><td>книга</td></tr><tr><td>2</td><td>табуретка</td><td>1</td><td>книга</td></tr><tr><td>1</td><td>книга</td><td>1</td><td>книга</td></tr><tr><td>3</td><td>карандаш</td><td>2</td><td>табуретка</td></tr><tr><td>2</td><td>табуретка</td><td>2</td><td>табуретка</td></tr><tr><td>1</td><td>книга</td><td>2</td><td>табуретка</td></tr><tr><td>3</td><td>карандаш</td><td>3</td><td>карандаш</td></tr><tr><td>2</td><td>табуретка</td><td>3</td><td>карандаш</td></tr><tr><td>1</td><td>книга</td><td>3</td><td>карандаш</td></tr></table>	id	name	id	name	3	карандаш	1	книга	2	табуретка	1	книга	1	книга	1	книга	3	карандаш	2	табуретка	2	табуретка	2	табуретка	1	книга	2	табуретка	3	карандаш	3	карандаш	2	табуретка	3	карандаш	1	книга	3	карандаш
id	name	id	name																																						
3	карандаш	1	книга																																						
2	табуретка	1	книга																																						
1	книга	1	книга																																						
3	карандаш	2	табуретка																																						
2	табуретка	2	табуретка																																						
1	книга	2	табуретка																																						
3	карандаш	3	карандаш																																						
2	табуретка	3	карандаш																																						
1	книга	3	карандаш																																						

MySQL не накладывает ограничений на использование разных типов объединений в одном запросе, поэтому можно формировать довольно сложные конструкции:

Запрос	Результат запроса																																																												
<pre>SELECT * FROM nomenclature AS t1 JOIN nomenclature AS t2 LEFT JOIN nomenclature AS t3 ON t1.id = t3.id AND t2.id = t1.id;</pre>	<table><tr><th>id</th><th>name</th><th>id</th><th>name</th><th>id</th><th>name</th></tr><tr><td>3</td><td>карандаш</td><td>1</td><td>книга</td><td>NULL</td><td>NULL</td></tr><tr><td>2</td><td>табуретка</td><td>1</td><td>книга</td><td>NULL</td><td>NULL</td></tr><tr><td>1</td><td>книга</td><td>1</td><td>книга</td><td>1</td><td>книга</td></tr><tr><td>3</td><td>карандаш</td><td>2</td><td>табуретка</td><td>NULL</td><td>NULL</td></tr><tr><td>2</td><td>табуретка</td><td>2</td><td>табуретка</td><td>2</td><td>табуретка</td></tr><tr><td>1</td><td>книга</td><td>2</td><td>табуретка</td><td>NULL</td><td>NULL</td></tr><tr><td>3</td><td>карандаш</td><td>3</td><td>карандаш</td><td>3</td><td>карандаш</td></tr><tr><td>2</td><td>табуретка</td><td>3</td><td>карандаш</td><td>NULL</td><td>NULL</td></tr><tr><td>1</td><td>книга</td><td>3</td><td>карандаш</td><td>NULL</td><td>NULL</td></tr></table>	id	name	id	name	id	name	3	карандаш	1	книга	NULL	NULL	2	табуретка	1	книга	NULL	NULL	1	книга	1	книга	1	книга	3	карандаш	2	табуретка	NULL	NULL	2	табуретка	2	табуретка	2	табуретка	1	книга	2	табуретка	NULL	NULL	3	карандаш	3	карандаш	3	карандаш	2	табуретка	3	карандаш	NULL	NULL	1	книга	3	карандаш	NULL	NULL
id	name	id	name	id	name																																																								
3	карандаш	1	книга	NULL	NULL																																																								
2	табуретка	1	книга	NULL	NULL																																																								
1	книга	1	книга	1	книга																																																								
3	карандаш	2	табуретка	NULL	NULL																																																								
2	табуретка	2	табуретка	2	табуретка																																																								
1	книга	2	табуретка	NULL	NULL																																																								
3	карандаш	3	карандаш	3	карандаш																																																								
2	табуретка	3	карандаш	NULL	NULL																																																								
1	книга	3	карандаш	NULL	NULL																																																								

Помимо выборок использовать объединения можно также и в запросах **UPDATE** и **DELETE**.

156. Для чего используется слово HAVING? Отличия от WHERE.

Ответ

В сущности, **HAVING** очень похож на **WHERE** - это тоже фильтр. Можно написать в **HAVING** `name = 'Anna'`, как и в **WHERE**, и ошибки не будет.

В чём же ключевое различие?

В **HAVING** и только в нём можно писать условия по агрегатным функциям (**SUM**, **COUNT**, **MAX**, **MIN** и т. д.).

То есть если надо сделать что-то вроде `COUNT() > 10`, то это возможно сделать только в **HAVING**.

Главное отличие **HAVING** от **WHERE** в том, что в **HAVING** можно наложить условия на результаты группировки, потому что порядок исполнения запроса устроен таким образом, что на этапе, когда выполняется **WHERE**, ещё нет групп, а **HAVING** выполняется уже после формирования групп.

Как работают HAVING и WHERE?

Всё кроется в том, как **SQL Server** выполняет запрос, в каком порядке происходит его разбор и работа с данными. **WHERE** выполняется до формирования групп **GROUP BY**. Это нужно для того, чтобы можно было оперировать как можно меньшим количеством данных и сэкономить ресурсы сервера и время пользователя.

Следующим этапом формируются группы, которые указаны в **GROUP BY**. После того как сформированы группы, можно накладывать условия на результаты агрегатных функций. И тут как раз наступает очередь **HAVING**: выполняются условия, которые вы задали.

157. Зачем нужно View и какие поля там будут.

Ответ

Что такое Представление (VIEW)?

Представление (VIEW) — объект базы данных, являющийся результатом выполнения запроса к базе данных, определенного с помощью оператора **SELECT**, в момент обращения к представлению.

VIEW иногда называют «виртуальными таблицами». Такое название связано с тем, что **VIEW** доступно для пользователя как таблица, но само оно не содержит данных, а извлекает их из таблиц в момент обращения к нему. Если данные изменены в базовой таблице, то пользователь получит актуальные данные при обращении ко **VIEW**, использующему данную таблицу; кэширования результатов выборки из таблицы при работе **VIEW** не производится. При этом, механизм кэширования запросов (`query cache`) работает на уровне запросов пользователя безотносительно к тому, обращается ли пользователь к таблицам или **VIEW**.

VIEW могут основываться как на таблицах, так и на других **VIEW**, т.е. могут быть вложенными (до 32 уровней вложенности).

Создание view

Для создания представления используется оператор `CREATE VIEW`, имеющий следующий синтаксис:

```
CREATE [OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

`view_name` — имя создаваемого представления. `select_statement` — оператор `SELECT`, выбирающий данные из таблиц и/или других представлений, которые будут содержаться в представлении

Оператор **CREATE VIEW** содержит 4 необязательные конструкции:

a) **OR REPLACE** — при использовании данной конструкции в случае существования `view` с таким именем старое будет удалено, а новое создано. В противном случае возникнет ошибка, информирующая о существовании `view` с таким именем и новое представление создано не будет. Следует отметить одну особенность — имена таблиц и `view` в рамках одной базы данных должны быть уникальны, т.е. нельзя создать `view` с именем уже существующей таблицы. Однако конструкция **OR REPLACE** действует только на `view` и замещать таблицу не будет.

b) **ALGORITHM** — определяет алгоритм, используемый при обращении к `view`.

c) **column_list** — задает имена полей `view`.

d) **WITH CHECK OPTION** — при использовании данной конструкции все добавляемые или изменяемые строки будут проверяться на соответствие определению представления. В случае несоответствия данное изменение не будет выполнено. Обратите внимание, что при указании данной конструкции для не обновляемого `view` возникнет ошибка и `view` не будет создано.

Колонки view

По умолчанию колонки `view` имеют те же имена, что и поля, возвращаемые оператором `SELECT` в определении `view`. При явном указании имен полей `view column_list` должен включать по одному имени для каждого поля разделенных запятой.

Вызов view

```
SELECT * FROM view_name;
```

158. Есть таблицы `Customer` и `Order`. Вывести всех `customer`, у которых суммарный заказ будет > 10000.

Ответ

Условие (таблицы)

```
CREATE TABLE customers (
  id BIGINT AUTO_INCREMENT PRIMARY KEY
) ENGINE=INNODB;
```

```
CREATE TABLE orders (
  customers_id BIGINT AUTO_INCREMENT
  PRIMARY KEY,
  INDEX customers_id_ind (customers_id),
  FOREIGN KEY (customers_id)
    REFERENCES customers (id),
  sum DOUBLE DEFAULT 0
) ENGINE=INNODB;
```

Решение

```
SELECT customers.id, orders.sum
FROM customers INNER JOIN orders
ON customers.id = orders.customers_id
WHERE orders.sum > 10000;
```

159. Что такое агрегирующая функция, примеры.

Ответ

Агрегатная функция выполняет вычисление над набором значений и возвращает одно значение. В табличной модели данных это значит, что функция берет ноль, одну или несколько строк для какой-то колонки и возвращает единственное значение.

Агрегатные функции используют с операторами `GROUP BY` и `HAVING`:

- **GROUP BY** — группирует строки с одинаковыми значениями в одну строку;

- **HAVING** — используется как фильтр для запросов, в которых есть оператор GROUP BY. С агрегатными функциями можно использовать ключевые слова DISTINCT и ALL. Агрегирующие функции:

- **COUNT**

COUNT считает количество строк в таблице. Она может принимать в качестве параметров как числовые, так и нечисловые типы данных.

COUNT (*) — специальная форма функции COUNT, которая возвращает количество всех строк в указанной таблице. COUNT (*) считает дубликат и NULL.

Пример:

Напишем запрос, который будет считать количество сотрудников для каждой роли:

Запрос	Результат	
<pre>SELECT role, COUNT(*) number_of_employee FROM employees GROUP BY role;</pre>	role	number_of _employee
	QA	1
	Manager	2
	SWE	2
Запрос	Результат	
<pre>SELECT COUNT(DISTINCT office_id) AS unique_offices FROM employees;</pre>	unique_offices	
	2	

- **SUM**

Функция SUM вычисляет суммы всех выбранных столбцов. Она работает только с числовыми полями.

Пример:

Посчитать суммарную зарплату для всех сотрудников.

Запрос	Результат	
<pre>SELECT SUM(salary) AS total_salary FROM employees;</pre>	total_salary	
	2750	

- **MAX и MIN**

Эти функции нужны для нахождения максимального и минимального значения для определенного столбца.

Примеры:

Минимальная зарплата среди всех сотрудников:

Запрос	Результат	
<pre>SELECT MIN(salary) AS min_salary FROM employees;</pre>	min_salary	
	500	

Максимальная зарплата для каждого офиса, где больше двух сотрудников:

Запрос	Результат	
<pre>SELECT office_id, MAX(salary) AS max_salary FROM employees GROUP BY office_id HAVING COUNT(*) > 2;</pre>	office_id	max_salary
	2	750

- **AVG**

Функция используется для вычисления среднего значения заданного столбца.

Рассчитаем среднюю зарплату по всем сотрудникам:

Запрос	Результат	
<pre>SELECT AVG(salary) AS avg_salary FROM employees;</pre>	avg_salary	
	687.5000000000000	

Функция AVG игнорирует значение NULL.

160. Результат запроса *Select * from Table1, Table2;*

Ответ

Покажет всё что есть из двух таблиц.

161. Есть 2 таблицы

IDDepartment	Department	
IDEmployer	Salary	IDDepartment

- Написать запросы на выборку Department, salary если сумма salary>100;
- Написать запрос на ту же выборку через join;

162. Атрибуты столбцов и таблиц

Ответ

С помощью атрибутов можно настроить поведение столбцов.

Атрибут PRIMARY KEY

Вопрос 152.

Атрибут AUTO_INCREMENT

Атрибут **AUTO_INCREMENT** позволяет указать, что значение столбца будет автоматически увеличиваться при добавлении новой строки. Данный атрибут работает для столбцов, которые представляют целочисленный тип или числа с плавающей точкой.

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    age INT,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20)  
) ENGINE=INNODB;
```

В данном случае значение столбца id каждой новой добавленной строки будет увеличиваться на единицу.

Атрибут UNIQUE

Атрибут **UNIQUE** указывает, что столбец может хранить только уникальные значения.

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    age INT,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20),  
    phone VARCHAR(13) UNIQUE  
) ENGINE=INNODB;
```

В данном случае столбец phone, который представляет телефон клиента, может хранить только уникальные значения. И мы не сможем добавить в таблицу две строки, у которых значения для этого столбца будут совпадать.

Также мы можем определить этот атрибут на уровне таблицы:

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    age INT,  
    first_name VARCHAR(20),  
    last_name VARCHAR(20),  
    email VARCHAR(30),  
    phone VARCHAR(20),  
)
```

```

        UNIQUE (email , phone)
    ) ENGINE=INNODB;

```

NULL и NOT NULL

Чтобы указать, может ли столбец принимать значение **NULL**, при определении столбца ему можно задать атрибут **NULL** или **NOT NULL**. Если этот атрибут явным образом не будет использован, то по умолчанию столбец будет допускать значение **NULL**. Исключением является тот случай, когда столбец выступает в роли первичного ключа - в этом случае по умолчанию столбец имеет значение **NOT NULL**.

```

CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    age INT,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    email VARCHAR(30) NULL,
    phone VARCHAR(20) NULL
) ENGINE=INNODB;

```

В данном случае столбец age по умолчанию будет иметь атрибут NULL.

Атрибут DEFAULT

Атрибут **DEFAULT** определяет значение по умолчанию для столбца. Если при добавлении данных для столбца не будет предусмотрено значение, то для него будет использоваться значение по умолчанию.

```

CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    age INT DEFAULT 18,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    email VARCHAR(30) NOT NULL UNIQUE,
    phone VARCHAR(20) NOT NULL UNIQUE
) ENGINE=INNODB;

```

Здесь столбец age в качестве значения по умолчанию имеет число 18.

Атрибут CHECK

Атрибут **CHECK** задает ограничение для диапазона значений, которые могут храниться в столбце. Для этого после **CHECK** указывается в скобках условие, которому должен соответствовать столбец или несколько столбцов. Например, возраст клиентов не может быть меньше 0 или больше 100:

```

CREATE TABLE customers (
    id INT AUTO_INCREMENT,
    age INT DEFAULT 18 CHECK (age > 0 AND age < 100),
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    email VARCHAR(30) CHECK (email != ''),
    phone VARCHAR(20) CHECK (phone != '')
) ENGINE=INNODB;

```

Кроме проверки возраста здесь также проверяется, что столбцы email и phone не могут иметь пустую строку в качестве значения (пустая строка не эквивалентна значению NULL).

Ключевое слово AND

Для соединения условий используется ключевое слово **AND**. Условия можно задать в виде операций сравнения больше (>), меньше (<), не равно (!=).

Также CHECK можно использовать на уровне таблицы:

```

CREATE TABLE customers (
    id INT AUTO_INCREMENT,
    age INT DEFAULT 18,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    email VARCHAR(30),

```

```

        phone VARCHAR(20),
        CHECK ((age > 0 AND age < 100)
              AND (email != '')
              AND (phone != ''))
    ) ENGINE=INNODB;

```

Оператор **CONSTRAINT**. Установка имени ограничений

С помощью ключевого слова **CONSTRAINT** можно задать имя для ограничений. Они указываются после ключевого слова **CONSTRAINT** перед атрибутами на уровне таблицы:

```

CREATE TABLE customers (
    id INT AUTO_INCREMENT,
    age INT,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    email VARCHAR(30),
    phone VARCHAR(20) NOT NULL,
    CONSTRAINT customers_pk PRIMARY KEY (id),
    CONSTRAINT customer_phone_uq UNIQUE (phone),
    CONSTRAINT customer_age_chk CHECK (age > 0 AND age < 100)
) ENGINE=INNODB;

```

В данном случае ограничение для PRIMARY KEY называется customers_pk, для UNIQUE - customer_phone_uq, а для CHECK - customer_age_chk. Смысл установки имен ограничений заключается в том, что впоследствии через эти имена мы сможем управлять ограничениями - удалять или изменять их.

Установить имя можно для ограничений PRIMARY KEY, CHECK, UNIQUE, а также FOREIGN KEY, который будет рассматриваться далее.

163. Изменение таблиц и столбцов

Ответ

Если таблица уже была ранее создана, и ее необходимо изменить, то для этого применяется команда **ALTER TABLE**. Ее сокращенный формальный синтаксис:

```

ALTER TABLE название_таблицы
{ ADD название_столбца тип_данных_столбца [атрибуты_столбца] |
DROP COLUMN название_столбца |
MODIFY COLUMN название_столбца тип_данных_столбца [атрибуты_столбца] |
ALTER COLUMN название_столбца SET DEFAULT значение_по_умолчанию |
ADD [CONSTRAINT] определение_ограничения |
DROP [CONSTRAINT] имя_ограничения }

```

Основные сценарии, с которыми можно столкнуться

Добавление нового столбца

Добавим в таблицу customers новый столбец address:

```

ALTER TABLE customers
ADD address VARCHAR(50) NULL;

```

В данном случае столбец address имеет тип VARCHAR и для него определен атрибут NULL.

Удаление столбца

Удалим столбец address из таблицы customers:

```

ALTER TABLE customers
DROP COLUMN address;

```

Изменение значения по умолчанию

Установим в таблице customers для столбца age значение по умолчанию 22:

```

ALTER TABLE customers
ALTER COLUMN age SET DEFAULT 22;

```

Изменение типа столбца

Изменим в таблице customers тип данных у столбца first_name на CHAR(100) и установим для него атрибут NULL:

```
ALTER TABLE customers  
MODIFY COLUMN first_name CHAR(100) NULL;
```

Добавление и удаление внешнего ключа

Пусть изначально в базе данных будут добавлены две таблицы, никак не связанные:

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    age INT,  
    first_name VARCHAR(20) NOT NULL,  
    last_name VARCHAR(20) NOT NULL  
) ENGINE=INNODB;  
  
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    customer_id INT,  
    created_at DATE  
) ENGINE=INNODB;
```

Добавим ограничение внешнего ключа к столбцу customer_id таблицы orders:

```
ALTER TABLE orders  
ADD FOREIGN KEY(customer_id) REFERENCES customers (id);
```

При добавлении ограничений мы можем указать для них имя, используя оператор **CONSTRAINT**, после которого указывается имя ограничения:

```
ALTER TABLE orders  
ADD CONSTRAINT orders_customers_fk  
FOREIGN KEY(customer_id) REFERENCES customers (id);
```

В данном случае ограничение внешнего ключа называется orders_customers_fk. Затем по этому имени мы можем удалить ограничение:

```
ALTER TABLE orders  
DROP FOREIGN KEY orders_customers_fk;
```

Добавление и удаление первичного ключа

Добавим в таблицу products первичный ключ:

```
CREATE TABLE products (  
    id INT,  
    model VARCHAR(20)  
) ENGINE=INNODB;
```

```
ALTER TABLE products  
ADD PRIMARY KEY (id);
```

Теперь удалим первичный ключ:

```
ALTER TABLE products  
DROP PRIMARY KEY;
```

Если в процессе данных операций возникнет следующая ошибка:



Error Code: 1075. Incorrect table definition; there can be only one auto column and it must be defined as a key



Она возможно из за того что в столбце стоит **AUTO_INCREMENT**

Technology

Spring

1. Рассказать про Spring;
2. Spring inversion of control. Что такое. Как используется и для чего.
3. Spring dependency injection. Что такое. Как используется и для чего.

Hibernate

1. Уровни кеша;
2. Способы запросов;
3. Как из `select * from a,b` достать только объекты `b`;
4. Entities состояния;
5. Сессия в хибернейте;
6. `sessionFactory.getCurrentSession()` vs `entityManagerFactory.createEntityManager()`
7. У какого объекта (если использовать Hibernate) вызываются методы для работы с БД;
8. Если в хибернейте сделать:

```
Object o = transaction.get(id, table);  
o.setName(name);  
o.commit();
```

Сохранится ли `name`?

JMS

1. Что такое JMS Ant \$ Maven
1. Различия Ant & Maven;
2. Команды в консоли для JVM, Maven, Ant;
3. Какой командой билдится проект? Web services