1. Какие есть типы данных в Java?
2. Чем отличаются объектные типы от примитивных типов данных?
3. В чем разница передачи параметров по ссылке и по значению?
4. Что такое OutOfMemoryError?
5. **Что такое стектрейс? Как его получить?**
6. Назовите все методы класса Object.
7. Что такое конструкторы? Какие типы знаете?
8. ***Объекты каких стандартных классов immutable в Java?***
9. ***Дайте краткую характеристику immutable object. Зачем они нужны? Как их создать? Какие преимущества immutable object перед обычными объектами?***
10. Что такое ООП? Назовите принципы с примерами.
11. Расскажите о принципе наследования в ООП. Зачем он нужен?
12. Дайте определение принципа полиморфизма в ООП. Как работает полиморфизм?
13. Дайте определение принципа абстракции в ООП.
14. Какие элементы языка отвечают за инкапсуляцию?
15. ***Какие элементы языка отвечают за наследование? Какие особенности использования ключевого слова super?***
16. Какие элементы языка отвечают за полиморфизм?
17. Что такое перегрузки (overloading) метода?
18. Что такое переопределение (override) метода?
19. Что такое класс, объект, интерфейс?
20. Какие элементы могут содержать класс?
21. ***Что такое сигнатура метода?***
22. ***Можно ли в конструкторе применять return?***
23. Можно ли в конструкторе выкинуть исключение (exception)?
24. Когда применяется ключевое слово this?
25. Что такое модификаторы доступа в Java? Назовите их. Для чего используются?
26. Назовите основную особенность статических переменных и методов. Какие основные ограничения действуют на статические переменные и методы?
27. ***Что означает ключевое слово static? Может ли статический метод быть переопределенным или перегруженным?***
28. Можно ли использовать статические методы внутри обычных? Наоборот? Почему?
29. Что означает ключевое слово final?
30. ***Что такое abstract? Абстрактный класс? Абстрактный метод?***
31. ***Что такое interface? Может ли быть final interface?***
32. ***В чем разница между абстрактным классом и интерфейсом в Java?***
33. Какие модификаторы доступа могут быть у класса?
34. ***Что такое класс Assert? Зачем и как его использовать?***
35. Расскажите про String в Java.
36. Какие есть способы создания объекта String? Где он создается?
37. Как сравнить две строки в Java и/или отсортировать их?
38. ***Что такое строковый пул и зачем он нужен?***
39. Какая разница между String, StringBuffer и StringBuilder?
40. Дайте краткую характеристику Enum в Java.
41. ***Может ли Enum реализовывать (implement) интерфейс?***
42. ***Может ли Enum расширить (extends) класс?***
43. ***Можно ли создать Enum без экземпляров объектов?***
44. ***Можем ли мы переопределить метод toString() для Enum?***
45. ***Что будет, если не переопределять метод toString() для Enum?***
46. Можем ли мы указать конструктор внутри Enum?
47. В чем разница между == и equals()?
48. Что делает метод ordinal() в Enum?
49. ***Можно ли использовать Enum с TreeSet или TreeMap в Java?***
50. ***Как связаны методы ordinal() и compareTo() в Enum?***
51. Можно ли использовать Enum в switch case?
52. Как получить все имеющиеся значения в экземпляре Enum?
53. ***Что такое функциональный интерфейс?***
54. ***Какие функциональные интерфейсы входят в стандартную библиотеку Java?***
55. Что такое лямбда-выражение?
56. Что такое ссылка на метод (::)? Чем это отличается от лямбда-выражения?
57. Что такое Stream API в Java?
58. Как создать Stream из коллекции в Java?
59. ***Что такое промежуточные операции в Stream API? Какие примеры промежуточных операций в Stream API вы знаете?***
60. ***Что такое терминальные операции в Stream API? Какие примеры терминальных операций в Stream API вы знаете?***
61. Что такое параллельные стримы?
62. ***Какие методы можно использовать для группировки элементов Stream?***
63. Какой параметр принимает в себя метод filter?
64. ***Какая иерархия коллекций в Java Collection Framework?***
65. ***Какое внутреннее строение ArrayList?***
66. ***Какое внутреннее строение LinkedList?***
67. ***Какое внутреннее строение HashMap?***
68. Чем отличаются ArrayList и LinkedList?
69. Чем отличаются ArrayList и HashSet?
70. ***Зачем в Java такое многообразие имплементации динамического массива?***
71. ***Зачем в Java такое многообразие имплементации key-value storage?***
72. Как отсортировать коллекцию элементов?
73. В чем преимущество массива перед ArrayList?
74. Дайте краткую характеристику class object в Java.
75. Для чего используют Equals and HashCode в Java?
76. Расскажите о контракте между Equals and HashCode в Java.
77. ***Какие условия выдвигаются касательно переопределения соглашения при переопределении Equals?***
78. Что будет, если не переопределить Equals and HashCode?
79. Какие значения мы получим, если у нас не перераспределены Equals and HashCode?
80. ***Почему симметричность выполняется, только если x.equals(y) возвращает true?***
81. Что такое коллизия в HashCode? Как с ней бороться?
82. ***Что будет, если элемент, участвующий в контракте с HashCode, меняет свое значение?***
83. ***В чем разница применения if (obj instanceof Student) и if (getClass() == obj.getClass())?***
84. Дайте краткую характеристику метода clone().
85. Дайте определение понятию exception (исключительная ситуация).
86. Какие знаете особенности использования оператора try ... catch?
87. ***В чем разница между error и exception?***
88. ***Какая разница между checked и unchecked, exception, throw, throws?***
89. ***Какая есть иерархия исключений?***
90. Что такое checked и unchecked exception?
91. Нужно ли проверять checked exception?
92. О чем говорит ключевое слово throws и как его использовать?
93. ***Какие возможные способы обработки исключений вы знаете?***
94. Какие есть правила для проверки исключений при наследовании?
95. Какой оператор позволяет принудительно выбросить исключение? Напишите пример.
96. ***Может ли метод main выбросить throws-исключение? Если да, то куда передаст?***
97. ***Как добиться StackOverflowError?***
98. В каких случаях возникает OutOfMemoryError?
99. ***Зачем нужен Garbage Collector?***
100. Что такое Heap? ***Что такое Stack?***
101. Что такое пул строк (string pool) и как он работает в Java?
102. ***Что такое процесс и поток?***
103. Что такое многопоточность и зачем она используется в Java?
104. Как создать и запустить новый поток в Java?
105. ***Какие способы синхронизации доступа к общим ресурсам в Java вы знаете?***
106. Какие проблемы могут возникнуть при работе с многопоточностью в Java?
107. ***Что такое состояние гонки (race condition) и как ее можно предотвратить в Java?***
108. ***Что такое дэдлок (deadlock) и как его можно избежать в Java?***
109. ***Какие классы и интерфейсы в Java используются для работы с многопоточностью?***
110. ***Какие методы и классы в Java используются для синхронизации потоков?***
111. ***Что такое пул потоков (thread pool) и как он используется в Java?***
112. ***Что такое ForkJoinPool?***
113. ***Что такое агрегатные операторы?***
114. Что такое нормальные формы? Какие нормальные формы вы знаете?
115. Что такое индексы? Какие индексы вы знаете? ***Какая структура данных лежит в их основе?***
116. ***В чем отличие WHERE от HAVING?***
117. Что такое JOIN? Какие типы JOINs вы знаете?
118. Какие основные типы данных в SQL вы знаете?
119. Что такое первичный ключ?
120. ***Что такое Spring Framework и какие преимущества он предоставляет?***
121. ***Какие модули входят в Spring Framework?***
122. ***Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)? Как они реализуются в Java?***
123. ***Что такое спринг бин? Как создать спринг бин?***
124. ***Опишите жизненный цикл бина.***
125. ***Что такое скоупы бина? Какие скоупы вы знаете?***
126. ***Зачем нужны аннотации @Qualifier и @Primary?***
127. ***Какие проблемы могут возникнуть при использовании Spring и как их можно решить?***
128. ***Что такое Spring Boot и какие преимущества он предоставляет?***
129. ***Какая разница между Spring и Spring Boot?***
130. ***Какая разница между аннотациями @Service, @Component и @Repository?***
131. ***Какие аннотации используются в Spring MVC и для чего они предназначены?***
132. Зачем нужна аннотация @Value?
133. ***Что такое Spring Boot Starter и как он используется?***
134. ***Какие средства предоставляет Spring Boot для управления конфигурацией приложения?***
135. Какие инструменты предоставляет Spring Boot для разработки и тестирования приложений?
136. ***Что такое репозиторий в Spring Data и как он используется?***
137. ***Как подключиться к базе данных с помощью Spring Data?***
138. Какие требования предъявляются энтити-классу?
139. Зачем нужна аннотация @Query?
140. ***Что такое фетч-стратегия? Какие они бывают?***
141. ***Что такое проблема N+1?***

Ответы:

1. \*\*Типы данных в Java:\*\*

В Java существует два основных типа данных: примитивные типы данных и ссылочные типы данных (объекты).

Примитивные типы данных:

- byte

- short

- int

- long

- float

- double

- char

- boolean

Ссылочные типы данных (объекты):

- классы

- интерфейсы

- массивы

2. \*\*Отличия между объектными и примитивными типами данных:\*\*

- Примитивные типы данных хранят само значение, тогда как объектные типы данных (объекты) хранят ссылку на объект в памяти.

- Примитивные типы данных имеют фиксированный размер, в то время как объектные типы данных могут занимать переменное количество памяти в зависимости от размера объекта.

- Для примитивных типов данных можно использовать операторы для выполнения простых операций, в то время как для объектных типов данных необходимо использовать методы.

3. \*\*Разница между передачей параметров по ссылке и по значению:\*\*

- При передаче параметра по значению передается само значение переменной.

- При передаче параметра по ссылке передается ссылка на объект в памяти, а не сам объект. Это означает, что если вы изменяете состояние объекта внутри метода, эти изменения будут видны в вызывающем коде.

4. \*\*OutOfMemoryError:\*\*

OutOfMemoryError - это исключение, которое возникает в Java, когда приложению не хватает памяти для выполнения операции. Это может произойти, например, когда приложение пытается выделить больше памяти, чем разрешено для своего процесса, или когда утечка памяти приводит к исчерпанию доступной памяти. Обычно это происходит из-за неправильного использования памяти в коде, например, создания большого количества объектов без их освобождения (сборка мусора).

5. \*\*Стек-трейс (stack trace):\*\*

Стек-трейс представляет собой список вызовов методов (фреймов стека), который был выполнен до возникновения исключения или в определенный момент времени. Он содержит информацию о том, как программа достигла текущего состояния, включая имена методов, классы, номера строк и т. д.

Получить стек-трейс можно с помощью метода `printStackTrace()` у объекта исключения, например:

```java

try {

// код, который может вызвать исключение

} catch (Exception e) {

e.printStackTrace();

}

```

6. \*\*Методы класса Object:\*\*

Все классы в Java наследуются от класса `Object`, поэтому они автоматически наследуют его методы. Некоторые из этих методов включают:

- `equals(Object obj)`: сравнивает текущий объект с объектом, переданным в качестве аргумента.

- `hashCode()`: возвращает хэш-код объекта.

- `toString()`: возвращает строковое представление объекта.

- `getClass()`: возвращает класс объекта.

- `clone()`: создает и возвращает копию объекта.

- `finalize()`: вызывается сборщиком мусора перед удалением объекта.

public final native void notify()

public final native void notifyAll()

public final native void wait(long timeout)

public final void wait(long timeout, intnanos)

public final void wait()

Методы для контроля доступа к объекту из различных нитей. Управление синхронизацией нитей.

7. \*\*Конструкторы:\*\*

Конструкторы - это специальные методы в Java, которые вызываются при создании объекта класса. Они используются для инициализации объекта, установки начальных значений его полей и выполнения других операций, необходимых для корректного его использования. Типы конструкторов:

- По умолчанию

- Без аргументов

- С параметрами.

8. \*\*Immutable объекты в Java:\*\*

Immutable (неизменяемые) объекты - это объекты, состояние которых не может быть изменено после их создания. Некоторые стандартные классы, представляющие неизменяемые объекты в Java, включают:

Из того, что тебе железобетонно нужно запомнить уже сейчас — все классы-обертки над примитивными типами — неизменяемые.

Integer, Byte, Character, Short, Boolean, Long, Double, Float — все эти классы создают Immutable объекты. Сюда же относятся и классы, используемые для создания больших чисел — BigInteger и BigDecimal.

StackTrace. Он тоже отсносится к неизменяемым объектам.

9. \*\*Immutable объекты в Java:\*\*

- \*\*Характеристика:\*\* Immutable объекты в Java - это объекты, состояние которых не может быть изменено после их создания.

- \*\*Зачем нужны:\*\* Они нужны для обеспечения безопасности, предотвращения ошибок и облегчения параллельного программирования. Immutable объекты также упрощают разработку и улучшают производительность.

- \*\*Как создать:\*\* Для создания immutable объекта в Java следует следовать нескольким правилам, таким как:

- Сделать класс `final`, чтобы нельзя было наследовать его.

- Сделать все поля `private` и `final`, чтобы они не могли быть изменены после инициализации.

- Не предоставлять методы для изменения состояния объекта.

- Обеспечить безопасность при работе с изменяемыми объектами внутри immutable объекта (если они используются).

- \*\*Преимущества перед обычными объектами:\*\*

- Неизменяемость гарантирует безопасность потоков и предотвращает гонки данных.

- Упрощает отладку и разработку кода, так как не нужно беспокоиться о изменениях состояния объекта.

- Immutable объекты могут быть использованы как ключи в коллекциях, так как их состояние не меняется.

10. \*\*ООП (объектно-ориентированное программирование):\*\*

ООП - это парадигма программирования, основанная на концепции объектов, которые могут содержать данные в виде полей и код в виде методов. Принципы ООП:

- \*\*Инкапсуляция\*\*: Сокрытие деталей реализации и предоставление доступа к объекту только через интерфейс.

Пример: использование геттеров и сеттеров для доступа к полям объекта.

- \*\*Наследование\*\*: Механизм, позволяющий создавать новые классы на основе существующих классов.

Пример: класс `Manager`, который наследует свойства и методы класса `Employee`.

- \*\*Полиморфизм\*\*: Возможность объектов разных типов обрабатываться общим способом.

Пример: метод `draw()` может рисовать различные типы фигур, такие как круг, прямоугольник или треугольник.

11. \*\*Наследование в ООП:\*\*

Наследование - это механизм ООП, позволяющий создавать новые классы на основе существующих классов. Новый класс, называемый подклассом или производным классом, наследует свойства и методы родительского класса, называемого суперклассом или базовым классом. Наследование позволяет повторно использовать код, расширять функциональность и создавать иерархии классов для организации кода.

12. \*\*Полиморфизм в ООП:\*\*

Полиморфизм - это принцип ООП, который позволяет объектам различных классов обрабатываться с использованием общего интерфейса. Это означает, что один и тот же метод может иметь различную реализацию в разных классах. Полиморфизм может быть достигнут через переопределение методов (runtime polymorphism) или через перегрузку методов (compile-time polymorphism). Наиболее распространенный вид полиморфизма - это переопределение методов в подклассах.

13. \*\*Принцип абстракции в ООП:\*\*

Абстракция в ООП означает представление объектов в системе программирования, которое скрывает все реализационные детали и позволяет программистам сосредоточиться только на необходимых характеристиках объекта. То есть, это процесс выделения общих характеристик объектов и игнорирования ненужных деталей. Принцип абстракции позволяет разработчикам создавать абстрактные классы и интерфейсы, которые определяют общие методы и свойства для объектов, не уточняя их конкретной реализации.

14. \*\*Элементы языка, отвечающие за инкапсуляцию:\*\*

Инкапсуляция в ООП обеспечивает управление доступом к данным и методам объекта. В Java элементы, отвечающие за инкапсуляцию, включают:

- Модификаторы доступа (`private`, `protected`, `public`).

- Геттеры и сеттеры (методы доступа к полям объекта).

- Ключевое слово `this` (для доступа к текущему объекту внутри его методов).

15. \*\*Элементы языка, отвечающие за наследование:\*\*

Наследование в Java обеспечивается с помощью ключевого слова `extends` и класса `Object`, который является корневым для всех классов в Java. Основные элементы языка, отвечающие за наследование:

- Ключевое слово `extends`: используется для указания родительского класса.

- Ключевое слово `super`: используется для вызова конструктора или метода суперкласса из подкласса.

- Переопределение методов: подкласс может переопределить методы суперкласса.

16. \*\*Элементы языка, отвечающие за полиморфизм:\*\*

Полиморфизм в Java обеспечивается через два основных механизма: переопределение методов (runtime polymorphism) и перегрузку методов (compile-time polymorphism). Элементы языка, отвечающие за полиморфизм:

- Переопределение методов: когда метод в подклассе имеет ту же сигнатуру, что и метод в суперклассе, происходит полиморфизм.

- Перегрузка методов: когда в классе есть несколько методов с одним и тем же именем, но с разными параметрами.

- Интерфейсы: интерфейсы позволяют реализовывать полиморфизм, поскольку они позволяют объектам различных классов обрабатываться через общий интерфейс.

17. \*\*Перегрузка метода (overloading):\*\*

Перегрузка метода - это возможность иметь несколько методов с одним и тем же именем в одном классе, но с разными списками параметров (типами или количеством параметров). При вызове перегруженного метода Java определяет, какой метод должен быть вызван на основе типов переданных аргументов. Перегрузка метода позволяет создавать более удобные интерфейсы для работы с разными типами данных или количеством параметров.

18. \*\*Переопределение метода (override):\*\*

Переопределение метода - это возможность в подклассе предоставить новую реализацию метода, который уже определен в его суперклассе. При этом сигнатура метода (имя, типы параметров, возвращаемый тип) должна оставаться той же. При вызове метода для объекта подкласса будет выполнена его переопределенная реализация, а не реализация из суперкласса.

19. \*\*Класс, объект, интерфейс:\*\*

- \*\*Класс:\*\* В Java класс - это шаблон, описывающий состояние и поведение объектов. Он определяет поля (переменные) и методы (функции), которые могут быть использованы объектами этого класса.

- \*\*Объект:\*\* Объект - это экземпляр класса. Он является конкретной реализацией класса и обладает своим собственным состоянием и поведением.

- \*\*Интерфейс:\*\* В Java интерфейс - это абстрактный тип данных, который определяет набор методов, но не предоставляет их реализацию. Классы могут реализовывать интерфейсы, обеспечивая реализацию всех методов, определенных в интерфейсе.

20. \*\*Элементы, содержащиеся в классе:\*\*

В Java класс может содержать следующие элементы:

- Поля (переменные): хранят состояние объекта.

- Методы (функции): определяют поведение объекта.

- Конструкторы: используются для инициализации объекта.

- Вложенные классы и интерфейсы: классы и интерфейсы, объявленные внутри другого класса.

Давайте рассмотрим эти вопросы:

21. \*\*Сигнатура метода:\*\*

Сигнатура метода - это его уникальная идентификация, которая включает в себя его имя и типы параметров. Сигнатура метода определяет уникальность метода в пределах класса или интерфейса. Это означает, что два метода с одним и тем же именем, но разной сигнатурой, считаются различными методами и могут быть перегружены в Java.

****Определение.**** Два компонента объявления метода включают *****сигнатуру метода***** - ****имя метода**** и ****параметры****.

Пример метода, описанного определения выше:

calculateAnswer(double, int, double, double)

Cигнатура метода в сочетании с типом возвращаемого значения называется ****контрактом метода****.

****Спецификация**** метода может рассматриваться как документация для метода.

****Итог:****

public void **moveTo(int x, int y)** throws IOException — *сигнатура*

**public void moveTo(int x, int y) throws IOException** — *контракт*

*Спецификация* -

/\*\*

\* Method - Description of the method

\* @param int x - description of parameter

\* @param int y - description of parameter

\* @return int - description of the return value

\*\*/

int moveTo(int x, int y){...}

22. \*\*Использование return в конструкторе:\*\*

Да, можно использовать оператор `return` в конструкторе. Он может быть использован для немедленного выхода из конструктора или для возврата значения из конструктора (например, в случае фабричных методов, когда объект создается внутри конструктора). Однако следует помнить, что если возвращается какое-либо значение, оно должно быть совместимо с типом возвращаемого значения конструктора.

23. \*\*Выброс исключения в конструкторе:\*\*

Да, можно выбрасывать исключения в конструкторе. Если в процессе создания объекта возникают некорректные условия или ошибки, то конструктор может выбросить исключение для сообщения об этой проблеме. Например, если переданные параметры конструктору недопустимы, или если объект не может быть корректно инициализирован, то это может привести к выбросу исключения.

24. \*\*Применение ключевого слова this:\*\*

Ключевое слово `this` в Java используется для ссылки на текущий объект. Оно может применяться в нескольких контекстах:

- Для разрешения конфликтов имен между полями и локальными переменными.

- Для вызова других конструкторов из текущего конструктора (с помощью `this()`).

- Для передачи текущего объекта в качестве аргумента в методы или конструкторы других классов.

- Для возврата текущего объекта из методов, обеспечивая цепочку вызовов методов (method chaining).

25. \*\*Модификаторы доступа в Java:\*\*

Модификаторы доступа в Java определяют уровень доступа к классам, полям, методам и конструкторам внутри и за пределами пакета. Они позволяют управлять видимостью и доступом к членам класса. Основные модификаторы доступа:

- `public`: доступен из любого места.

- `protected`: доступен внутри пакета и для наследников класса.

- `default` (не указывается явно): доступен внутри пакета.

- `private`: доступен только внутри класса.

26. \*\*Особенность статических переменных и методов:\*\*

Основная особенность статических переменных и методов - они принадлежат классу, а не конкретному экземпляру класса (объекту). Это означает, что они доступны для использования без создания экземпляра класса. Основные ограничения статических переменных и методов:

- Они разделяются между всеми объектами данного класса.

- Нельзя обращаться к нестатическим полям или методам напрямую из статического контекста.

27. \*\*Ключевое слово static:\*\*

Ключевое слово `static` используется для создания статических переменных и методов в Java. Оно указывает, что переменная или метод принадлежит классу, а не конкретному объекту. Статический метод не может быть переопределен или перегружен, так как он связан с классом, а не с его экземплярами.

28. \*\*Использование статических методов:\*\*

- Да, статические методы могут быть использованы внутри обычных методов.

- Однако обычные методы не могут быть использованы внутри статических методов напрямую, так как они принадлежат конкретному экземпляру класса, а статический метод может быть вызван без создания объекта.

29. \*\*Ключевое слово final:\*\*

В Java ключевое слово `final` используется для указания, что сущность не может быть изменена после инициализации. Оно может применяться к переменным, методам и классам.

- Для переменных: значение переменной не может быть изменено после инициализации (константа).

- Для методов: метод не может быть переопределен в подклассах.

- Для классов: класс не может быть наследован.

30. \*\*Абстракция в Java:\*\*

- \*\*abstract:\*\* Ключевое слово `abstract` используется для определения абстрактного класса или абстрактного метода.

- \*\*Абстрактный класс:\*\* Это класс, который содержит хотя бы один абстрактный метод. Абстрактные классы не могут быть инстанциированы, то есть созданы объекты этого класса.

- \*\*Абстрактный метод:\*\* Это метод, который объявлен без реализации. Он должен быть переопределен (реализован) в подклассе.

31. \*\*Интерфейс (interface):\*\*

Интерфейс в Java - это абстрактный тип данных, который содержит только константы и абстрактные методы (без реализации). Он описывает функциональность, которую должны реализовать классы. Интерфейсы позволяют реализовывать множественное наследование. Нельзя использовать `final` с интерфейсом.

32. \*\*Разница между абстрактным классом и интерфейсом:\*\*

- \*\*Абстрактный класс:\*\* Может содержать как абстрактные, так и конкретные методы. Может иметь поля. Может расширять только один класс. Может использовать модификаторы доступа.

- \*\*Интерфейс:\*\* Содержит только абстрактные методы и константы. Не может иметь полей. Может реализовываться множеством классов. Методы по умолчанию (default methods) в интерфейсе имеют реализацию по умолчанию. Не может использовать модификаторы доступа, кроме `public`.

Продолжим:

33. \*\*Модификаторы доступа у класса:\*\*

Класс может иметь следующие модификаторы доступа:

- `public`: класс доступен из любого места.

- `default` (не указывается явно): класс доступен только внутри пакета.

- `final`: класс не может быть наследован.

- `abstract`: класс является абстрактным и не может быть инстанциирован.

34. \*\*Класс Assert:\*\*

Класс `Assert` в Java предоставляет простой механизм проверки предположений в программе. Он позволяет программистам вставлять утверждения в код, которые должны быть истинными во время выполнения программы. Если утверждение оказывается ложным, то программа генерирует исключение `AssertionError`.

35. \*\*String в Java:\*\*

`String` в Java - это класс, представляющий строки символов. Он является неизменяемым (immutable), что означает, что после создания объекта `String` его значение не может быть изменено. `String` в Java также является классом стандартной библиотеки, что делает его доступным для использования без необходимости импортирования.

36. \*\*Способы создания объекта String:\*\*

- С помощью литералов: `String str = "Hello";`

- С использованием оператора `new`: `String str = new String("Hello");`

- С помощью методов класса `String`: `String str = String.valueOf(123);`

Объекты `String` создаются в куче (heap), но литералы строк могут храниться в специальном пуле строк (string pool) для оптимизации использования п

Продолжим:

37. \*\*Сравнение и сортировка строк в Java:\*\*

- Для сравнения строк используются методы `equals()` для сравнения содержимого и `compareTo()` для сравнения лексикографического порядка.

- Для сортировки строк можно использовать методы `sort()` из класса `Arrays` или `Collections.sort()` для коллекций.

```java

String str1 = "hello";

String str2 = "world";

// Сравнение строк

if (str1.equals(str2)) {

// строки равны

}

// Сортировка строк

String[] array = {"apple", "banana", "orange"};

Arrays.sort(array);

```

38. \*\*Строковый пул (string pool):\*\*

Строковый пул - это специальный механизм в Java, который используется для хранения литералов строк. Он представляет собой часть памяти, где хранятся уникальные строки. Когда создается строковый литерал, JVM сначала проверяет, есть ли он уже в пуле строк. Если есть, то возвращается ссылка на существующий объект, иначе создается новый объект и помещается в пул строк. Строковый пул помогает экономить память и улучшает производительность, так как предотвращает создание дубликатов строк.

39. \*\*String, StringBuffer и StringBuilder:\*\*

- `String`: неизменяемый (immutable) класс строк, подходит для работы с неизменяемыми строками.

- `StringBuffer`: потокобезопасный (thread-safe) класс, предназначенный для манипуляции изменяемыми строками в многопоточной среде.

- `StringBuilder`: не потокобезопасный (non-thread-safe) класс, предназначенный для манипуляции изменяемыми строками в однопоточной среде. Обеспечивает более высокую производительность за счет отсутствия синхронизации.

40. \*\*Enum в Java:\*\*

Enum (перечисление) в Java - это специальный тип данных, который используется для определения набора именованных констант. Enum в Java обеспечивает безопасность типов и удобство использования. Перечисления могут содержать конструкторы, методы и поля. Перечисления также могут реализовывать интерфейсы. Каждый элемент перечисления представляет собой отдельный объект. Преимущества использования Enum включают повышенную читаемость кода и защиту от опечаток при использовании констант.

Продолжим:

41. \*\*Enum и реализация интерфейса:\*\*

Да, Enum в Java может реализовывать интерфейс. Это позволяет использовать перечисления для определения констант с определенной функциональностью. Каждый элемент Enum может реализовывать методы, определенные в интерфейсе.

```java

public enum Day implements Printable {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

@Override

public void print() {

System.out.println("Today is " + this);

}

}

interface Printable {

void print();

}

```

42. \*\*Enum и расширение класса:\*\*

В Java Enum не может расширять классы, так как он уже наследует абстрактный класс `Enum`, который содержит основной функционал перечислений.

43. \*\*Создание Enum без экземпляров:\*\*

Да, можно создать Enum без экземпляров объектов. Это может быть полезно, если перечисление содержит только константы без какого-либо дополнительного поведения или данных.

```java

public enum Direction {

NORTH, SOUTH, EAST, WEST;

}

```

44. \*\*Переопределение метода toString() для Enum:\*\*

Да, метод `toString()` может быть переопределен для Enum. По умолчанию метод `toString()` для Enum возвращает имя элемента Enum. Однако его можно переопределить, чтобы возвращать необходимое строковое представление элемента Enum.

```java

public enum Direction {

NORTH("North"),

SOUTH("South"),

EAST("East"),

WEST("West");

private final String value;

Direction(String value) {

this.value = value;

}

@Override

public String toString() {

return value;

}

}

```

Продолжим изучение Java:

45. \*\*Поведение без переопределения метода toString() для Enum:\*\*

Если не переопределить метод `toString()` для Enum, то по умолчанию будет использоваться реализация из класса `Enum`, которая возвращает имя элемента Enum в виде строки. Таким образом, если не переопределить `toString()`, то при вызове этого метода будет возвращено имя элемента Enum.

46. \*\*Указание конструктора в Enum:\*\*

Конструкторы в Enum могут быть объявлены только приватными и вызываться только внутри самого Enum. Это ограничение обеспечивает правильное создание и инициализацию экземпляров Enum.

```java

public enum Day {

MONDAY("Monday"),

TUESDAY("Tuesday"),

WEDNESDAY("Wednesday"),

THURSDAY("Thursday"),

FRIDAY("Friday"),

SATURDAY("Saturday"),

SUNDAY("Sunday");

private final String name;

private Day(String name) {

this.name = name;

}

}

```

47. \*\*Разница между == и equals():\*\*

- `==` используется для сравнения ссылок на объекты. Он проверяет, указывают ли две переменные на один и тот же объект в памяти.

- `equals()` используется для сравнения содержимого объектов. По умолчанию метод `equals()` сравнивает адреса памяти, но его можно переопределить для сравнения содержимого объектов.

48. \*\*Метод ordinal() в Enum:\*\*

Метод `ordinal()` в Enum возвращает порядковый номер (индекс) элемента Enum. Нумерация элементов начинается с нуля. Этот метод полезен для получения порядкового номера элемента Enum или для использования его в циклах или массивах.

```java

Day day = Day.MONDAY;

int ordinal = day.ordinal(); // возвращает 0

```

Продолжим:

49. \*\*Использование Enum с TreeSet или TreeMap:\*\*

Да, Enum можно использовать с `TreeSet` и `TreeMap`. Эти структуры данных требуют, чтобы элементы были сравнимыми (реализовали интерфейс `Comparable`). Поскольку Enum в Java реализует интерфейс `Comparable`, его можно без проблем использовать с `TreeSet` и `TreeMap`.

```java

import java.util.TreeSet;

public class Main {

public static void main(String[] args) {

TreeSet<Day> treeSet = new TreeSet<>();

treeSet.add(Day.MONDAY);

treeSet.add(Day.TUESDAY);

// ...

}

}

```

50. \*\*Связь методов ordinal() и compareTo() в Enum:\*\*

Метод `ordinal()` возвращает порядковый номер элемента Enum, который используется в методе `compareTo()` для сравнения порядковых номеров двух элементов Enum. По умолчанию реализация `compareTo()` в Enum основана на порядковых номерах (ordinal), однако этот метод можно переопределить для собственной логики сравнения.

51. \*\*Использование Enum в switch case:\*\*

Да, Enum можно использовать в конструкции `switch case`. Это удобно и позволяет написать более читаемый код для обработки различных значений перечисления.

```java

Day day = Day.MONDAY;

switch (day) {

case MONDAY:

// делаем что-то

break;

case TUESDAY:

// делаем что-то

break;

// ...

}

```

52. \*\*Получение всех значений Enum:\*\*

Для получения всех значений Enum можно воспользоваться методом `values()`, который возвращает массив всех значений перечисления.

```java

Day[] days = Day.values();

for (Day day : days) {

System.out.println(day);

}

```

Продолжим:

53. \*\*Функциональный интерфейс:\*\*

Функциональный интерфейс в Java - это интерфейс, который содержит только один абстрактный метод. Он предназначен для использования в качестве функциональных (функций высшего порядка) или замыканий. Функциональные интерфейсы позволяют обеспечивать гибкую и удобную работу с анонимными функциями в Java.

54. \*\*Функциональные интерфейсы в стандартной библиотеке Java:\*\*

В стандартной библиотеке Java содержится множество функциональных интерфейсов, некоторые из которых включают:

- `Runnable`

- `Callable`

- `Comparator`

- `Consumer`

- `Supplier`

- `Function`

- `Predicate`

- и другие.

55. \*\*Лямбда-выражение:\*\*

Лямбда-выражение в Java - это сокращенный способ создания анонимных функций или замыканий. Оно позволяет передавать функции как параметры и использовать их в качестве аргументов методов. Лямбда-выражения обеспечивают более лаконичный и читаемый код.

```java

Function<Integer, Integer> square = x -> x \* x;

```

56. \*\*Ссылка на метод (::) и отличия от лямбда-выражения:\*\*

Ссылка на метод (::) в Java - это способ передачи ссылки на существующий метод вместо явного написания лямбда-выражения. Он позволяет сократить код и улучшить его читаемость. Отличия от лямбда-выражения:

- Ссылка на метод используется для вызова существующего метода, а лямбда-выражение - для определения анонимной функции.

- Ссылка на метод обеспечивает прямую связь с методом, тогда как лямбда-выражение создает анонимный экземпляр функционального интерфейса.

- Ссылка на метод может использоваться, когда в аргументах лямбда-выражения передается вызов метода без изменений.

Продолжим:

57. \*\*Stream API в Java:\*\*

Stream API в Java представляет собой набор функций для работы с коллекциями и потоками данных. Он позволяет выполнять различные операции над элементами коллекций, такие как фильтрация, сортировка, преобразование и агрегация, с использованием функционального программирования.

58. \*\*Создание Stream из коллекции в Java:\*\*

В Java Stream можно создать из коллекции с помощью метода `stream()` или `parallelStream()`, доступных у интерфейса `Collection` и его наследников.

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Stream<Integer> stream = numbers.stream();

```

59. \*\*Промежуточные операции в Stream API:\*\*

Промежуточные операции в Stream API выполняются после создания потока и перед терминальной операцией. Они позволяют модифицировать или фильтровать элементы потока. Некоторые примеры промежуточных операций:

- `filter(Predicate)`

- `map(Function)`

- `sorted()`

- `distinct()`

- `limit(long)`

- `skip(long)`

60. \*\*Терминальные операции в Stream API:\*\*

Терминальные операции в Stream API завершают последовательность операций и производят результат. Они могут быть использованы для сбора, упрощения или агрегации данных. Некоторые примеры терминальных операций:

- `forEach(Consumer)`

- `collect(Collectors)`

- `count()`

- `min(Comparator)`

- `max(Comparator)`

- `reduce(BinaryOperator)`

- `anyMatch(Predicate)`

- `allMatch(Predicate)`

- `noneMatch(Predicate)`

Продолжим:

61. \*\*Параллельные стримы:\*\*

Параллельные стримы в Java - это специальный тип стримов, который позволяет выполнять операции над данными параллельно, то есть использовать несколько потоков для обработки данных. Параллельные стримы упрощают параллельную обработку данных и могут значительно ускорить выполнение операций для больших объемов данных.

62. \*\*Методы для группировки элементов Stream:\*\*

Для группировки элементов Stream можно использовать метод `Collectors.groupingBy()`. Этот метод позволяет сгруппировать элементы по заданному критерию и сохранить результат в виде `Map`. Также существует вариант `groupingBy()` с дополнительным коллектором.

```java

List<String> words = Arrays.asList("apple", "banana", "pear", "orange", "peach");

Map<Character, List<String>> groupedByFirstLetter = words.stream()

.collect(Collectors.groupingBy(s -> s.charAt(0)));

```

63. \*\*Параметр метода filter:\*\*

Метод `filter()` принимает в качестве параметра функцию-предикат (Predicate). Предикат - это функция, которая принимает один аргумент и возвращает булево значение (`true` или `false`). В методе `filter()` передается предикат, который определяет условие фильтрации элементов потока.

```java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

```

64. \*\*Иерархия коллекций в Java Collection Framework:\*\*

Иерархия коллекций в Java Collection Framework начинается с интерфейса `Collection`, который является базовым для всех коллекций. От `Collection` наследуются интерфейсы `List`, `Set` и `Queue`. От `List` происходит интерфейс `Deque`. От `Set` происходят интерфейсы `SortedSet` и `NavigableSet`. От `Queue` происходят интерфейсы `Deque` и `BlockingQueue`. От `Deque` происходит интерфейс `BlockingDeque`. Классы-реализации коллекций (например, `ArrayList`, `HashSet` и т. д.) реализуют соответствующие интерфейсы.

Продолжим:

65. \*\*Внутреннее строение ArrayList:\*\*

`ArrayList` в Java основан на массиве (array). Это динамический массив, который автоматически расширяется по мере добавления элементов. Внутренний массив в `ArrayList` увеличивается или уменьшается при добавлении или удалении элементов.

66. \*\*Внутреннее строение LinkedList:\*\*

`LinkedList` в Java представляет собой двусвязный список. Каждый элемент списка (узел) содержит ссылку на предыдущий и следующий элементы. Это позволяет эффективно добавлять и удалять элементы в середине списка, но доступ к элементам происходит последовательно.

67. \*\*Внутреннее строение HashMap:\*\*

`HashMap` в Java основана на хэш-таблице. Она состоит из массива (bucket array), каждый элемент которого представляет собой связанный список или дерево (в Java 8 и выше). Ключи хэшируются, и используется хеш-код для быстрого поиска и доступа к значениям.

68. \*\*Отличия между ArrayList и LinkedList:\*\*

- \*\*По структуре данных:\*\* `ArrayList` основан на массиве, а `LinkedList` на двусвязном списке.

- \*\*По производительности:\*\* Вставка/удаление элементов в середине `LinkedList` происходит быстрее, но доступ к элементам `ArrayList` быстрее благодаря индексам.

- \*\*По занимаемой памяти:\*\* `ArrayList` требует меньше памяти на хранение элементов, так как не требует дополнительных ссылок между элементами.

- \*\*По итерации:\*\* Итерация по `ArrayList` быстрее, так как доступ к элементам осуществляется по индексу, а в `LinkedList` требует проход по всему списку.

Продолжим:

69. \*\*Отличия между ArrayList и HashSet:\*\*

- `ArrayList` является списком, основанным на массиве, и сохраняет порядок элементов вставки.

- `HashSet` является набором, основанным на хэш-таблице, и не гарантирует порядок элементов. Он также гарантирует уникальность элементов.

70. \*\*Многообразие имплементаций динамического массива в Java:\*\*

В Java представлено много различных реализаций динамического массива, таких как `ArrayList`, `Vector`, `CopyOnWriteArrayList` и другие. Это позволяет выбирать подходящую реализацию в зависимости от требований к производительности, многопоточности и других факторов.

71. \*\*Многообразие имплементаций key-value storage в Java:\*\*

В Java представлено множество реализаций key-value хранилищ, таких как `HashMap`, `TreeMap`, `LinkedHashMap`, `ConcurrentHashMap` и другие. Каждая из них имеет свои особенности и преимущества, позволяя выбирать подходящую реализацию в зависимости от требований по производительности, порядку элементов, многопоточности и т. д.

72. \*\*Способы сортировки коллекции элементов:\*\*

В Java коллекцию элементов можно отсортировать с использованием методов класса `Collections` или методов Stream API.

- С использованием методов класса `Collections`:

```java

List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);

Collections.sort(numbers); // Сортировка по возрастанию

Collections.sort(numbers, Collections.reverseOrder()); // Сортировка по убыванию

```

- С использованием методов Stream API:

```java

List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);

List<Integer> sortedNumbers = numbers.stream()

.sorted()

.collect(Collectors.toList()); // Сортировка по возрастанию

List<Integer> reverseSortedNumbers = numbers.stream()

.sorted(Comparator.reverseOrder())

.collect(Collectors.toList()); // Сортировка по убыванию

```

Продолжим:

73. \*\*Преимущество массива перед ArrayList:\*\*

Основное преимущество массива перед `ArrayList` заключается в более низком использовании памяти и меньшем времени доступа к элементам. Это связано с тем, что массивы в Java являются непрерывными участками памяти, а `ArrayList` - динамическим массивом, который требует дополнительной памяти для управления размером и хранения дополнительных данных.

74. \*\*Характеристика class object в Java:\*\*

`Object` является базовым классом для всех объектов в Java. Он определяет базовые методы, которые доступны для всех объектов, такие как `equals()`, `hashCode()`, `toString()` и т. д. Все классы в Java наследуются от `Object`, поэтому они автоматически наследуют эти методы.

75. \*\*Использование Equals и HashCode в Java:\*\*

Методы `equals()` и `hashCode()` используются для сравнения объектов и для работы с коллекциями в Java.

- `equals()` используется для сравнения содержимого объектов.

- `hashCode()` используется для создания хеш-кода объекта, который используется в хеш-таблицах для быстрого поиска и доступа к данным.

76. \*\*Контракт между Equals и HashCode в Java:\*\*

В Java существует контракт между методами `equals()` и `hashCode()`, который состоит в следующем:

- Если два объекта равны согласно методу `equals()`, то их хеш-коды должны быть равны.

- Если хеш-коды двух объектов равны, это не обязательно означает, что объекты равны согласно методу `equals()`, так как возможны коллизии хеш-кодов. Однако, если два объекта не равны согласно методу `equals()`, их хеш-коды не обязательно должны быть разными, но это может улучшить производительность хеш-таблиц.

Продолжим:

77. \*\*Условия переопределения соглашения при переопределении Equals:\*\*

При переопределении метода `equals()` в Java следует соблюдать следующие условия:

- Рефлексивность: Объект должен быть равен самому себе.

- Симметричность: Если `x.equals(y)` возвращает `true`, то `y.equals(x)` также должен возвращать `true`.

- Транзитивность: Если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, то `x.equals(z)` также должен возвращать `true`.

- Консистентность: Для неизменных объектов результат вызова метода `equals()` должен быть постоянным.

- Сравнение с `null`: `x.equals(null)` всегда должен возвращать `false`.

78. \*\*Последствия отсутствия переопределения Equals and HashCode:\*\*

Если не переопределить методы `equals()` и `hashCode()`, то объекты будут сравниваться по ссылке (по умолчанию это реализовано в классе `Object`). Это может привести к неправильному поведению при использовании объектов в коллекциях, где они должны сравниваться по значению.

79. \*\*Значения, если не перераспределены Equals and HashCode:\*\*

Если методы `equals()` и `hashCode()` не переопределены, то объекты будут сравниваться по ссылке, а хеш-код будет возвращать адрес объекта в памяти. Это может привести к тому, что объекты с одинаковым содержимым будут считаться разными.

80. \*\*Почему симметричность выполняется, только если x.equals(y) возвращает true:\*\*

Симметричность гарантирует, что если `x.equals(y)` возвращает `true`, то `y.equals(x)` также должен возвращать `true`. Это важно для правильной работы сравнения объектов и избежания непредсказуемого поведения. Если симметричность не выполняется, это может привести к ошибкам и некорректным результатам сравнения объектов.

Продолжим:

81. \*\*Коллизия в HashCode и способы борьбы с ней:\*\*

Коллизия в хеш-функции происходит, когда разные объекты получают одинаковый хеш-код. Это может привести к тому, что объекты будут храниться в одной корзине (bucket) хеш-таблицы. Для борьбы с коллизиями можно использовать следующие методы:

- \*\*Хорошо спроектированные хеш-функции:\*\* Хорошо распределенные хеш-функции могут уменьшить вероятность коллизий.

- \*\*Использование структур данных с разрешением коллизий:\*\* Например, метод цепочек (chaining) или метод открытой адресации (open addressing).

- \*\*Увеличение размера хеш-таблицы:\*\* Увеличение размера таблицы может уменьшить вероятность коллизий.

82. \*\*Изменение значения элемента, участвующего в контракте с HashCode:\*\*

Если элемент, участвующий в контракте с `hashCode()`, меняет свое значение после того, как он был помещен в хеш-структуру (например, в `HashMap`), это может привести к тому, что этот элемент станет недоступным для поиска или будет находиться в неправильной корзине, что может привести к непредсказуемому поведению.

83. \*\*Разница между использованием `if (obj instanceof Student)` и `if (getClass() == obj.getClass())`:\*\*

- `if (obj instanceof Student)`: Проверяет, является ли объект `obj` экземпляром класса `Student` или его подклассом.

- `if (getClass() == obj.getClass())`: Сравнивает класс объекта `obj` с классом текущего объекта посредством вызова метода `getClass()`. Этот подход сравнивает конкретные классы объектов, игнорируя иерархию наследования.

84. \*\*Метод `clone()`:\*\*

Метод `clone()` в Java используется для создания копии объекта. Класс, который поддерживает клонирование, должен реализовать интерфейс `Cloneable`. Метод `clone()` создает и возвращает непрозрачную копию объекта, что означает, что изменения в копии не влияют на оригинал и наоборот. Однако использование `clone()` может быть проблематичным из-за его поведения по умолчанию и потенциальных проблем с производительностью.

Продолжим:

85. \*\*Определение исключения (exception):\*\*

Исключение (exception) в Java - это объект, который сигнализирует о возникновении некоторой аномальной ситуации во время выполнения программы. Это может быть вызвано различными причинами, такими как ошибка ввода-вывода, деление на ноль, неправильное использование API и другие.

86. \*\*Особенности использования оператора try ... catch:\*\*

Оператор `try ... catch` в Java используется для обработки исключений. Код, который может вызвать исключение, помещается в блок `try`, а обработка исключения - в блок `catch`. Основные особенности использования `try ... catch`:

- Можно использовать несколько блоков `catch` для обработки различных исключений.

- Блок `finally` может использоваться для выполнения кода независимо от того, было ли исключение или нет.

87. \*\*Разница между error и exception:\*\*

- \*\*Исключение (exception):\*\* Это аномальная ситуация, которая может возникнуть во время выполнения программы и может быть обработана. Исключения обычно вызваны ошибками в программе или внешними условиями, такими как ошибка ввода-вывода или деление на ноль.

- \*\*Ошибка (error):\*\* Это серьезное состояние, которое обычно указывает на проблемы среды выполнения или внутренние ошибки JVM. Ошибки обычно не могут быть обработаны и должны быть решены на уровне системы, например, увеличением выделенной памяти для приложения.

88. \*\*Разница между checked и unchecked exception, throw, throws:\*\*

- \*\*Checked exception (проверяемые исключения):\*\* Это исключения, которые должны быть обработаны в коде программы с использованием блока `try ... catch` или объявлены в сигнатуре метода с помощью ключевого слова `throws`.

- \*\*Unchecked exception (непроверяемые исключения):\*\* Это исключения, которые не требуется объявлять в сигнатуре метода и могут возникать во время выполнения программы. Непроверяемые исключения обычно наследуются от класса `RuntimeException`.

- \*\*Throw:\*\* Оператор `throw` используется для генерации исключения в программе.

- \*\*Throws:\*\* Ключевое слово `throws` используется для объявления метода, который может возбуждать проверяемое исключение и передает ответственность за его обработку вызывающему коду.

Продолжим:

89. \*\*Иерархия исключений:\*\*

Иерархия исключений в Java начинается с класса `Throwable`, который является суперклассом для всех исключений в Java. Из `Throwable` наследуются два основных типа исключений: `Error` и `Exception`. `Error` используется для серьезных проблем среды выполнения или виртуальной машины, а `Exception` используется для всех остальных исключительных ситуаций.

90. \*\*Checked и unchecked exception:\*\*

- \*\*Checked exception (проверяемое исключение):\*\* Это исключение, которое должно быть обработано в коде программы с использованием блока `try ... catch` или объявлено в сигнатуре метода с помощью ключевого слова `throws`.

- \*\*Unchecked exception (непроверяемое исключение):\*\* Это исключение, которое не требуется объявлять в сигнатуре метода и может возникать во время выполнения программы. Непроверяемые исключения обычно наследуются от класса `RuntimeException`.

91. \*\*Нужно ли проверять checked exception?\*\*

Да, checked exception должны быть обработаны или объявлены в сигнатуре метода с помощью ключевого слова `throws`. Непроверенные исключения, такие как ошибки в программе или сбои в работе JVM, могут быть проглочены, но проверяемые исключения, такие как ошибки ввода-вывода или проблемы с сетью, должны быть обработаны.

92. \*\*Ключевое слово throws и его использование:\*\*

- Ключевое слово `throws` используется для объявления метода, который может возбуждать проверяемое исключение.

- Оно указывает на то, что метод может возбудить исключение определенного типа и передает ответственность за его обработку вызывающему коду.

- Метод, объявленный с ключевым словом `throws`, может возбуждать исключение, но не обязательно должен его обрабатывать.

Продолжим:

93. \*\*Способы обработки исключений:\*\*

В Java исключения можно обрабатывать с помощью следующих методов:

- Блок `try ... catch`: Используется для обработки исключений в определенном блоке кода.

- Блок `finally`: Используется для выполнения кода независимо от того, было ли исключение или нет.

- Ключевое слово `throws`: Используется для передачи исключения вызывающему коду.

- Блок `try-with-resources`: Используется для автоматического закрытия ресурсов после использования.

94. \*\*Правила для проверки исключений при наследовании:\*\*

При наследовании:

- Подкласс может выбрасывать проверенные исключения, которые являются подтипами исключений, выбрасываемых его суперклассом.

- Подкласс не может выбрасывать исключения, которые не выбрасывает его суперкласс.

95. \*\*Оператор, позволяющий принудительно выбросить исключение:\*\*

Оператор `throw` используется для явного выбрасывания исключения в программе. Пример:

```java

throw new IllegalArgumentException("Invalid argument");

```

96. \*\*Может ли метод `main` выбросить `throws`-исключение?\*\*

Да, метод `main` может объявить `throws` для исключений, но в большинстве случаев это не рекомендуется, так как метод `main` является точкой входа в программу, и нет другого кода, который мог бы обработать эти исключения. Если исключение выброшено в методе `main` и не обработано, программа завершится, и стек вызовов будет выведен на консоль.

Продолжим:

97. \*\*Достижение StackOverflowError:\*\*

`StackOverflowError` возникает, когда стек вызовов программы переполняется из-за слишком глубокой рекурсии или из-за создания слишком многих локальных переменных. Пример:

```java

public class Main {

public static void main(String[] args) {

recursiveMethod();

}

public static void recursiveMethod() {

recursiveMethod();

}

}

```

98. \*\*Случаи появления OutOfMemoryError:\*\*

`OutOfMemoryError` возникает, когда виртуальная машина Java (JVM) исчерпывает всю доступную память. Это может произойти из-за:

- Создания слишком больших объектов.

- Утечек памяти, когда объекты не освобождаются после использования.

- Загрузки слишком многих классов или ресурсов.

99. \*\*Назначение Garbage Collector:\*\*

Garbage Collector (сборщик мусора) в Java отвечает за автоматическое освобождение памяти, занятой объектами, которые больше не доступны. Он отслеживает объекты, которые больше не используются, и удаляет их, чтобы освободить память для других объектов.

100. \*\*Heap и Stack:\*\*

- \*\*Heap (куча):\*\* Это область памяти, в которой хранятся все объекты Java. Куча является общей памятью, доступной для всех потоков выполнения программы. Здесь хранятся объекты, создаваемые во время выполнения программы.

- \*\*Stack (стек):\*\* Это область памяти, используемая для хранения локальных переменных и вызовов методов каждого потока. Каждый поток имеет свой собственный стек, который используется для хранения данных о методах, их аргументах и локальных переменных.

Продолжим:

101. \*\*Пул строк (string pool) в Java:\*\*

Пул строк - это механизм в Java, который хранит строковые литералы в памяти для повторного использования. Когда создается новая строка литерал, JVM сначала проверяет пул строк. Если строка уже существует в пуле, JVM возвращает ссылку на существующий объект строки, а не создает новый объект. Это повышает эффективность использования памяти.

102. \*\*Процесс и поток:\*\*

- \*\*Процесс:\*\* Это экземпляр программы, запущенный в операционной системе. Процесс имеет собственное адресное пространство памяти и ресурсы, такие как файлы и сокеты.

- \*\*Поток:\*\* Это последовательность выполнения внутри процесса. Один процесс может содержать несколько потоков, которые могут выполняться параллельно или конкурентно.

103. \*\*Многопоточность и её использование в Java:\*\*

Многопоточность в Java позволяет выполнять несколько потоков выполнения внутри одного процесса. Она используется для:

- Улучшения производительности за счет параллельного выполнения задач.

- Увеличения отзывчивости приложений, позволяя выполнению задач в фоновом режиме.

- Улучшения использования ресурсов компьютера путем максимального использования процессорного времени.

104. \*\*Создание и запуск нового потока в Java:\*\*

В Java новый поток можно создать, унаследовав класс от `Thread` и переопределив метод `run()`, либо реализовав интерфейс `Runnable` и передавая его в конструктор класса `Thread`. После этого поток можно запустить вызовом метода `start()`.

Пример создания и запуска нового потока с использованием класса `Thread`:

```java

public class MyThread extends Thread {

public void run() {

System.out.println("Hello from a new thread!");

}

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start();

}

}

```

Пример создания и запуска нового потока с использованием интерфейса `Runnable`:

```java

public class MyRunnable implements Runnable {

public void run() {

System.out.println("Hello from a new thread!");

}

public static void main(String[] args) {

Thread thread = new Thread(new MyRunnable());

thread.start();

}

}

```

105. \*\*Способы синхронизации доступа к общим ресурсам в Java:\*\*

- Использование ключевого слова `synchronized` для синхронизации методов или блоков кода.

- Использование мониторов и методов `wait()`, `notify()` и `notifyAll()` для реализации взаимного исключения.(ждать и возобновить процессы)

- Использование синхронизированных коллекций из пакета `java.util.concurrent`.

- Использование конструкций с атомарными операциями из пакета `java.util.concurrent.atomic`.

- Использование блокировок (`Lock`) из пакета `java.util.concurrent.locks`.Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

106. \*\*Проблемы при работе с многопоточностью в Java:\*\*

- Состояние гонки (race condition).

- Дэдлоки (deadlocks).

- Потеря обновлений (lost updates).

- Зависание (livelock).

- Неопределенное поведение при использовании несинхронизированных данных.

107. \*\*Состояние гонки (race condition) и предотвращение в Java:\*\*

Состояние гонки возникает, когда несколько потоков пытаются одновременно доступаться к общему ресурсу и изменять его, что может привести к непредсказуемому поведению программы. Для предотвращения состояния гонки можно использовать синхронизацию с помощью ключевого слова `synchronized`, блокировок (`Lock`), или использовать атомарные операции.

108. \*\*Дэдлок (deadlock) и предотвращение в Java:\*\*

Дэдлок возникает, когда два или более потока ожидают друг друга для освобождения ресурсов, которые они сами удерживают, и не могут продолжить свое выполнение. Для предотвращения дэдлоков можно использовать следующие подходы:

- Избегать вложенных блокировок.

- Использовать таймауты при ожидании ресурсов.

- Избегать циклической зависимости между потоками.

- Использовать стратегии предоставления ресурсов в одном и том же порядке для всех потоков.

109. \*\*Классы и интерфейсы для работы с многопоточностью в Java:\*\*

Некоторые из классов и интерфейсов, используемых для работы с многопоточностью в Java, включают:

- `Thread`: Класс для создания и управления потоками.

- `Runnable`: Интерфейс для выполнения кода в отдельном потоке.

- `ThreadLocal`: Класс для создания потоковых переменных, доступных только в пределах потока.

- `Callable`: Интерфейс для выполнения вычислений, возвращающих результат и возникающих исключений.

- `Executor`: Интерфейс для выполнения асинхронных задач.

- `ExecutorService`: Интерфейс для управления жизненным циклом потоков в пуле потоков.

110. \*\*Методы и классы для синхронизации потоков в Java:\*\*

- `synchronized`: Ключевое слово для синхронизации доступа к общим ресурсам.

- `wait()`, `notify()`, `notifyAll()`: Методы класса `Object` для ожидания и уведомления других потоков.

- `Lock`, `ReentrantLock`: Интерфейс и его реализация для явной блокировки.

- `ReadWriteLock`: Интерфейс для блокировки чтения и записи.

- `Semaphore`, `CountDownLatch`, `CyclicBarrier`: Классы для управления параллельным выполнением.

111. \*\*Пул потоков (thread pool) в Java:\*\*

Пул потоков - это механизм управления и переиспользования потоков для выполнения асинхронных задач. Он содержит фиксированное количество потоков, готовых к выполнению задач из очереди. Пул потоков позволяет избежать создания и уничтожения потоков для каждой задачи, что улучшает производительность и эффективность использования ресурсов.

112. \*\*ForkJoinPool:\*\*

`ForkJoinPool` - это специализированный вид пула потоков, предназначенный для параллельного выполнения рекурсивных задач с разветвлением и слиянием. Он широко используется для вычислений, таких как разделение задачи на подзадачи, рекурсивные вычисления и обработка данных в параллельном режиме. `ForkJoinPool` предоставляет высокоуровневый интерфейс для работы с параллельными задачами, представленными в виде классов `ForkJoinTask` и `RecursiveTask`.

113. \*\*Агрегатные операторы:\*\*

Агрегатные операторы - это функции, которые выполняются над группой строк или значений и возвращают одно значение для каждой группы. В SQL такие операторы используются для вычисления сумм, средних значений, максимумов, минимумов и других агрегированных данных в результате запроса.

114. \*\*Нормальные формы в базах данных:\*\*

Нормальные формы - это формальные правила, определяющие структуру таблицы базы данных и ее зависимости от данных. Нормализация помогает устранить избыточность данных и предотвратить аномалии при обновлении, вставке и удалении данных. Некоторые из наиболее известных нормальных форм:

- Первая нормальная форма (1NF). нет массивов

- Вторая нормальная форма (2NF). нет зависимстей

- Третья нормальная форма (3NF).

- Нормальная форма Бойса-Кодда (BCNF).

- Четвертая нормальная форма (4NF).

- Пятая нормальная форма (5NF).

115. \*\*Индексы в базах данных:\*\*

Индексы - это структуры данных, используемые для ускорения операций поиска и сортировки данных в базах данных. Они позволяют быстро находить строки по значению столбца, для которого создан индекс. Некоторые из наиболее распространенных типов индексов:

- Индекс с одним столбцом.

- Индекс с несколькими столбцами.

- Уникальный индекс.

- Полнотекстовый индекс.

- Кластеризованный индекс.

116. \*\*Отличие WHERE от HAVING:\*\*

- `WHERE` используется для фильтрации строк до их группировки в операторе `GROUP BY`.

- `HAVING` используется для фильтрации групп после их группировки в операторе `GROUP BY`. Он применяется к агрегированным данным, например, сумме или среднему значению, и позволяет отфильтровать результаты по этим агрегированным значениям.

117. \*\*JOIN в SQL:\*\*

JOIN - это операция объединения таблиц в SQL, которая позволяет объединить строки из двух или более таблиц на основе условия соединения. Некоторые типы JOINs включают:

- INNER JOIN: Возвращает только строки, имеющие совпадения в обеих таблицах.

- LEFT JOIN (или LEFT OUTER JOIN): Возвращает все строки из левой таблицы и совпадающие строки из правой таблицы.

- RIGHT JOIN (или RIGHT OUTER JOIN): Возвращает все строки из правой таблицы и совпадающие строки из левой таблицы.

- FULL JOIN (или FULL OUTER JOIN): Возвращает строки, имеющие совпадения в любой из таблиц.

- CROSS JOIN: Возвращает декартово произведение двух таблиц.

118. \*\*Основные типы данных в SQL:\*\*

Некоторые из основных типов данных в SQL:

- INTEGER: Целочисленный тип данных.

- VARCHAR: Строковый тип переменной длины.

- CHAR: Строковый тип фиксированной длины.

- DATE: Тип данных для хранения даты.

- TIMESTAMP: Тип данных для хранения даты и времени.

- BOOLEAN: Логический тип данных.

119. \*\*Первичный ключ (Primary Key):\*\*

Первичный ключ - это уникальный идентификатор для каждой строки в таблице базы данных. Он обеспечивает уникальность идентификации строк и позволяет быстро находить, обновлять и удалять записи. Первичный ключ может состоять из одного или нескольких столбцов.

120. \*\*Spring Framework:\*\*

Spring Framework - это платформа для разработки Java-приложений, которая обеспечивает поддержку инверсии управления (IoC), внедрения зависимостей (DI) и множество других функций для разработки корпоративных приложений. Он предоставляет модульную структуру для управления различными аспектами приложения, такими как веб-приложения, транзакции, безопасность и другие. Преимущества Spring Framework включают упрощение разработки, повышение модульности и обеспечение устойчивости к изменениям.

121. \*\*Модули Spring Framework:\*\*

Spring Framework состоит из множества модулей, каждый из которых решает определенные задачи. Некоторые из основных модулей в Spring Framework включают:

- Spring Core Container: Основные функции IoC и DI.

- Spring AOP: Поддержка аспектно-ориентированного программирования.

- Spring JDBC: Упрощенный доступ к базам данных через JDBC.

- Spring ORM: Интеграция с ORM (Object-Relational Mapping) фреймворками.

- Spring Web: Поддержка создания веб-приложений.

- Spring Security: Механизмы аутентификации и авторизации.

122. \*\*Инверсия контроля (IoC) и внедрение зависимостей (DI):\*\*

Инверсия контроля (IoC) и внедрение зависимостей (DI) - это паттерны проектирования, которые позволяют управлять зависимостями между объектами в приложении. IoC - это паттерн проектирования, который определяет, что объекты должны зависеть от абстракций, а не от конкретных реализаций, и что объекты должны быть созданы и настроены вне зависимых классов. DI - это процесс, при котором IoC применяется для внедрения зависимостей в объекты. Другими словами, DI предоставляет объектам зависимости, которые им необходимы. Эти концепции позволяют избежать жестких связей между компонентами системы, что делает код более гибким, поддерживаемым и тестируемым.

123. \*\*Спринг бин (Spring Bean):\*\*

Спринг бин - это объект, управляемый контейнером Spring, который обычно создается и настраивается в контексте приложения. Для создания спринг бина необходимо объявить его в конфигурационном файле Spring (например, XML или Java-конфигурации) и указать все его зависимости.

124. \*\*Жизненный цикл бина в Spring:\*\*

Жизненный цикл бина в Spring состоит из следующих фаз:

- \*\*Создание (Instantiation):\*\* Контейнер Spring создаёт экземпляр бина.

- \*\*Инициализация (Initialization):\*\* Вызывается метод для инициализации бина, например, метод с аннотацией `@PostConstruct`.

- \*\*Использование (In Use):\*\* Бин используется в приложении.

- \*\*Уничтожение (Destruction):\*\* Когда бин больше не нужен, вызывается метод для уничтожения, например, метод с аннотацией `@PreDestroy`.

Продолжим:

125. \*\*Скоупы бина в Spring:\*\*

Скоупы бина определяют время жизни и доступность объекта-бина в контейнере Spring. Некоторые из основных скоупов бина в Spring:

- Singleton: Бин создается один раз за весь контекст приложения. По умолчанию используется скоуп Singleton.

- Prototype: Для каждого запроса создается новый экземпляр бина.

- Request: Создается новый экземпляр бина для каждого HTTP-запроса.

- Session: Создается новый экземпляр бина для каждой сессии HTTP.

- Custom: Возможность создания пользовательского скоупа.

126. \*\*Аннотации @Qualifier и @Primary:\*\*

Например, когда у нас есть несколько бинов удовлетворяющих условию для использования, то нужно выбрать какой из них нужно использовать idea сама не догадается поэтому есть эти две аннотации.

Аннотация @Primary задает бин, который будет внедрен по умолчанию (при отсутствии других указаний).

Аннотация @Qualifier позволяет уточнить имя бина, который надо внедрить. Используется прямо перед аргументом.

127. \*\*Проблемы при использовании Spring и их решения:\*\*

- Неоднозначность бинов: Использование `@Qualifier` для разрешения неоднозначности.

- Жизненный цикл и управление зависимостями: Использование правильных скоупов бинов и управление их жизненным циклом.

- Производительность: Оптимизация запросов к базе данных, кэширование данных, управление транзакциями.

- Ошибки конфигурации: Тестирование приложения, используя автоматизированные тесты.

128. \*\*Spring Boot:\*\*

Spring Boot - это фреймворк на основе Spring, который позволяет создавать автономные приложения Java с минимальной конфигурацией. Он облегчает развертывание и запуск приложений, так как содержит встроенные зависимости и сервер приложений. Некоторые из преимуществ Spring Boot:

- Упрощение конфигурации: Автоматическое управление зависимостями и конфигурациями.

- Встроенные серверы приложений: Поддержка встроенных серверов, таких как Tomcat, Jetty или Undertow.

- Автоматическое конфигурирование: Предоставление значений по умолчанию для многих настроек приложения.

- Улучшенное управление ошибками: Централизованная обработка ошибок и логирование.

129. \*\*Разница между Spring и Spring Boot:\*\*

- \*\*Spring:\*\* Spring - это обширный фреймворк для создания приложений на Java, который предоставляет решения для различных аспектов разработки, таких как управление зависимостями, внедрение зависимостей, аспектно-ориентированное программирование и т. д.

- \*\*Spring Boot:\*\* Spring Boot - это надстройка над фреймворком Spring, которая упрощает и ускоряет процесс создания приложений. Он предоставляет автоматическую конфигурацию, встроенные серверы приложений и множество других функций для быстрого старта проекта.

130. \*\*Разница между аннотациями @Service, @Component и @Repository:\*\*

- \*\*@Component:\*\* Эта аннотация используется для пометки класса как компонента Spring. Она является обобщенной аннотацией и может быть использована для любых компонентов.

- \*\*@Service:\*\* Эта аннотация используется для пометки класса как сервиса в приложении. Обычно используется для сервисного слоя приложения.

- \*\*@Repository:\*\* Эта аннотация используется для пометки класса как репозитория, обычно для классов, представляющих доступ к данным (например, базе данных).

131. \*\*Аннотации в Spring MVC:\*\*

В Spring MVC используются следующие аннотации:

1. @Controller: указывает, что класс является контроллером, который обрабатывает HTTP-запросы.

2. @RequestMapping: определяет URL-пути, на которые будет реагировать контроллер.

3. @ResponseBody: указывает, что ответ должен быть возвращен в формате JSON или XML, а не передан в представление.

4. @ModelAttribute: используется для передачи данных из формы в контроллер.

5. @SessionAttributes: управляет сохранением атрибутов сессии.

6. @RequestParam: используется для получения параметров запроса.

7. @PathVariable: используется для получения значений из частей URL-адреса.

8. @InitBinder: применяется для настройки данных формы перед их обработкой.

9. @Valid: указывает, что параметр должен быть проверен на наличие валидационных аннотаций.

10. @ExceptionHandler: определяет метод, который будет вызываться при возникновении исключения.

11. @ControllerAdvice: позволяет определить общие методы обработки исключений для всех контроллеров.

12. @Repository: указывает, что класс является репозиторием для доступа к данным.

13. @Service: указывает, что класс является сервисным слоем, предоставляющим бизнес-логику.

14. @Transactional: управляет транзакциями в рамках метода или класса.

15. @Autowired: используется для автоматического внедрения зависимостей.

16. @Value: позволяет получать значения из свойств контейнера.

17. @Qualifier: используется для явного указания инъекции конкретного bean.

18. @PostConstruct: метод, выполняемый после конструирования объекта и его инъекции зависимостей.

19. @PreDestroy: метод, выполняемый перед уничтожением объекта.

132. \*\*Аннотация @Value:\*\*

Аннотация @Value используется в Spring для внедрения значения из внешних источников (например, файла свойств) в поле класса или аргумент метода. Это позволяет легко настраивать значения в приложении без необходимости изменения кода.

133. \*\*Spring Boot Starter:\*\*

Spring Boot Starter - это зависимость Maven или Gradle, которая содержит набор библиотек и конфигураций, необходимых для разработки определенного типа приложения. Он упрощает начало работы с проектом, предоставляя все необходимые зависимости и настройки по умолчанию для разработки конкретного типа приложения, таких как веб-приложения, приложения для работы с данными и т. д.

134. \*\*Управление конфигурацией в Spring Boot:\*\*

Spring Boot предоставляет несколько средств для управления конфигурацией приложения:

- Файлы свойств (application.properties или application.yml): Используются для определения настроек приложения, таких как порт сервера, база данных и другие параметры.

- Внешние источники конфигурации: Spring Boot позволяет загружать настройки из внешних источников, таких как файлы свойств, переменные среды, системные свойства и другие.

- Аннотации: С помощью аннотаций можно настраивать различные компоненты приложения, такие как бины, контроллеры и другие.

135. \*\*Инструменты разработки и тестирования в Spring Boot:\*\*

Spring Boot предоставляет набор инструментов для упрощения разработки и тестирования приложений, включая:

- Встроенные серверы приложений: Spring Boot включает в себя встроенные серверы, такие как Tomcat, Jetty или Undertow, что упрощает запуск и тестирование приложений.

- Встроенные зависимости: Spring Boot автоматически добавляет зависимости, такие как Spring Web MVC, Spring Data JPA и другие, что упрощает конфигурацию проекта.

- Автоматическая конфигурация: Spring Boot автоматически настраивает многие аспекты приложения на основе классов на пути классов, аннотаций и файлов свойств.

136. \*\*Репозиторий в Spring Data:\*\*

Репозиторий в Spring Data - это интерфейс, который предоставляет удобный способ взаимодействия с базой данных. Он предоставляет CRUD (Create, Read, Update, Delete) операции для работы с данными без необходимости написания SQL-запросов вручную. Репозиторий определяется интерфейсом и наследуется от специальных интерфейсов Spring Data, таких как JpaRepository. При запуске приложения Spring Boot автоматически создает реализацию репозитория и инъектирует его в контекст приложения.

137. \*\*Подключение к базе данных с помощью Spring Data:\*\*

Для подключения к базе данных с помощью Spring Data необходимо выполнить следующие шаги:

- Определить зависимости Maven или Gradle для JDBC-драйвера вашей базы данных и Spring Data.

- Создать конфигурационный файл (например, application.properties или application.yml) для указания параметров подключения к базе данных.

- Создать репозиторий, который расширяет один из интерфейсов Spring Data (например, JpaRepository) и определить методы для доступа к данным.

- Использовать репозиторий в вашем приложении для выполнения операций CRUD с данными.

138. \*\*Требования к энтити-классу:\*\*

Энтити-класс - это класс, который представляет таблицу в базе данных. Он должен соответствовать определенным требованиям:

- Должен иметь аннотацию `@Entity`, чтобы Spring Data мог распознать его как объект, с которым он будет работать.

- Должен иметь обязательное поле, которое является первичным ключом (аннотация `@Id`).

- Может иметь аннотацию `@Table`, чтобы указать имя таблицы, если оно отличается от имени класса.

139. \*\*Аннотация @Query:\*\*

Аннотация `@Query` используется в Spring Data JPA для определения пользовательских SQL-запросов, которые могут быть выполнены с помощью репозитория. Она позволяет разработчику определять более сложные запросы, которые не могут быть сформулированы с помощью стандартных методов репозитория. Кроме того, `@Query` может быть использована для оптимизации выполнения запросов, включая использование именованных параметров, поддержки параметров метода и других возможностей.

Продолжим:

140. \*\*Фетч-стратегия (Fetch Strategy):\*\*

Фетч-стратегия определяет, как JPA (Java Persistence API) будет извлекать связанные сущности из базы данных при выполнении запросов. Она влияет на то, как JPA загружает и связывает связанные объекты в сессии сущностей. Основные фетч-стратегии в JPA:

- \*\*Eager (жадная загрузка)\*\*: При использовании этой стратегии связанные сущности загружаются сразу, вместе с основной сущностью.

- \*\*Lazy (ленивая загрузка)\*\*: При использовании этой стратегии связанные сущности загружаются только по требованию, когда к ним происходит доступ.

141. \*\*Проблема N+1:\*\*

Проблема N+1 - это ситуация, когда при использовании ленивой загрузки встречается неэффективность в запросах к базе данных. Она возникает, когда загружается коллекция объектов, а затем для каждого объекта в этой коллекции делается дополнительный запрос для загрузки связанных сущностей. Например, если у нас есть коллекция объектов `Parent`, и для каждого из них делается дополнительный запрос для загрузки его дочерних объектов `Child`, это приведет к выполнению N+1 запросов (1 запрос для загрузки `Parent` и N запросов для загрузки `Child`). Это может привести к значительному увеличению количества запросов к базе данных и снижению производительности приложения.