Вопросы и ответы на собеседование по основам Java

(восьмая часть)

211). HashMap и его особенности

HashMap - это реализация интерфейса Map, которая представляет собой **хэш-таблицу**. Вот основные характеристики HashMap:

1. **Неупорядоченность**: HashMap не гарантирует порядок элементов. Это означает, что элементы могут храниться и извлекаться в произвольном порядке. Если вам важен порядок элементов, вы можете использовать LinkedHashMap, который сохраняет порядок вставки элементов.
2. **Уникальные ключи**: HashMap не может содержать дубликаты ключей. Если вы попытаетесь добавить элемент с ключом, который уже существует в мапе, старое значение будет перезаписано новым значением.
3. **Быстрый доступ к элементам**: Одной из ключевых особенностей HashMap является быстрый доступ к значениям по ключу. Это достигается с использованием хэш-таблицы, которая использует хэш-функцию для быстрого поиска местоположения элемента в таблице.
4. **Хэширование ключей**: Ключи, добавленные в HashMap, должны быть хорошо реализованными объектами и должны правильно реализовывать методы **hashCode**() и **equals**(). Эти методы используются для определения уникальности ключей и разрешения коллизий.
5. **Поддержка null**: HashMap может содержать один ключ со значением null и несколько значений со значением null. Это позволяет представлять отсутствие значения для ключа.
6. **Эффективность**: HashMap обеспечивает высокую производительность для операций добавления, поиска и удаления элементов. Это делает ее хорошим выбором для множества задач, таких как кэширование, хранение настроек и т. д.

HashMap не является потокобезопасной структурой данных. Если необходимо обеспечить безопасность при доступе к HashMap из нескольких потоков, следует использовать **ConcurrentHashMap** или синхронизировать доступ к HashMap вручную

212). Как работает HashMap при добавлении элемента?

1. **Вычисление хэш-кода ключа**:
   - Когда вы добавляете элемент в HashMap, система вызывает метод **hashCode**() для ключа этого элемента. Метод hashCode() возвращает целое число, которое называется хэш-кодом.
   - Цель хорошей хэш-функции состоит в том, чтобы преобразовать ключ в уникальное целое число. Это число будет использоваться для определения индекса внутри массива.
2. **Определение индекса в массиве**:
   - Полученный хэш-код преобразуется в индекс внутри массива HashMap. Для этого обычно используется операция взятия остатка от деления хэш-кода на размер массива.
3. **Добавление элемента в связанный список**:
   - Если в этой ячейке массива (по полученному индексу) уже есть элементы, HashMap использует связанный список, чтобы хранить их.
   - Далее, HashMap сравнивает хэш и ключ нового элемента поочередно с хэшами и ключами элементов в связанном списке.
   - Если хэш и ключ нового элемента совпадают с хэшами и ключами одного из элементов из списка, значение этого элемента перезаписывается новым значением.
   - Если не найдено совпадения хэшей и ключей, новый элемент добавляется в конец связанного списка.

Важно отметить, что процесс добавления элемента выполняется очень быстро, так как HashMap может найти местоположение элемента с использованием хэш-кода ключа и не требует перебора всех элементов массива.

213). Hashtable

Hashtable - это структура данных, предназначенная для хранения пар ключ-значение. Она была доступна до введения коллекций (collection framework) в Java и была частью исторической Dictionary API. Вот основные характеристики Hashtable:

1. **Реализация старого API**: Hashtable была частью оригинального API Java и расширяла класс Dictionary. С введением коллекций (collection framework) Hashtable была адаптирована для интерфейса Map.
2. **Хранение ключей и значений**: Hashtable предоставляет удобный способ хранить пары ключ-значение. Вы можете использовать ключи для быстрого доступа к значениям.
3. **Отсутствие допустимых значений null**: Hashtable не разрешает использование ключей или значений со значением null. Попытка вставки null будет приводить к исключению **NullPointerException**.
4. **Синхронизация**: Hashtable является синхронизированной структурой данных, что означает, что она обеспечивает потокобезопасность. Это может быть полезно в многопоточных приложениях, но также может вызывать снижение производительности в однопоточных сценариях.

Hashtable была одной из первых структур данных для хранения ключ-значение в Java, но из-за своей синхронизации и ограничений в использовании null она часто заменяется более современными альтернативами, такими как HashMap из коллекций (collection framework).

214). Разница между HashMap и Hashtable

ХАРАКТЕРИСТИКА	HASHMAP	HASHTABLE
Синхронизация	Не синхронизирована	Синхронизирована
Потокобезопасность	Не является потокобезопасной	Является потокобезопасной
Допустимость null	Допускает ключи и значения null	Не допускает ключи и значения null
Наследование	Расширяет интерфейс Map	Расширяет Dictionary, а затем интерфейс Map
Эффективность	Обычно более эффективна в однопоточных сценариях из-за отсутствия синхронизации	Может вызывать снижение производительности в однопоточных сценариях из-за синхронизации
Использование	Рекомендуется для новых приложений, если не требуется синхронизация	Редко используется в современных приложениях из-за синхронизации и ограничения в использовании null
История	Введена в Java 1.2 (collection framework)	Исторически доступна в Java до введения коллекций

215). Разница между ArrayList и LinkedList

ХАРАКТЕРИСТИКА	ARRAYLIST	LINKEDLIST
Реализация	Реализация на основе динамического массива	Реализация на основе двусвязного списка
Добавление элемента	Добавление элемента в конец массива эффективно	Добавление элемента в начало/конец списка эффективно, в середину - менее эффективно
Удаление элемента	Удаление элемента по индексу эффективно	Удаление элемента из начала/конца списка эффективно, в середине - менее эффективно
Доступ к элементам	Доступ по индексу быстрый	Доступ к элементам по индексу менее эффективен, но быстрый доступ к началу/концу списка
Итерация	Итерация по индексам быстрая	Итерация по элементам списка быстрая
Память	Занимает меньше памяти из-за отсутствия ссылок на следующие элементы	Занимает больше памяти из-за наличия ссылок на следующие и предыдущие элементы
Использование	Рекомендуется, когда требуется быстрый доступ по индексу и частые операции добавления/удаления в конец массива	Рекомендуется, когда требуется быстрый доступ к началу/концу списка и операции добавления/удаления из начала/конца списка. Эффективно при частой вставке и удалении элементов в середине списка.

216). Разница между Comparator и Comparable

Read & Write Code

ХАРАКТЕРИСТИКА	COMPARABLE	COMPARATOR
Интерфейс	Comparable - интерфейс, который класс реализует сам для сравнения себя	Comparator - интерфейс, предоставляющий внешний компаратор для сравнения объектов
Методы	Класс реализует метод compareTo(Object obj), который определяет, как сравнивать текущий объект с другим объектом	Внешний класс реализует метод compare(Object obj1, Object obj2), который определяет, как сравнивать два объекта, которые необходимо сравнить
Цель	Используется для сравнения объектов того же типа	Используется для сравнения объектов разных типов или для настройки альтернативных способов сравнения объектов
Реализация	Реализуется внутри класса объекта, который вы хотите сравнить	Реализуется в отдельном классе, который предоставляет компаратор для сравнения объектов
Естественный порядок сортировки	Определяется внутри класса (если реализован Comparable)	Может быть определен внешним образом для любых объектов с использованием компаратора

216). Продолжение

ХАРАКТЕРИСТИКА	COMPARABLE	COMPARATOR
Изменение порядка сортировки	Изменить порядок сортировки можно только путем изменения кода внутри класса	Порядок сортировки можно изменить, создав новый компаратор и передав его в метод сортировки

Пример	public class MyClass implements Comparable<MyClass>{
	
	public int compareTo(MyClass other) {
	
	// логика сравнения
	
	}
	}
		public class MyComparator implements Comparator<MyClass>{
		
		public int compare(MyClass obj1, MyClass obj2) {
		
		// логика сравнения
		
		}
		}

217). Что вы знаете о нотации "большое O" (Big-O) и можете ли вы дать примеры, касающиеся различных структур данных?

Нотация "большое O" (**Big-O**) просто описывает, насколько хорошо алгоритм масштабируется или выполняется в худшем случае при увеличении количества элементов в структуре данных. Нотация "большое O" также может использоваться для описания других характеристик, таких как потребление памяти. Поскольку классы коллекций на самом деле представляют собой **структуры данных**, мы обычно используем нотацию "большое O" для выбора наилучшей реализации.

Вот некоторые примеры использования нотации "большое O" для разных структур данных:

1. Список (**List**):
   - **ArrayList**: Вставка и доступ к элементам выполняются за **O(1)** (константное время) в среднем случае. Однако вставка или удаление элемента в середине списка может потребовать **O(n)** (линейное время) для смещения элементов.
   - **LinkedList**: Вставка и удаление элементов в середине списка выполняются за **O(1)** (константное время), но доступ к элементам может потребовать **O(n)** (линейное время) в худшем случае.
2. Множество (**Set**):
   - **HashSet**: Вставка, удаление и поиск элементов выполняются в среднем за **O(1)** (константное время). Однако в худшем случае сложность может быть **O(n)** из-за коллизий.
   - **TreeSet**: Вставка, удаление и поиск элементов выполняются в среднем за **O(log n)** (логарифмическое время).
3. Хеш-таблица (**HashMap**):
   - **HashMap**: Вставка, удаление и поиск элементов выполняются в среднем за **O(1)** (константное время) при хорошем распределении хешей. В худшем случае сложность может быть **O(n)** из-за коллизий.
4. Дерево (**Tree**):
   - **BinarySearchTree**: Вставка, удаление и поиск элементов выполняются в среднем за **O(log n)** (логарифмическое время), если дерево сбалансировано. В худшем случае сложность может быть **O(n)** для несбалансированных деревьев.

Это всего лишь некоторые примеры, и нотация "большое O" помогает сравнивать производительность разных структур данных и выбирать наилучший инструмент для конкретной задачи, учитывая ожидаемый объем данных и операции, которые нужно выполнить.

218). Что такое приоритетная очередь (PriorityQueue) в Java?

PriorityQueue - это неограниченная очередь, основанная на приоритетной куче, и ее элементы упорядочены в естественном порядке.
При создании PriorityQueue мы можем предоставить компаратор, который отвечает за упорядочивание элементов в очереди.
PriorityQueue не допускает значения null, объекты, которые не предоставляют естественного упорядочивания, или объекты, для которых нет ассоциированного компаратора.

Наконец, Java PriorityQueue **не является потокобезопасной** и требует **O(log(n))** времени для операций вставки и извлечения элементов.

219). ConcurrentHashMap

ConcurrentHashMap - это реализация хэш-таблицы, предназначенная для использования в многопоточных средах. Он включен в пакет **java.util.concurrent** и предоставляет механизмы синхронизации для безопасной работы с данными **в нескольких потоках**. Вот основные характеристики и особенности ConcurrentHashMap:

1. **Многопоточная безопасность**: Несколько потоков могут одновременно читать и модифицировать данные в ConcurrentHashMap без блокировки всей структуры данных.
2. **Распараллеливание операций**: Разделяет свое внутреннее хранилище на несколько сегментов (**buckets**), и каждый сегмент может блокироваться отдельно. Это позволяет нескольким потокам выполнять операции чтения и записи **параллельно** в разных сегментах, что повышает производительность в сценариях многопоточного доступа.
3. **Отсутствие блокировок для части записей**: Для некоторых операций записи, таких как **put**(), **remove**(), блокировки применяются только к соответствующим сегментам, а не ко всей таблице. Это позволяет одновременно выполнять множество операций записи в разных сегментах.
4. **Ограниченная синхронизация**: ConcurrentHashMap достигает своей безопасности благодаря синхронизации только в тех частях, где это необходимо, минимизируя таким образом ограничение на производительность.
5. **Отсутствие поддержки null**: Не разрешает ключи или значения равными null, что отличается от обычной HashMap.
6. **Поддержка атомарных операций**: Предоставляет атомарные операции, такие как **putIfAbsent**(), **remove**(key, value), **replace**(key, oldValue, newValue) и другие, которые позволяют выполнять операции безопасно в многопоточной среде.

ConcurrentHashMap **является мощным** инструментом для обеспечения безопасного и эффективного доступа к данным в многопоточных приложениях, где производительность и безопасность играют важную роль.

220). Различия между ConcurrentHashMap, Hashtable и Collections.synchronizedMap

Различия между **ConcurrentHashMap**, **Hashtable** и **Collections.synchronizedMap** можно разделить на следующие аспекты:

Механизм блокировки:

- ConcurrentHashMap использует более современный и эффективный механизм блокировки, называемый "lock striping". Он разделяет хэш-таблицу на множество сегментов (buckets) и блокирует только один сегмент, позволяя другим потокам параллельно работать с другими сегментами. Это обеспечивает более высокую параллельность и масштабируемость.
- Hashtable использует устаревший механизм синхронизации, который блокирует всю структуру данных при выполнении любой операции, что может вызывать блокировку для всех потоков, даже если они работают с разными ключами.
- Collections.synchronizedMap оборачивает обычную HashMap в синхронизированный объект, что также блокирует всю структуру данных при выполнении операций.

Concurrency и ConcurrentModificationException:

- ConcurrentHashMap обеспечивает более высокую параллельность, так как блокирует только сегменты, а не всю структуру данных. Его итераторы не выбрасывают ConcurrentModificationException, что означает, что несколько потоков могут безопасно использовать итераторы параллельно.
- Hashtable и Collections.synchronizedMap используют общую блокировку для всех операций, что ограничивает параллелизм и может вызывать ConcurrentModificationException при одновременных изменениях.

Null Values:

- ConcurrentHashMap не разрешает ключи или значения, равные null.
- Hashtable также не разрешает ключи или значения, равные null.
- Collections.synchronizedMap может разрешать ключи и значения, равные null, в зависимости от базовой HashMap.

Итог: ConcurrentHashMap обычно предпочтительнее в многопоточных приложениях из-за своего более эффективного механизма блокировки и более высокой параллельности. Hashtable и Collections.synchronizedMap остаются вариантами для старых приложений, но они менее эффективны и более склонны к блокировкам при высокой нагрузке.

221). CopyOnWriteArrayList и когда его следует использовать?

CopyOnWriteArrayList - это одна из реализаций интерфейса List в Java, предназначенная для работы в многопоточных приложениях. Этот класс обеспечивает потокобезопасность при доступе к списку из нескольких потоков, особенно при ситуациях, когда одни потоки выполняют итерацию по списку, а другие модифицируют его содержимое.

Основной принцип работы заключается в создании **копии списка каждый раз**, когда он изменяется (добавление, удаление или изменение элементов). Это означает, что при изменении списка **оригинальный список не меняется**, а создается **его копия с внесенными изменениями**. Это позволяет избежать конфликтов при одновременном доступе итераторов и потоков, выполняющих модификации.

Преимущества :

1. **Потокобезопасность**: Класс обеспечивает потокобезопасность без необходимости явной синхронизации.
2. **Итерация**: Можно выполнять итерацию по списку во время модификации без опасности выбросить ConcurrentModificationException.
3. **Простота использования**: Не требует написания собственной синхронизации при работе с множеством потоков.

Однако есть и **недостатки**:

1. **Высокий расход памяти**: Поскольку каждая модификация создает копию, это может потреблять много памяти для больших списков.
2. **Устаревшие данные**: Из-за копирования, итераторы могут работать с устаревшими данными, если много изменений происходит в списке.

CopyOnWriteArrayList следует использовать в тех случаях, когда часто происходит итерация по списку, и изменения в списке происходят редко или в меньшем объеме. Этот класс не подходит для ситуаций, когда изменения в списке происходят часто, так как копирование списка при каждой модификации может быть неэффективным с точки зрения производительности и потребления памяти.

222). В чем разница между HashSet и TreeSet?

HashSet реализован с использованием хеш-таблицы и, следовательно, его элементы не упорядочены.
Методы **add**, **remove** и **contains** в HashSet имеют постоянную временную сложность **O(1)**. С другой стороны, TreeSet реализован с использованием структуры дерева.
Элементы в TreeSet упорядочены, и поэтому методы **add**, **remove** и **contains** имеют временную сложность **O(log n).**

223). Объясните, что такое "fail-fast" итераторы в Java?

Когда итератор выполняет итерацию по коллекции, коллекция не должна изменяться, за исключением изменений, внесенных этим итератором. Изменение означает, что коллекцию нельзя модифицировать другим потоком, пока другой поток выполняет итерацию; если такая модификация все же произойдет, то будет сгенерировано исключение ConcurrentModificationException. Итераторы, которые проявляют такое поведение, называются "**fail-fast**" (быстро завершающими) итераторами.

Примеры коллекций, которые используют "fail-fast" итераторы, включают ArrayList, HashSet и HashMap. Практически все итераторы, реализованные в библиотеке коллекций Java, являются "fail-fast".

Fail-fast итераторы действуют для обеспечения целостности коллекции и предотвращения некорректных изменений во время итерации. Если коллекция модифицируется внутри цикла итератора **другим потоком** (или в другом контексте), итератор замечает эту модификацию и мгновенно генерирует **исключение**, предотвращая тем самым непредсказуемое поведение или повреждение данных.

Важно помнить, что "fail-fast" итераторы не предназначены для использования в многопоточных сценариях без соответствующей синхронизации. Для безопасной итерации по коллекциям из нескольких потоков следует использовать средства синхронизации, такие как synchronized блоки или конкурентные коллекции, чтобы избежать исключений **ConcurrentModificationException** и обеспечить правильное взаимодействие между потоками.

224). Что такое сериализация в Java?

Serialization в Java - это процесс преобразования объекта в последовательность байтов, чтобы его можно было передавать по сети или сохранять в файле и воссоздавать позднее. Сериализованный объект представляет собой объект, представленный как последовательность байтов, которая включает в себя данные объекта, тип объекта и типы данных, хранящиеся в объекте.
Процесс сериализации позволяет сохранять состояние объекта, включая его поля и значения, так чтобы оно можно было передать или сохранить, например, для постоянного хранения или обмена данными между разными приложениями или на разных уровнях одного приложения. Сериализация особенно полезна при передаче объектов между клиентом и сервером в распределенных приложениях или при сохранении состояния объектов в файловой системе.
Для выполнения сериализации объект в Java должен реализовать интерфейс Serializable. Когда объект сериализуется, его поля и состояние сохраняются в бинарном формате. Затем этот бинарный поток байтов можно передать по сети или сохранить в файле. Позднее объект можно десериализовать, то есть воссоздать из байтового представления, чтобы получить точную копию исходного объекта.
Примеры использования сериализации включают в себя сохранение состояния игры, передачу объектов данных между клиентом и сервером, а также кеширование объектов. Важно отметить, что не все классы могут быть сериализованы, и некоторые поля могут быть помечены как **transient**, чтобы исключить их из процесса сериализации.

225). Альтернативы сериализации в Java

Альтернативы сериализации в Java включают в себя следующие:

1. Передача данных на основе **XML** (XML-based data transfer): Вместо сериализации объектов в бинарном формате, вы можете преобразовывать данные объектов в формат XML и передавать их между системами. Для этого вы можете использовать технологии, такие как JIBX (Java-to-XML Binding) или **JAXB** (Java Architecture for XML Binding), которые позволяют маршалировать данные объектов в XML и демаршалировать XML обратно в объекты.
2. Передача данных на основе **JSON** (JSON-based data transfer): JSON (JavaScript Object Notation) - это легковесный формат обмена данными, который широко используется для передачи данных между приложениями. Вы можете преобразовывать данные объектов в формат JSON и передавать их. В Java для работы с JSON существует множество библиотек, таких как Jackson, Gson и org.json, которые облегчают маршалинг и демаршалинг данных.
3. Эти альтернативы предоставляют более читаемый и человеко-читаемый формат данных, чем бинарная сериализация, и могут быть полезными в сценариях, где важна человеко-читаемость данных или взаимодействие с другими системами, которые не являются Java-ориентированными. Однако, при использовании XML или JSON для передачи данных, вы теряете некоторые преимущества, такие как быстродействие и оптимизированный размер данных, которые может предоставить бинарная сериализация.

226). Интерфейс Serializable в Java

Интерфейс Serializable предназначен для обеспечения возможности сериализации объектов класса. Этот интерфейс находится в пакете **java.io** и служит как маркерный интерфейс, то есть он не содержит никаких абстрактных методов для реализации.

Класс, который реализует интерфейс Serializable, предоставляет информацию виртуальной машине Java (JVM) о том, что его экземпляры могут быть сериализованы в байтовое представление и десериализованы обратно.

Syntax:

```
public interface Serializable {
}
```

227). Как сделать объект сериализуемым в Java?

Чтобы сделать объект сериализуемым в Java, выполните следующие шаги:

1. Нашему классу необходимо реализовать интерфейс **Serializable**. Если наш объект содержит другие объекты, то и эти классы также должны реализовать интерфейс Serializable.
2. Мы используем **ObjectOutputStream**, который расширяет OutputStream и используется для записи объектов в поток.
3. Мы используем **ObjectInputStream**, который расширяет InputStream и используется для чтения объектов из потока.

228). Что такое серийный идентификатор версии (Serial Version UID) и какова его важность в Java?

Серийный идентификатор версии (Serial Version UID, SUID) в Java - это 64-битное длинное значение, которое служит уникальным идентификатором для сериализуемого класса. Его важность заключается в следующем:

1. **Управление версией**: SUID используется для управления версией класса во время процесса сериализации и десериализации. Это позволяет гарантировать, что сериализованное представление класса совместимо с текущим определением класса.
2. **Совместимость вперед и назад**: При обновлении класса, например, добавлении или удалении полей или изменении структуры класса, SUID помогает обеспечить совместимость с ранее сериализованными объектами. При сохранении одного и того же SUID для класса, старые сериализованные объекты могут быть десериализованы с новой версией класса, и отсутствующие или новые поля будут обработаны корректно.
3. **Предотвращение несовместимой десериализации**: Если SUID не совпадает между сериализованным объектом и классом, пытающимся его прочитать, Java генерирует исключение InvalidClassException. Это предотвращает десериализацию объектов с несовместимыми определениями классов и помогает обеспечить целостность данных.
4. **Явное управление версией**: Вы можете явно управлять версией вашего класса, указывая пользовательский SUID (с помощью поля serialVersionUID). Это может быть полезно в случаях, когда вы хотите предотвратить непреднамеренную совместимость между классами с одинаковыми именами, но разной структурой.

Вот пример явного указания пользовательского SUID в классе:

```
import java.io.Serializable;

public class МойКласс implements Serializable {
    private static final long serialVersionUID = 123456789L; // Пользовательский SUID// Поля и методы класса
}
```

Итак, серийный идентификатор версии (Serial Version UID) в Java играет важную роль в обеспечении совместимости при сериализации и десериализации объектов, а также в контроле версий классов.

229). Что происходит, если мы не определяем серийный идентификатор версии (serial version UID)?

Если мы не определяем собственный серийный идентификатор версии (serialVersionUID) в нашем сериализуемом классе, JVM автоматически создаст один для нас на основе структуры класса, его полей и сигнатур методов. Однако это имеет несколько последствий:

1. **Динамический расчет**: Если вы опускаете serialVersionUID, JVM будет вычислять его динамически на основе структуры класса и переменных-членов каждый раз при сериализации или десериализации объекта этого класса. Этот процесс включает вычисление хэш-кода всех свойств класса, что может быть вычислительно затратным и влиять на производительность сериализации и десериализации.
2. **Управление версией**: Без явно определенного serialVersionUID, автоматически созданный будет изменяться, если вы измените структуру класса. Это может привести к несовместимости версий, когда пытаетесь десериализовать старые объекты, созданные с предыдущими версиями класса, в новой версии класса.
3. **Сериализация статических полей**: Обратите внимание, что статические поля не могут быть сериализованы. Исключение составляет serialVersionUID, который будет сериализован вместе с объектом, даже если он является статическим полем.

В итоге, хотя JVM может автоматически создать serialVersionUID, рекомендуется явно указывать собственный serialVersionUID, чтобы иметь более точный контроль над версией класса и избежать нежелательных несовместимостей при сериализации и десериализации объектов.

230). Можем ли мы сериализовать статические переменные в Java?

Нет, вы не можете сериализовать статические переменные в Java.
Причина заключается в том, что статические переменные связаны с классом, а не с отдельным объектом.

Механизм сериализации в Java разработан для сохранения и восстановления состояния отдельных объектов, а не состояния класса.

Поскольку статические переменные являются общими для всех экземпляров класса и принадлежат самому классу, они не являются частью состояния объекта и, следовательно, не сериализуются.

При сериализации объекта механизм сериализации сохраняет только состояние переменных экземпляра объекта, которое специфично для этого объекта.
Статические переменные, так как они связаны с классом и не привязаны к конкретному объекту, не включаются в процесс сериализации.

Если вам нужно сохранить или передать данные, связанные с классом (состояние класса), вы обычно делаете это отдельно от сериализации, например, записывая данные, специфичные для класса, в файл или базу данных или используя другие механизмы, которые не связаны с сериализацией объектов.

231). Когда мы сериализуем объект, сохраняет ли механизм сериализации и его ссылки?

Когда вы сериализуете объект в Java, механизм сериализации **также** сохраняет его ссылки **на другие объекты**.
Однако существуют определенные условия, которые должны быть выполнены для успешной сериализации ссылок.

Для того чтобы ссылки были успешно сериализованы, объект, на который указывают ссылки, также должен реализовать интерфейс Serializable.
В этом случае объекты, на которые указывают ссылки, также будут сериализованы.
Если вы не сделаете сериализуемыми объекты, на которые есть ссылки, то вы получите исключение **NotSerializableException**

232). Если мы не хотим, чтобы некоторые поля не сериализовались. Как это сделать?

Если вы не хотите, чтобы некоторые поля сериализовывались во время сериализации, вы можете объявить эти переменные как **transient**. Во время десериализации переменные, объявленные как transient, будут инициализированы значениями **по умолчанию** для примитивных типов данных и значением **null** для ссылок на объекты.

233). Какая связь между методами equals и hashCode?

Если у двух объектов одного и того же класса содержимое одинаковое, то и хеш коды должны быть одинаковые (**но не наоборот**).
Если объекты не равны по результатам выполнения метода equals , тогда их hashcode могут быть как одинаковыми, так и разными.
Однако для повышения производительности, лучше, чтобы разные объекты возвращали
разные коды.

234). Для чего предназначена сборка мусора (garbage collection) в Java?

Целью сборки мусора является выявление и удаление тех объектов, которые больше не нужны приложению, чтобы чтобы ресурсы были утилизированы и повторно использованы.

235). Что делают методы System.gc() и Runtime.gc()?

Эти методы могут быть использованы как подсказка для JVM (Java Virtual Machine), чтобы начать сборку мусора. Однако решение о том, начать ли сборку мусора немедленно или позже, остается за самой JVM.

236). Какова структура кучи(Heap) Java? Что такое Perm Gen space в куче?

У JVM есть куча (**heap**), которая представляет собой область данных времени выполнения, из которой выделяется память для всех экземпляров классов и массивов. Она создается при запуске JVM. Память кучи для объектов освобождается автоматической системой управления памятью, известной как сборщик мусора (garbage collector). Память кучи состоит из живых и мертвых объектов. Живые объекты доступны приложению и не подлежат сборке мусора. Мертвые объекты – это те, которые больше недоступны приложению, но еще не были собраны сборщиком мусора. Такие объекты занимают пространство в памяти кучи до тех пор, пока их в конечном итоге не соберет сборщик мусора.

Perm Gen (**Permanent Generation**) или просто PermSpace (Permanent Space) - это одна из областей кучи Java, которая используется для хранения метаданных классов, методов и другой информации, которая остается постоянной во время выполнения приложения. Это включает в себя данные, такие как код классов, информацию о структуре классов и другие метаданные, необходимые для работы приложения. Perm Gen был частью кучи в старых версиях Java, но начиная с Java 8, Perm Gen был заменен на область метаданных (**Metaspace**), что улучшило управление памятью и избавило от многих проблем, связанных с Perm Gen.

237). Какие различия между String, StringBuilder , StringBuffer

Класс **String** является **immutable** , т.е. любое изменение строки приводит к созданию нового объекта.
Классы **StringBuilder** , **StringBuffer** являются **mutable** .
StringBuilder был добавлен в Java 5.0 он во всем идентичен классу StringBuffer , за исключением того, что он не синхронизирован, что делает его значительно быстрее.

238). Что такое пул строк?

Пул строк(**String Pool**) это набор строк, который хранится в памяти Java **heap**. Когда строка создается с использованием **литералов** (например, **"Hello"**), JVM проверяет пул строк на наличие такой строки. Если такая строка уже существует в пуле, то вместо создания новой строки она ссылается на существующую.

Пример

```
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");
```

Если вам нужно явно добавить строку в пул, вы можете использовать метод intern():

```
String s4 = s3.intern(); // Добавляет строку в пул строк и возвращает ссылку на нее

System.out.println(s1==s2); //true
System.out.println(s1==s3); // false
System.out.println(s1==s4); // true
```

239). Что такое пул Integer?

Если мы создаем Integer в этом промежутке **[-128;127]** без использования ключевого слова **new** , то вместо того, чтобы каждый раз создавать новый объект, JVM берет их из пула.

Пример
Integer i1 =10; // Используется ссылка на объект Integer из пула (если 10 находится в диапазоне -128 до 127)

Integer i2 = 10; // Используется та же ссылка на объект Integer из пула

Integer i3 = new Integer(10);
Integer i4 = 130;
Integer i5 = 130;
System. out .println (i1 == i2); //**true**
System. out .println (i1 == i3); //**false**
System. out .println (i4 == i5); //**false**

240). Какие шаблоны проектирования вы знаете?

Порождающие - описывают создание объектов (**Singleton**, **Builder**, **AbstractFactory**..)
Структурные - определяют отношения между классами и объектами (**Adapter**, **Facade**..)
Поведенческие - упрощают взаимодействие между объектами (**Iterator** , **Strategy**..)