

## 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. OOP 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 своих суперклассов (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 {<br/>    private static final long serialVersionUID = -3733782742070723489L;<br/>    private int id;<br/>    transient private String name; // не должны быть в методе hashCode()<br/>    private static int staticField;<br/><br/>    public User(int id, String name) {<br/>        this.id = id;<br/>        this.name = name;<br/>    }<br/><br/>    public int getId() {<br/>        return id;<br/>    }<br/><br/>    public void setId(int id) {<br/>        this.id = id;<br/>    }<br/><br/>    public String getName() {<br/>        return name;<br/>    }<br/><br/>    public void setName(String name) {<br/>        this.name = name;<br/>    }<br/>}</pre> |   |
| Сериализация объекта  | Десериализация объекта  |
| <pre>public static void main(String[] args)<br/>    throws IOException {<br/>    User user = new User(1, "Alex");<br/><br/>    FileOutputStream fos = new<br/>        FileOutputStream("pathName.bin");<br/>    ObjectOutputStream oos = new<br/>        ObjectOutputStream(fos);<br/>    oos.writeObject(user);<br/>    oos.close();<br/>}</pre>   | <pre>public static void main(String[] args)<br/>    throws ClassNotFoundException,<br/>        IOException {<br/>    FileInputStream fis = new<br/>        FileInputStream("pathName.bin");<br/>    ObjectInputStream ois = new<br/>        ObjectInputStream(fis);<br/>    User user = (User) ois.readObject();<br/>    ois.close();<br/>}</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-монтекса, который был «залочен (заблокирован)» текущей нитью, в противном случае метод выкинет исключение **`IllegalMonitorStateException`**.

В результате вызова этого метода, блокировка с объекта-монтекса 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 исполняется. 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 интерфейс. ArrayList представляет собой реализацию массива с изменяемым размером, тогда как LinkedList представляет собой реализацию двусвязного списка List интерфейс.

- **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 имплементирует 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 передается передаваемый объект, а в качестве значения добавляется объект, занесенный 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.

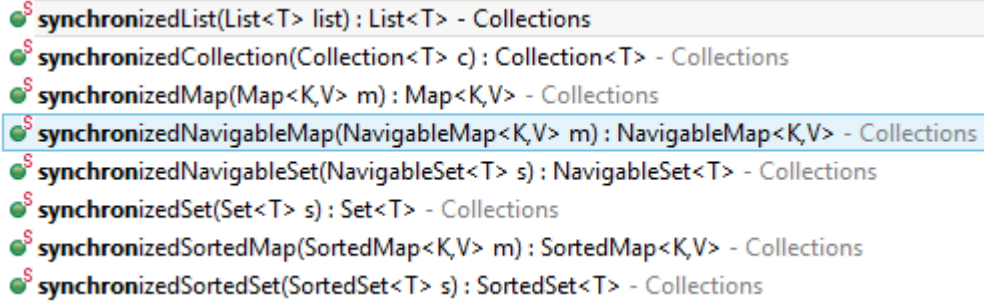
Ответ

**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` class used for synchronizing collections. Each method is preceded by a green icon with a red 'S'. The methods listed 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&lt;T&gt; {     boolean test(T t); }</pre> | <pre>public static void main(String[] args) {     Predicate&lt;Integer&gt; isEvenNumber = x -&gt; 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&lt;T&gt; {     void accept(T t); }</pre> | <pre>public static void main(String[] args) {     Consumer&lt;String&gt; greetings = x -&gt;     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&lt;T&gt; {     T get(); }</pre> | <pre>public static void main(String[] args) {     List&lt;String&gt; nameList = new ArrayList&lt;&gt;();     nameList.add("Elena");     nameList.add("John");     nameList.add("Alex");     nameList.add("Jim");     nameList.add("Sara");      Supplier&lt;String&gt; randomName = () -&gt; {         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&lt;T, R&gt; {     R apply(T t); }</pre> | <pre>public static void main(String[] args) {     Function&lt;String, Integer&gt; valConvert = x -&gt; 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&lt;String&gt; list = new ArrayList&lt;&gt;();     Collections.addAll(list, "Привет", "как",         "дела?");     list.forEach((s) -&gt; System.out.println(s)); }</pre> | <pre>public static void main(String[] args) {     List&lt;String&gt; list = new ArrayList&lt;&gt;();     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&lt;Character&gt;         .map(String::valueOf) //     Stream&lt;String&gt;         .map(Integer::parseInt); //     Stream&lt;Integer&gt;     }</pre> | <pre>public static void main(String[] args) {     List&lt;Integer&gt; a = Arrays.asList(1, 2, 3);     List&lt;Integer&gt; b = Arrays.asList(4, 5);     List&lt;Integer&gt; c = Arrays.asList(6, 7, 8);      List&lt;List&lt;Integer&gt;&gt; listOfListOfInts =         Arrays.asList(a, b, c); // [[1, 2,     3], [4, 5], [6, 7, 8]]      List&lt;Integer&gt; listOfInts =         listOfListOfInts.stream()             .flatMap(list -&gt; 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.

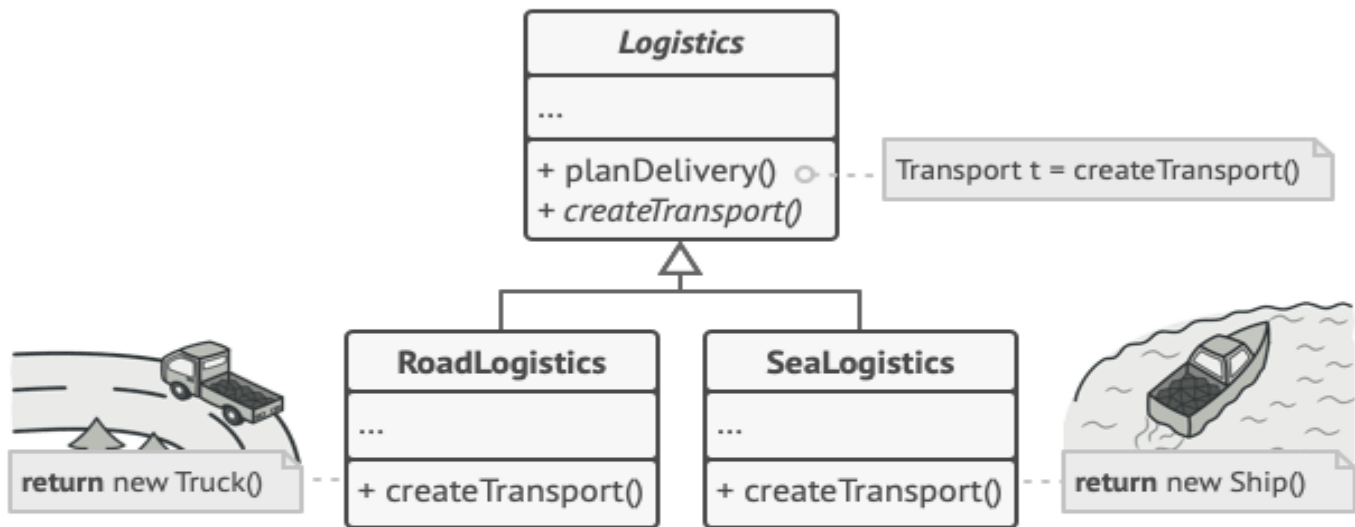
Ответ

### 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).

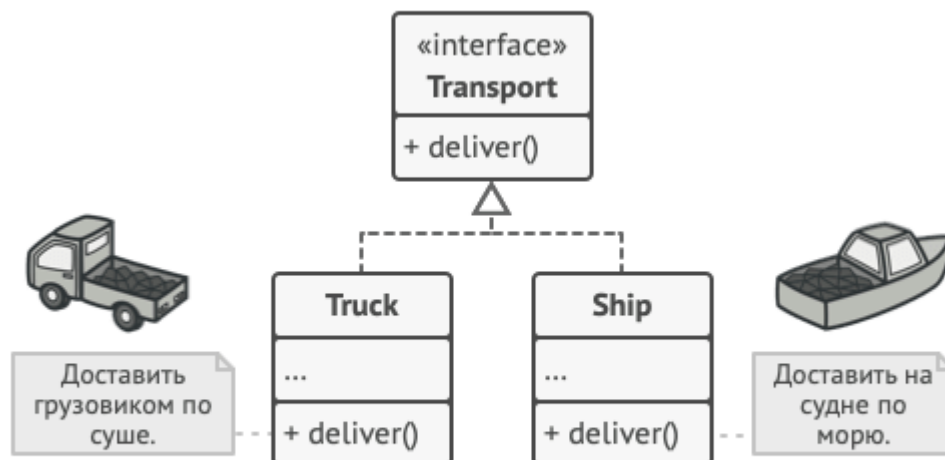
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор `new`, а через вызов особого *фабричного* метода. Не пугайтесь, объекты всё равно будут создаваться при помощи `new`, но делать это будет фабричный метод.



*Подклассы могут изменять класс создаваемых объектов.*

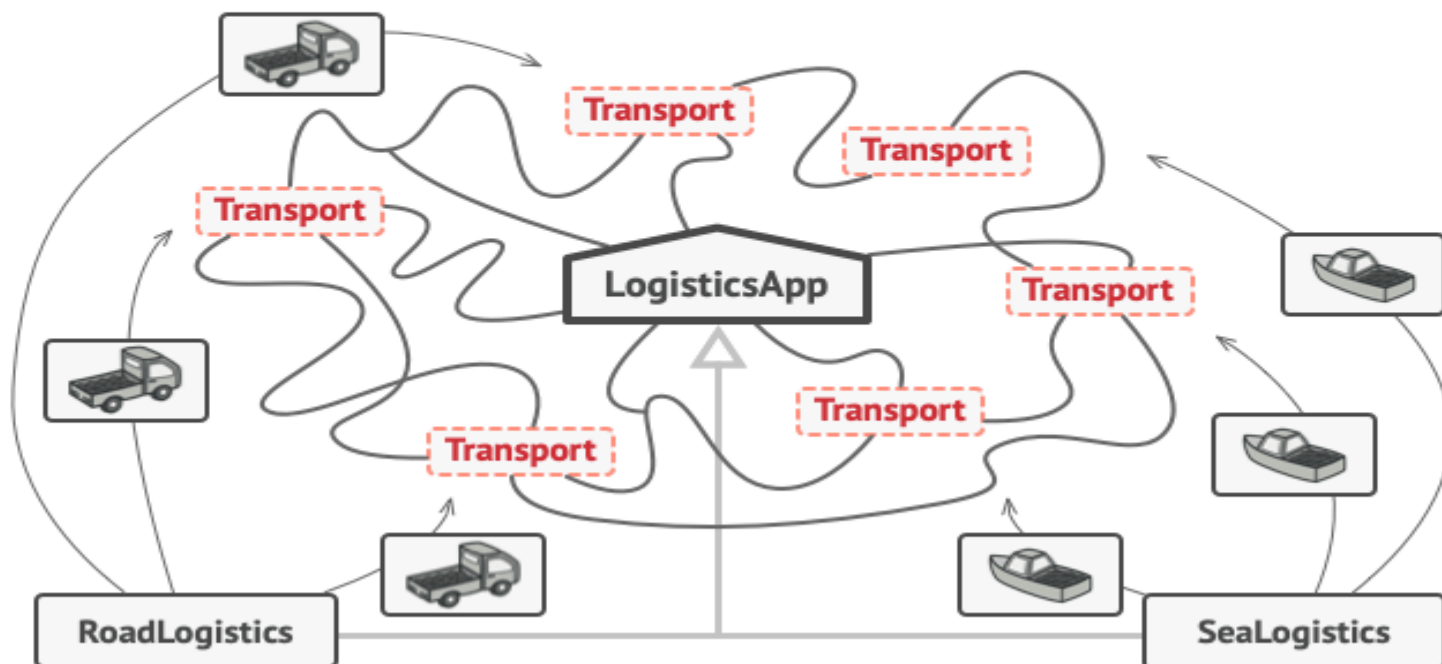
На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

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.

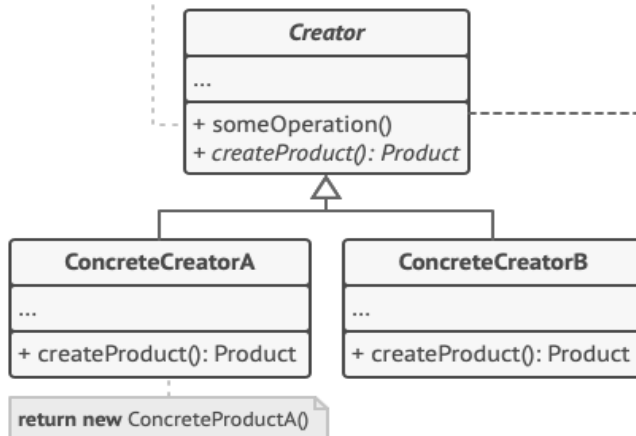
## Структура

**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.

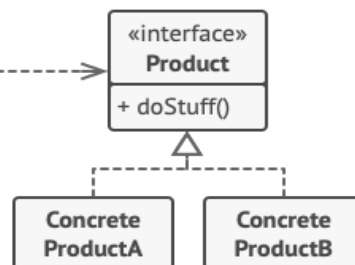
```
Product p = createProduct()
p.doStuff()
```



**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.

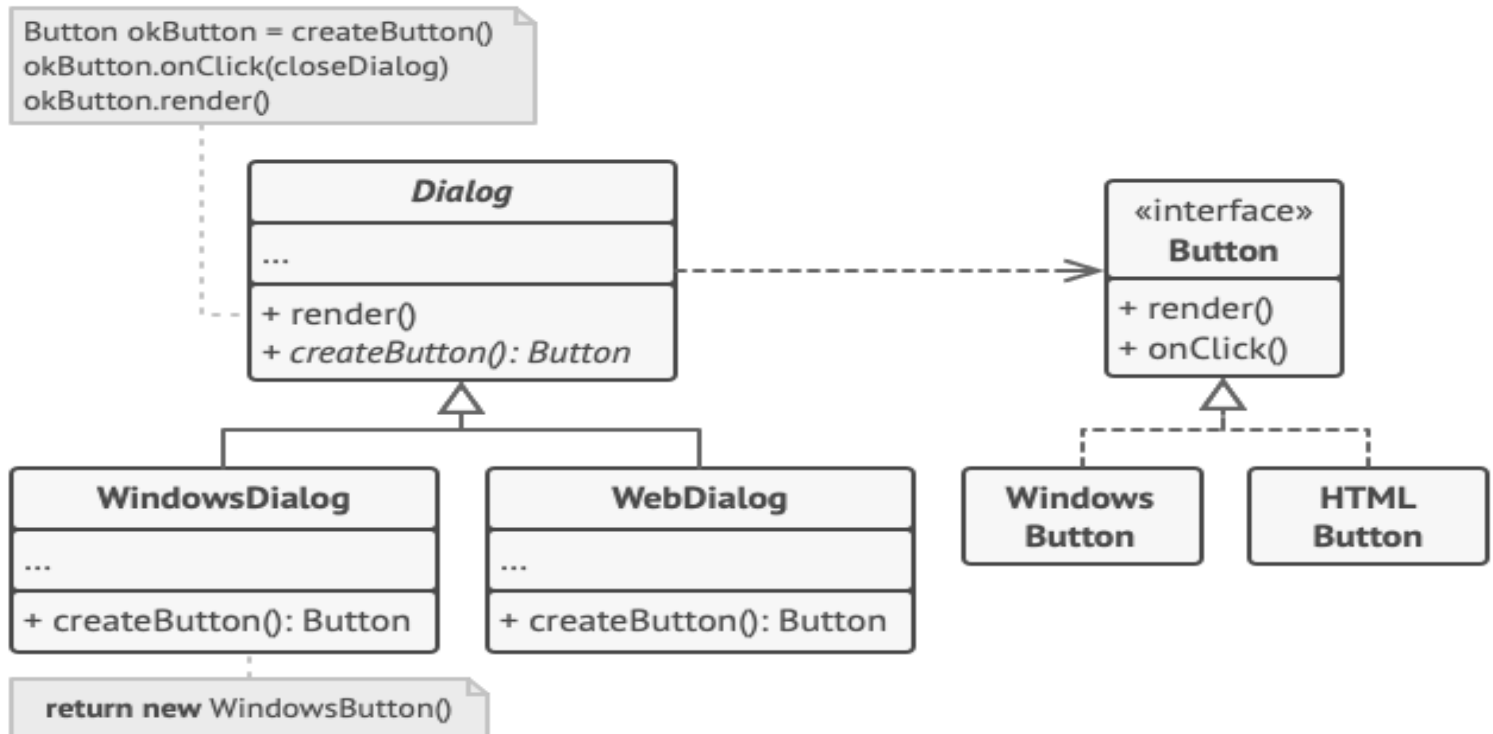
**1 Продукт** определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.



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

## 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

### 1 Интерфейс строителя

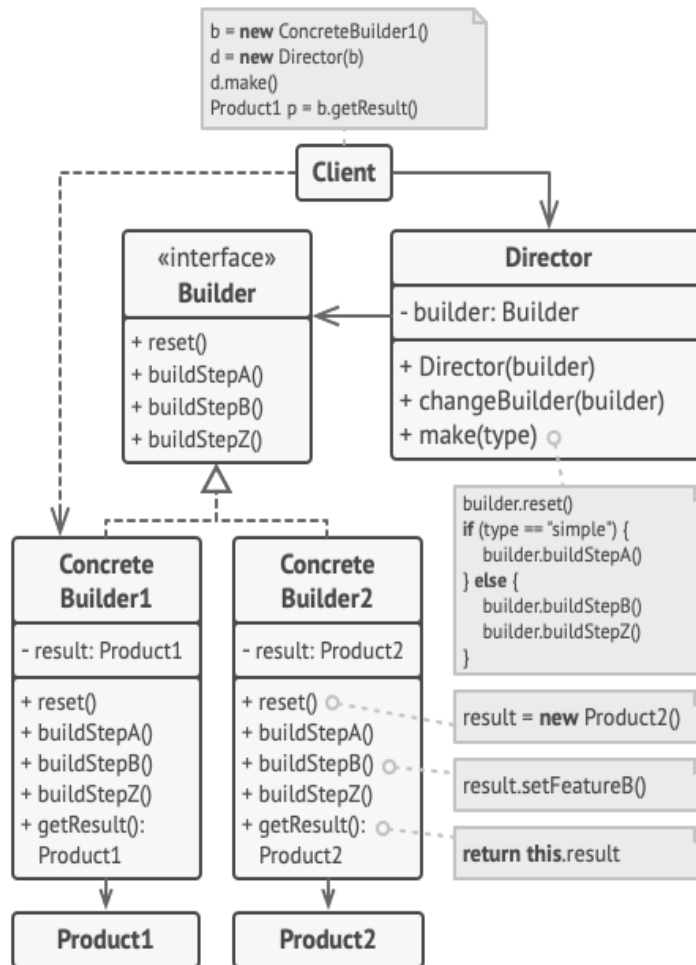
объявляет шаги конструирования продуктов, общие для всех видов строителей.

### 2 Конкретные строители

реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.

### 3 Продукт —

создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.



4 **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.

5 Обычно **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

## 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:

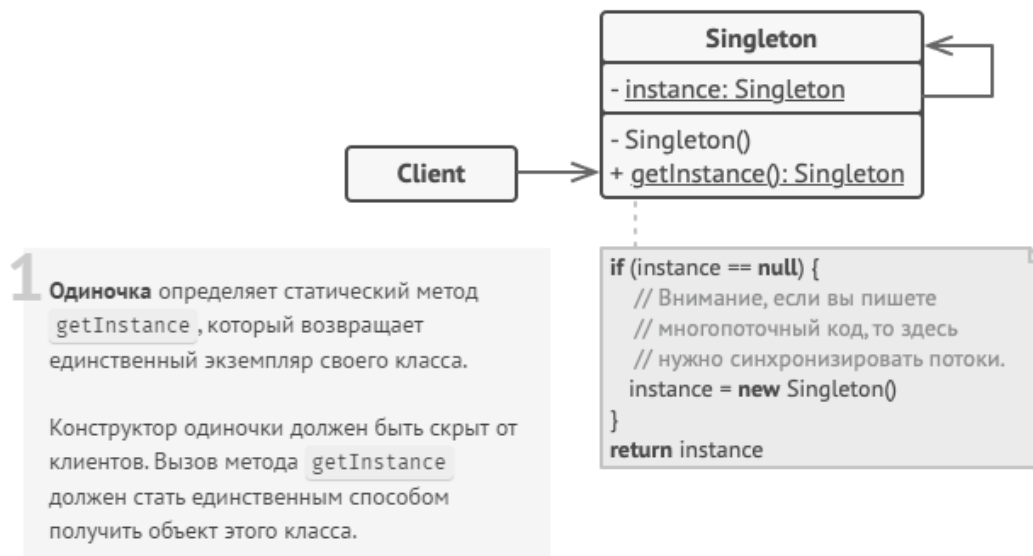
а) **Гарантирует наличие единственного экземпляра класса.** Чаше всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

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`'а 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

**130.** State (Thread.State)

Ответ

**131.** Iterator (Enumeration, Iterator, ListIterator)

Ответ

**132.** Proxy

Ответ

**133.** Observer (Listener-s)

Ответ



134. Wrapper

Ответ

135. Immutable

Ответ

136. MVC

Ответ

137. DAO vs. Repository

Ответ

### **JEE**

138. Что входит в JEE?

Ответ

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

Ответ

140. Что такое контейнер сервлетов? Что такое сервлет?

Ответ

141. Методы сервлета. Жизненный цикл сервлета.

Ответ

142. Что такое jsp. Жизненный цикл.

Ответ

143. Что такое сессия? Жизненный цикл.

Ответ

144. Что такое request? Из чего состоит? Жизненный цикл.

Ответ

145. Cookies. Как можно достать Cookies, а если Cookies удалить, можно ли достать сессию?

Ответ

146. Что нужно написать в браузерной строке, чтобы обратиться к сервлету? Можно ли из браузерной строки напрямую вызвать метод сервлета?

Ответ

**147.** Чем отличаются методы POST и GET. Если не указать напрямую, какой из этих методов выполнится по умолчанию?

Ответ

**148.** Как сделать redirect незаметно для пользователя?

Ответ

**149.** Какие scopes (области видимости) переменных существуют в JSP?

Ответ

**150.** В чём различие forward и redirect?

Ответ

**151.** Отличия `getAttribute()` от `getParameter()` в сервлете.

Ответ

**152.** Из чего состоит url?

Ответ

**153.** Отличие `jsp:include` от директивы `include`.

Ответ

**154.** Применение классов `HttpServletRequestWrapper` и `HttpServletResponseWrapper`.

Ответ

**155.** Что делает `RequestDispatcher.include()`?

Ответ

**156.** В какой последовательности выполняются сервлет фильтры?

Ответ

**157.** Как обрабатывается тег с телом?

Ответ

**158.** JSTL.

Ответ

**159.** 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

**160.** Можно ли в web.xml определить сервлет без указания url паттерна и как к нему обратиться?

Ответ

**161.** Что такое HTTP? Отличия HTTP 1.0 и HTTP 2.

Ответ

**162.** Сохраняет ли http протокол своё состояние.

Ответ

**163.** Как сервер понимает, что для пользователя создана сессия и не нужно её создавать?

Ответ

**164.** Отличие авторизации от аутентификации.

Ответ

**165.** Сервер приложений. Веб-сервер. Отличия.

Ответ

**166.** Пошагово рассказать, что происходит, когда пользователь нажимает на кнопку. (с формы поля сетаются в request, потом вызывается контейнер сервлетов, потом он както по request понимает, куда нужно идти дальше (в дескриптор развертывания, а их может быть несколько, нужно как-то понимать в какой).

Ответ

**167.** Какие бывают Webservice?

Ответ

**168.** RESTful сервис. Принципы работы.

Ответ

## **SQL**

**169.** Что такое нормализация.

Ответ

**170.** Какие есть типы связей в базе данных. Привести пример.

Ответ

171. Что такое primary key (первичный ключ)?

Ответ

172. Что такое foreign key (внешний ключ)?

Ответ

173. Что такое индексы в базе данных? Для чего их используют? Чем они хороши и чем плохи?

Ответ

174. Какие есть типы JOIN'ов. Кратко опишите каждый из типов.

Ответ

175. Для чего используется слово HAVING? Отличия от WHERE.

Ответ

176. Зачем нужно View и какие поля там будут.

Ответ

177. Есть таблицы Customer и Order. Вывести всех customer, у которых суммарный заказ будет > 10000.

Ответ

178. Что такое агрегирующая функция, примеры.

Ответ

179. Результат запроса Select \* from Table1, Table2;

Ответ

180. 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;

## 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