Вопросы и ответы на собеседование по основам Java

(пятая часть)

121). Что такое межпоточная связь и как она происходит в Java?

Межпоточное взаимодействие в Java позволяет потокам взаимодействовать между собой, чтобы выполнить связанные задачи синхронизированно. Межпоточная связь в Java осуществляется с помощью следующих трех методов:

1. **wait**()
2. **notify**()
3. **notifyAll**()

122). Расскажите про методы wait(), notify() и notifyAll()

Методы wait(), notify() и notifyAll() из класса Object используются для управления потоками и обеспечения взаимодействия между ними:

1. **wait**(): Метод wait() вызывается на объекте и заставляет текущий поток "уснуть" и освобождает монитор объекта. Поток будет ожидать, пока другой поток не вызовет метод notify() или notifyAll() на том же объекте, чтобы разбудить его.
2. **notify**(): Метод notify() вызывается на объекте и будит один из потоков, ожидающих на этом объекте с помощью метода wait(). Какой именно поток будет разбужен, зависит от планировщика потоков.
3. **notifyAll**(): Метод notifyAll() вызывается на объекте и будит все потоки, ожидающие на этом объекте с помощью метода wait(). Все ожидающие потоки становятся активными и конкурируют за доступ к монитору объекта.

Эти методы используются для обеспечения синхронизации между потоками и позволяют им взаимодействовать друг с другом. Они вызываются внутри блока синхронизации, и потоки должны обрабатывать **InterruptedException**, которое может быть выброшено при вызове этих методов.

123). Почему методы wait(), notify() и notifyAll() находятся в классе Object, а не в классе Thread?

Потому что они предназначены для управления блокировками объектов, а не самих потоков.
Эти методы позволяют потокам синхронизировать доступ к общим ресурсам, используя монитор объекта.
Метод **wait**() заставляет текущий поток "уснуть" и освободить монитор объекта, а методы **notify**() и **notifyAll**() используются для оповещения ожидающих потоков о возможности продолжить выполнение.
Потоки работают с общими объектами, и методы wait(), notify() и notifyAll() позволяют им синхронизировать свою работу с этими объектами. Поскольку объекты могут использоваться множеством потоков, важно, чтобы методы для управления блокировками находились в классе Object, чтобы они были доступны для всех объектов и потоков.

124). IllegalMonitorStateException когда будет выброшено?

IllegalMonitorStateException возникает, когда методы wait(), notify() и notifyAll() вызываются в контексте, не синхронизированном с объектом. Эти методы должны всегда вызываться в синхронизированном контексте, используя блокировку объекта, иначе возникает это исключение времени выполнения.

125). Какой из следующих методов снимает блокировку?

METHOD	СНИМАЕТ БЛОКИРОВКУ (ДА ИЛИ НЕТ)
yield()	Нет
sleep()	Нет
join()	Нет
wait()	Да
Notify()	Да
notifyAll()	Да

126). Что такое группа потоков?

Группы потоков (**Thread Groups**) представляют собой способ категоризации и управления потоками в Java. Они обеспечивают возможность организации потоков в иерархические структуры, что позволяет выполнять действия над наборами потоков внутри группы.
Например, с помощью групп потоков можно запускать и останавливать все потоки в группе. Однако класс групп потоков (**ThreadGroup**) редко используется в современном программировании на Java. По умолчанию все потоки, созданные в приложении, принадлежат группе потоков главного потока (main thread). Каждый поток принадлежит определенной группе потоков, и потоки, принадлежащие определенной группе потоков, не могут модифицировать потоки, принадлежащие другой группе потоков.
Использование групп потоков редко применяется в современном программировании на Java, поскольку существуют более гибкие и мощные механизмы управления потоками, такие как **ExecutorService** и фреймворк **java.util.concurrent**. Эти средства предоставляют более высокоуровневые и удобные способы работы с потоками и пулами потоков.

127). Что такое локальные переменные потока?

Thread Local Variables - это переменные, которые связаны с определенным потоком выполнения в многозадачном приложении. Каждый поток имеет свою собственную копию этих переменных, и они не разделяются между потоками. Это означает, что каждый поток может иметь свою собственную уникальную версию переменной с локальным потоком, и изменения, внесенные одним потоком в такую переменную, не повлияют на значения этой переменной в других потоках.
Обычно используются, когда вам необходимо хранить данные, специфичные для каждого потока, и вы хотите избежать конфликтов доступа к этим данным между потоками. Это удобно, например, при создании логгирования для каждого потока или при хранении информации о сессиях для веб-приложений в многопоточной среде.
Для создания переменных с локальным потоком в Java обычно используется класс **ThreadLocal**.
Мы объявляем объект ThreadLocal как статическую переменную в классе.

```
private static final ThreadLocal myThreadLocalInteger = new ThreadLocal ();
```

Каждый раз, когда новый поток обращается к объекту с использованием методов getter или setter, мы обращаемся к копии объекта.

128). Что такое демон-потоки (daemon threads) в Java?

Демон-потоки (daemon threads) - это потоки, которые работают в фоновом режиме и выполняются в пользу других потоков. Они предназначены для выполнения задач, которые не требуют прямого взаимодействия с пользовательским кодом и могут выполняться в фоновом режиме без ожидания завершения всех остальных потоков. Демон-потоки могут быть полезными, например, для автоматической очистки ресурсов или выполнения периодических задач.

По умолчанию все потоки в Java являются недемоническими. Чтобы создать демон-поток, вы можете вызвать метод **setDaemon**(**true**) для потока перед его запуском(перед вызовом метода start()). Если вы попытаетесь установить поток как демонический после того, как он уже начал выполнение, вы получите **IllegalThreadStateException**.

Если родительский поток является демоническим, то все его дочерние потоки также будут демоническими.

Главное отличие демон-потоков от обычных заключается в том, что если все недемонические потоки завершают свою работу, JVM завершает выполнение программы, даже если есть активные демон-потоки. Для этого потоки, которые не являются демоническими, должны завершить свою работу.

Пример использования демон-потока - сборщик мусора, который работает в фоновом режиме и автоматически освобождает неиспользуемую память.

```
Thread myThread = new Thread(myRunnable); // Создание потока
myThread.setDaemon(true); // Установка его как демонического потока
myThread.start(); // Запуск потока
```

129). Можем ли мы сделать поток main() демоном?

Основной поток (main thread) в Java всегда является недемоническим (non-daemon) потоком, и вы не можете изменить его характеристику, чтобы он стал демоническим потоком.

130). Что такое вложенные классы в Java?

Вложенные классы (**nested classes**) - это классы, объявленные внутри другого класса. В Java есть два типа вложенных классов:

1) Статический вложенный класс (**Static Nested Class**)
2) Нестатический вложенный класс (**Non-Static Nested Class**) или внутренний класс (**Inner Class**)

131). Зачем использовать вложенные классы или какова цель вложенного класса в Java?

Вложенные классы в Java имеют несколько целей и могут быть полезными в различных сценариях:

1. **Группировка связанных классов**: Вложенные классы позволяют логически группировать классы, которые имеют тесную связь между собой. Это упрощает организацию кода и делает его более читаемым, так как классы, связанные между собой, могут находиться в одном месте.
2. **Увеличение инкапсуляции**: Вложенные классы могут получать доступ к закрытым (private) членам внешнего класса, что позволяет сохранять инкапсуляцию. Они имеют доступ к private переменным и методам внешнего класса, что делает их удобными для работы с внутренней логикой внешнего класса.
3. **Сокрытие реализации**: Вложенные классы могут использоваться для сокрытия деталей реализации от внешнего мира. Они могут быть закрытыми (private) для внешнего доступа и предоставлять только интерфейс или абстракцию для внешних классов.
4. **Улучшение читаемости и поддержки кода**: Вложенные классы могут улучшить читаемость кода, так как они определены ближе к месту, где они используются. Это делает код более легким для понимания и поддержки.
5. **Минимизация создания лишних классов**: В случаях, когда классы не переиспользуются и создаются только для решения конкретной задачи, можно использовать вложенные классы вместо создания отдельных классов.

Использование вложенных классов в Java зависит от конкретных потребностей и структуры приложения, но они предоставляют удобный механизм для организации кода и улучшения его читаемости и поддержки.

132). Объясните о статических вложенных классах (Static Nested Class) в Java?

Этот тип вложенных классов объявляется с ключевым словом **static** перед его определением. Они связаны с внешним классом, и могут быть использованы независимо от него.

Основные характеристики статических вложенных классов:

1. **Независимость от экземпляра внешнего класса:** Статический вложенный класс не требует создания экземпляра внешнего класса для создания своих собственных экземпляров. Вы можете создавать объекты статического вложенного класса напрямую.
2. **Ограниченный доступ к членам внешнего класса**: Статический вложенный класс имеет доступ только к статическим членам внешнего класса. Он не имеет доступа к нестатическим (instance) переменным и методам внешнего класса.
3. **Удобство упаковки и организации кода**: Статические вложенные классы могут быть полезными для организации связанных классов в одном месте. Они могут использоваться, когда есть логическая связь между классами, но нет необходимости в доступе к экземплярам внешнего класса.

132). продолжение

Пример статического вложенного класса:

```
public class OuterClass {
    private static int outerValue = 10;

    public static class StaticNestedClass {
        public void printOuterValue() {
            System.out.println("OuterValue: " + outerValue);
        }
    }
}
```

В приведенном выше примере StaticNestedClass - это статический вложенный класс, и он имеет доступ к **outerValue** без создания экземпляра OuterClass.
Для создания экземпляра статического вложенного класса:

```
OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();
nestedObj.printOuterValue(); // Выводит "OuterValue: 10"
```

Статические вложенные классы полезны, когда требуется логическая группировка классов и доступ к статическим членам внешнего класса.

133). Что такое внутренние классы или нестатические вложенные классы в Java?

Внутренние классы или нестатические вложенные классы (**inner classes** или **non-static nested classes**) в Java - это классы, объявленные внутри другого класса без использования ключевого слова **static** в определении класса. Эти классы связаны с экземпляром внешнего класса и имеют доступ ко всем его членам, включая нестатические поля и методы.

Существует три типа внутренних классов в Java:

1. (**Local Inner Class**)
2. (**Member Inner Class**)
3. (**Anonymous Inner Class**)

Использование внутренних классов позволяет логически группировать классы и управлять доступом к их членам в зависимости от контекста.

134). Что такое Local Inner Class?

Это классы, определенные внутри метода другого класса. Эти классы существуют исключительно в контексте метода, в котором они определены. Как только метод завершает выполнение, объекты локальных вложенных классов выходят из области видимости и перестают существовать.

Основные характеристики локальных вложенных классов:

1. **Область видимости**: Локальные вложенные классы видны только внутри метода, в котором они определены. Они недоступны извне метода.
2. **Локальные переменные:** Локальные вложенные классы могут обращаться к локальным переменным метода, но только тем переменным, которые объявлены как final или эффективно final. Это связано с тем, что они могут продолжать существовать после завершения метода, поэтому они должны иметь доступ к неизменным переменным.
3. **Уровень инкапсуляции:** Локальные вложенные классы могут иметь доступ ко всем членам внешнего класса, включая его приватные члены. Однако, как правило, их используют для выполнения определенных операций внутри метода.

134). продолжение

Пример локального вложенного класса:

```
public class OuterClass {
    private int outerValue = 10;

    public void createLocalInnerClass() {
        class LocalInner {
            public void printOuterValue() {
                System.out.println("OuterValue: " + outerValue);
            }
        }

        LocalInner localInnerObj = new LocalInner();
        localInnerObj.printOuterValue();
    }
}
```

В приведенном выше примере LocalInner - это локальный вложенный класс, определенный внутри метода createLocalInnerClass(). Его объект создается и используется только внутри этого метода.

Локальные вложенные классы удобны, когда нужно ограничить видимость класса только определенным методом и управлять инкапсуляцией кода внутри этого метода.

135). Что такое Member Inner Class?

Этот класс объявляется внутри внешнего класса и связан с экземпляром внешнего класса. Он может обращаться к нестатическим членам внешнего класса напрямую. Они полезны, когда необходимо создать класс, который имеет доступ к членам внешнего класса и способен работать с их состоянием.
Вот некоторые ключевые особенности:

1. **Модификаторы:** Член внутреннего класса может иметь модификаторы доступа, такие как public, private, protected или по умолчанию, которые определяют его видимость и доступность снаружи внешнего класса.
2. **Абстрактный или финальный**: Члены внутренних классов могут быть объявлены как абстрактные или финальные, как и обычные классы. Абстрактный внутренний класс должен быть подклассом для предоставления конкретной реализации, тогда как финальный внутренний класс не может быть подклассом.
3. **Расширение и реализация**: Члены внутренних классов могут расширять другие классы или реализовывать интерфейсы, что делает их универсальными для различных сценариев проектирования.
4. **Отсутствие статических членов**: Внутренний класс, включая членов внутренних классов, не может объявлять статические поля или методы. Это ограничение связано с тем, что внутренние классы имеют тесные отношения с экземплярами их внешних классов, а статические члены не принадлежат экземплярам.

135). продолжение

Приведем пример:

```
public class OuterClass {
    private int outerValue = 10;

    public class MemberInner {
        public void printOuterValue() {
            System.out.println("OuterValue: " + outerValue);
        }
    }
}
```

В этом примере MemberInner - это член внутреннего класса OuterClass. Он может обращаться к частному члену outerValue из OuterClass.

Члены внутренних классов полезны, когда вам необходимо определить класс, имеющий крепкие отношения с его внешним классом и требующий доступа к его членам. Они обеспечивают лучшую инкапсуляцию по сравнению со статическими вложенными классами и могут использоваться для различных шаблонов проектирования и сценариев.

```
OuterClass outer = new OuterClass();
OuterClass.MemberInner inner = outer.new MemberInner();
inner.printOuterValue();
```

136). Что такое Anonymous Inner Class?

Анонимные внутренние классы в Java - это внутренние классы, которые определены без имени класса. Они объявляются и создаются с использованием ключевого слова new. Основная цель анонимных внутренних классов в Java - предоставить реализацию интерфейса. Анонимные классы используются, когда нам нужен всего один экземпляр класса.

Когда мы компилируем анонимные внутренние классы, компилятор создает два файла:

1. **EnclosingName.class**
2. **EnclosingName$1.class** - для анонимного внутреннего класса.

Анонимный внутренний класс не может иметь конструктор, потому что у класса нет имени.

Анонимный внутренний класс не может определять статические методы, поля или классы.

Нельзя анонимно определить интерфейс.

Анонимный внутренний класс можно создать только один раз.

137). Это допустимо в Java? можем ли мы создать экземпляр интерфейса в Java?

Runnable r = new Runnable() { @Override public void run() { } };

Да, это допустимо в Java. Вы можете создать анонимный внутренний класс, который реализует интерфейс, такой как Runnable. В нашем примере, мы создаем **анонимный** внутренний класс, который реализует интерфейс Runnable. Это позволяет нам предоставить реализацию метода run() интерфейса Runnable внутри анонимного класса.
Таким образом, хотя код выглядит так, как будто мы создаем экземпляр интерфейса, фактически мы создаем анонимный класс, который реализует этот интерфейс, и затем мы создаем экземпляр этого анонимного класса.

138). Как выполнить инкапсуляцию в Java?

Инкапсуляция в Java - это механизм, который обеспечивает защиту данных, скрывая их от прямого доступа и предоставляя доступ к ним только через методы. Для реализации инкапсуляции в Java следуйте этим шагам

1.Сделайте переменные экземпляра (поля) частными (private):

```
private int age;
private String name;
```

2.Определите методы доступа (геттеры и сеттеры) для получения и установки значений полей. Геттеры используются для получения значений полей, а сеттеры - для установки новых значений:

```
public int getAge() {
    return age;
}

public void setAge(int age) {
    if (age >= 0) { // Допустимая проверкаthis.age = age;
    } else {
        System.out.println("Возраст не может быть отрицательным.");
    }
}
```

138). продолжение

```
public String getName() {
    return name;
}

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

Теперь вы можете получать и устанавливать значения полей, используя геттеры и сеттеры вместо прямого доступа к переменным экземпляра:

```
Person person = new Person();
person.setAge(30);
person.setName("John");

int age = person.getAge();
String name = person.getName();
```

Использование инкапсуляции помогает управлять доступом к данным и обеспечивает более безопасное и надежное программирование.

139). Что такое ссылочные переменные в Java?

Переменные, которые используются **для доступа к объектам** в Java, называют ссылочными переменными.

Например:
Employee emp = new Employee();

В примере **emp** - это переменная ссылки.
Переменная ссылки может иметь только один тип.
Переменная ссылки может указывать на любое количество объектов. Но если переменная ссылки объявлена как **final**, она не может указывать на другие объекты.
Переменную ссылки можно объявить как с типом класса, так и с типом интерфейса.
Если переменная ссылки объявлена с типом интерфейса, она указывает **на класс, реализующий** этот интерфейс.

140). Если в классе есть параметризованный конструктор, компилятор создаст конструктор по умолчанию?

Компилятор **не создаст** конструктор по умолчанию, если в классе есть параметризованный конструктор. Например, если у меня есть класс без конструкторов, то компилятор создаст конструктор по умолчанию.

Пример:

```
public class Car {}
```

В классе Car **нет конструкторов**, поэтому компилятор создаст конструктор по умолчанию.

```
public class Car {
    Car(String name) {
    }
}
```

В этом примере компилятор **не создаст** конструктор по умолчанию, потому что **уже** есть один конструктор в классе Car.

141). Можем ли мы иметь имя метода такое же, как имя класса в Java?

Да, в Java можно использовать имя метода, совпадающее с именем класса. Это не вызовет ошибку компиляции, но будет показываться предупреждение о том, что имя метода совпадает с именем класса.

142). Можем ли мы переопределить конструкторы в Java?

В Java можно переопределять **только** методы, конструкторы **нельзя** унаследовать, поэтому нет смысла переопределять конструкторы в Java.

143). Могут ли статические методы получать доступ к переменным экземпляра?

Нет, статические методы **не** могут обращаться к переменным экземпляра (instance variables) в Java. При попытке обращения к переменным экземпляра из статического метода возникает ошибка компиляции с сообщением: "Cannot make a static reference to the non-static field [имя переменной]".

144). Как нам получить доступ к статическим членам в Java?

Для доступа к статическим членам (переменным и методам) в Java используется имя класса, а не ссылка на объект. Вызов статических методов и доступ к статическим переменным осуществляется следующим образом:
Для доступа к статической переменной:

ClassName.staticVariableName

Для вызова статического метода:

ClassName.staticMethodName()

где ClassName - имя класса, содержащего статические члены.

145). Можем ли мы переопределить статические методы в Java?

Статические методы в Java **не могут быть** переопределены. Если в суперклассе и подклассе есть статический метод с одинаковой сигнатурой, это не считается переопределением. Такие методы ведут себя как обычные методы, принадлежащие своим классам, и вызываются с использованием имени класса, а не через механизм полиморфизма, как это происходит с переопределенными методами экземпляра.

146). Разница между объектом и ссылкой в Java?

Разница между объектом и ссылкой заключается в том, что объекты представляют собой **экземпляры класса**, размещенные в памяти **кучи**. Объекты не имеют собственного имени, поэтому для доступа к ним используются **ссылки**. Нет другого способа доступа к объектам, кроме как через ссылки.

Объект не может быть назначен другому объекту, и объект не может быть передан в качестве аргумента методу. Ссылка - это переменная, которая используется для доступа к содержимому объекта. Ссылку можно назначить другой ссылке, **передать в метод**.

147). Объекты или ссылки, какие из них подлежат сборке мусора?

Объекты подвергаются сборке мусора, а не их ссылки.

148). Можем ли мы передавать объекты в качестве аргумента в Java?

Мы можем передавать в методы **только ссылки** на объекты, а не сами объекты. Нельзя передавать объекты как аргументы методов. Максимальный размер данных, которые можно передавать в качестве параметров, - это long или double.

149). Объяснение классов-оболочек (wrapper classes) в Java

Классы-оболочки в Java используются для преобразования примитивных типов данных в объекты. До версии Java 1.5 классы-оболочки использовались для преобразования примитивных типов данных в объекты. Начиная с Java 1.5, появилась новая функция - автоупаковка (**autoboxing**), которая автоматически преобразует примитивные типы данных в объекты. Однако при использовании классов-оболочек программист должен **самостоятельно заботиться** о преобразовании примитивных типов данных в объекты.

Классы-оболочки в Java являются **неизменяемыми**. Как только значение присвоено объекту-оболочке, **его нельзя изменить.**

150). В Java существует восемь классов-оболочек, каждый из которых представляет соответствующий примитивный тип данных.

PRIMTIVE	WRAPPER CLASS
boolean	Boolean
int	Integer
float	Float
char	Character
byte	Byte
long	Long
short	Short