# Вопросы и ответы на собеседование по основам Java

(четвёртая часть)

# 91). Объясните важность планировщика потоков в Java

Планировщик потоков (**thread scheduler**) в Java имеет большое значение для эффективного управления выполнением многопоточных программ. Вот почему планировщик потоков важен:

1. **Определение порядка выполнения**: В многопоточных приложениях может быть несколько потоков, и планировщик потоков определяет порядок их выполнения. Он решает, какой поток должен выполняться в данный момент времени.
2. **Управление приоритетами**: Планировщик потоков также учитывает приоритеты потоков. Высокоприоритетные потоки могут получать больше процессорного времени по сравнению с низкоприоритетными потоками. Это важно для управления ресурсами и отзывчивостью приложения.
3. **Предотвращение гонок и блокировок**: Планировщик потоков может предотвратить состояния гонки (**race conditions**) и блокировки (**deadlocks**), управляя доступом потоков к общим ресурсам и предоставляя им равные возможности выполнения.
4. **Учет времени**: Планировщик потоков также учитывает кванты времени, которые каждый поток получает для выполнения. Если поток завершает свою работу или блокируется, планировщик переключает контекст и выбирает следующий готовый к выполнению поток.
5. **Обеспечение справедливости**: Планировщик потоков также стремится к справедливому распределению процессорного времени между потоками, чтобы каждый поток имел равные шансы выполнения.

Итак, планировщик потоков играет важную роль в эффективном управлении многопоточными приложениями, обеспечивая правильный порядок выполнения, учет приоритетов и предотвращение различных проблем синхронизации и блокировок.

# 92). Объясните жизненный цикл thread

Вот пять состояний жизненного цикла потока (thread) в Java:

1. **New** (Новый): Когда создается экземпляр потока с помощью оператора new, поток находится в состоянии New. Например:

Thread thread = new Thread();

В этом примере поток находится в состоянии New. Он создан, но еще не активирован. Чтобы активировать поток, необходимо вызвать метод start().

2. **Runnable** (Готовый к выполнению): Поток находится в состоянии Runnable, когда ему разрешено выполняться. Это может произойти двумя способами:
   - Когда вызывается метод start(), и поток переходит из состояния New в состояние Runnable.
   - Поток может вернуться в состояние Runnable после блокировки, ожидания или простоя.
3. **Running** (Выполняющийся): Поток находится в состоянии Running, когда планировщик потоков (thread scheduler) назначает ему процессорное время, и он активно выполняется.
4. **Waiting/Blocking/Sleeping** (Ожидание/Блокировка/Сон): Поток может находиться в этом состоянии временно, когда:
   - Поток ожидает получения блокировки объекта.
   - Поток ожидает завершения другого потока.
   - Поток ожидает уведомления от другого потока.
5. **Dead** (Завершенный): Поток находится в состоянии Dead, когда выполнение его метода run() завершено. Он автоматически умирает после завершения выполнения метода run(), и объект потока подлежит сборке мусора.

Эти состояния представляют жизненный цикл потока, начиная с создания и активации (или перехода в состояние готовности) и заканчивая завершением выполнения.

# 93). Можем ли мы перезапустить мертвый поток (dead thread) в Java?

Нет, вы не можете перезапустить завершенный поток в Java. Как только поток завершил выполнение и достиг состояния "мертв", его нельзя перезапустить с использованием метода start(). Если вы попытаетесь перезапустить завершенный поток, вы получите исключение времени выполнения, а именно **IllegalThreadStateException**.

Чтобы выполнить задачу снова, вам следует создать новый экземпляр потока и, если необходимо, запустить этот новый поток. Потоки в Java не предназначены для повторного использования после завершения своего выполнения.

# 94). Может ли один поток блокировать другой поток?

Нет, один поток не может заблокировать другой поток в Java. Он может заблокировать текущий поток, который выполняется в данный момент.

# 95). Можем ли мы перезапустить поток, уже запущенный в Java?

Нет, в Java нельзя перезапустить уже запущенный поток. Если мы вызываем метод start() второй раз после того, как поток уже был запущен, это вызовет исключение времени выполнения (**IllegalThreadStateException**). Запущенный поток нельзя перезапустить.

# 96). Что произойдет, если мы не переопределим метод run?

Если мы не переопределим метод run(), то будет выполнена реализация метода run() по умолчанию из класса Thread, и, следовательно, поток никогда **не будет** находиться в состоянии выполнения (**runnable state**).

# 97). Можем ли мы перегрузить (overload) метод run() в Java?

Да, мы можем перегрузить метод run() в Java. Однако метод start() класса Thread всегда вызывает метод run() без аргументов. Перегруженный метод run() не будет вызван автоматически методом start(), и его нужно будет вызывать явным образом.

# 98). Что такое блокировка или цель блокировок в Java?

Блокировка, также называемая **монитором**, используется для предотвращения доступа к общему ресурсу несколькими потоками. Когда поток хочет получить доступ к общему ресурсу, он должен сначала получить блокировку. Если блокировка уже была получена другим потоком, поток не может получить доступ к этому общему ресурсу и должен ожидать, пока другой поток не освободит блокировку. Для блокировки объекта в Java используется **синхронизация**. Блокировка **защищает** участок кода, позволяя выполняться только одному потоку за раз.

# 99). Сколькими способами мы можем выполнить синхронизацию в Java?

В Java существует два способа синхронизации:

1. Синхронизированные методы
2. Синхронизированные блоки

Для выполнения синхронизации в Java используется ключевое слово **synchronized**.

# 100). Что такое синхронизированные методы?

Синхронизированные методы - это методы, для которых объявлен модификатор **synchronized**. Если метод объекта объявлен как синхронизированный, это означает, что только один поток может выполнить этот метод в данном объекте в любой момент времени. Другие потоки должны ожидать, пока текущий поток не завершит выполнение синхронизированного метода и **не освободит блокировку**.
Пример синхронизированного метода:

```
public synchronized void synchronizedMethod() {
    // Код, который нужно синхронизировать
}
```

В данном примере только один поток может выполнить метод synchronizedMethod() объекта в определенный момент времени.

# 101). Когда мы используем синхронизированные методы в Java?

Синхронизированные методы используются в Java в следующих случаях:

1. **Когда несколько потоков имеют доступ к одному и тому же объекту и могут изменять его состояние**. Синхронизация метода гарантирует, что только один поток будет иметь доступ к методу в данный момент времени, предотвращая конфликты и гонки данных.
2. **Когда необходимо обеспечить атомарность операций**. Синхронизированный метод гарантирует, что все инструкции внутри метода будут выполнены целиком до того, как другой поток сможет выполнить этот метод.
3. **Когда требуется защитить критические разделяемые ресурсы**. Синхронизация методов может быть использована для защиты доступа к общим ресурсам, таким как разделяемые переменные или структуры данных.
4. **Когда нужно избежать гонок данных**. Синхронизированные методы помогают предотвратить гонки данных, которые могут возникнуть при одновременном доступе нескольких потоков к общим ресурсам.

Важно помнить, что синхронизированные методы могут вызывать некоторое снижение производительности из-за ожидания блокировки, поэтому их следует использовать осторожно и только там, где это действительно необходимо.

# 102). Когда один поток выполняет синхронизированный метод, другие потоки могут ? одновременно выполнять другие синхронизированные методы этого объекта

Когда один поток выполняет синхронизированный метод, другие потоки не могут одновременно выполнять другие синхронизированные методы этого объекта.
Синхронизация **на объекте** означает, что только один поток может войти **в любой из синхронизированных методов этого объекта** в данный момент времени.
Это гарантируется блокировкой, которая устанавливается при входе в синхронизированный метод и снимается при его завершении. Пока блокировка установлена одним потоком, другие потоки будут ожидать ее снятия, прежде чем они смогут войти в синхронизированный метод этого объекта.
Таким образом, синхронизированные методы обеспечивают последовательное выполнение только для этого объекта, предотвращая одновременное выполнение других синхронизированных методов объекта.

# 103). Когда поток выполняет синхронизированный метод объекта, он может? получить доступ к другим синхронизированным методам этого объекта

Да, когда поток выполняет синхронизированный метод объекта, он также может получить доступ к другим синхронизированным методам **этого объекта**.
Когда поток вызывает синхронизированный метод, он **автоматически захватывает блокировку объекта,** на котором выполняется метод. Если нет других потоков, которые уже захватили блокировку на этом объекте, то поток получает блокировку и выполняет синхронизированный метод. После завершения выполнения метода блокировка автоматически освобождается.
Это означает, что поток может последовательно вызывать разные синхронизированные методы **одного и того же объекта**, при условии, что он каждый раз успешно захватывает блокировку объекта перед вызовом метода.

# 104). Что такое синхронизированные блоки в Java?

Синхронизированные блоки в Java позволяют синхронизировать только определенные **участки кода**, а не весь метод, с использованием ключевого слова **synchronized**.
Сигнатура синхронизированного блока выглядит следующим образом:

```
synchronized (ссылка на объект) {
    // код, который требуется синхронизировать
}
```

Когда поток входит в синхронизированный блок, он захватывает блокировку для указанного объекта, выполняет код внутри блока и затем освобождает блокировку. Если другой поток пытается войти в синхронизированный блок с тем же объектом, он будет ожидать, пока блокировка не освободится. Это позволяет контролировать доступ к критическим участкам кода, где множество потоков может конфликтовать за общие ресурсы.

# 105). Когда мы используем синхронизированные блоки? и какие есть преимущества использования синхронизированных блоков?

Мы используем синхронизированные блоки, когда нам нужно синхронизировать только небольшие участки кода, и есть несколько преимуществ в использовании синхронизированных блоков:

1. **Уменьшение ожидания потоков**: Синхронизированные блоки позволяют потокам быстрее получать доступ к критическим участкам кода, чем синхронизированные методы. Это снижает вероятность блокировки и улучшает производительность системы.
2. **Более гибкое управление блокировками**: Вы можете синхронизировать только те участки кода, которые действительно требуют синхронизации, а не весь метод. Это позволяет более точно контролировать доступ к общим ресурсам.
3. **Повышение производительности**: Использование синхронизированных блоков может уменьшить накладные расходы на синхронизацию по сравнению с синхронизированными методами, что может привести к улучшению производительности приложения.

Таким образом, синхронизированные блоки предоставляют более гибкий и эффективный способ управления многозадачностью в Java.

# 106). Что такое блокировка на уровне класса?

Захватывание блокировки на экземпляре класса, а не на объекте класса, называется блокировкой на уровне класса. Разница между блокировкой на уровне класса и блокировкой на уровне объекта заключается в том, что при блокировке на уровне класса блокировка захватывается на экземпляре класса **.class**, а при блокировке на уровне объекта блокировка захватывается на **объекте класса**.

# 107). Можем ли мы синхронизировать статические методы в Java?

Да, в Java мы можем синхронизировать статические методы с помощью ключевого слова synchronized. Когда статический метод объявляется с ключевым словом synchronized, он захватывает блокировку на уровне класса. Это означает, что только один поток может выполнить статический синхронизированный метод данного класса в любой момент времени. Захваченная блокировка приводит к тому, что другие потоки должны ждать, пока блокировка не будет освобождена.

Пример синхронизации статического метода:

```
public class MyClass {
    public static synchronized void staticMethod() {
        // Синхронизированный статический метод
    }

    public static void anotherStaticMethod() {
        synchronized (MyClass.class) {
            // Синхронизированный блок на уровне класса
        }
    }
}
```

Оба примера выше продемонстрировали синхронизацию статических методов на уровне класса. Важно помнить, что синхронизация на уровне класса не влияет на другие типы методов, такие как обычные статические методы, обычные методы экземпляра и синхронизированные методы экземпляра других объектов. Эти методы могут выполняться параллельно с синхронизированными статическими методами, пока они сами не используют синхронизацию.

# 108). Можем ли мы использовать синхронизированный блок для примитивных типов данных напрямую в Java?

Нет, мы не можем использовать синхронизированный блок для примитивных типов данных напрямую в Java. Синхронизированные блоки могут быть применены только к объектам. Если мы попробуем использовать синхронизированный блок для примитивов, мы получим ошибку времени компиляции.

Чтобы синхронизировать операции с примитивами, вы можете использовать объекты-оболочки (wrapper objects) для этих примитивов, такие как Integer, Double, Boolean и другие, и синхронизироваться на них.

# 109). Каковы приоритеты потоков и важность приоритетов потоков в Java?

Когда в ожидании находится несколько потоков, приоритеты потоков определяют, какой поток запустить.
В языке JAVA каждый поток имеет приоритет. Поток наследует приоритет своего родительского потока.По умолчанию поток имеет обычный приоритет 5.
Планировщик потоков использует приоритеты потоков, чтобы решить, когда какому потоку разрешить работать.
Планировщик потоков сначала запускает потоки с более высоким приоритетом.
Приоритеты потоков в Java позволяют вам влиять на порядок выполнения потоков. Несмотря на то, что это может быть полезным инструментом для управления поведением потоков, их следует использовать осторожно и обдуманно, чтобы избежать потенциальных проблем.

# 110). Разница между типами приоритета потоков

В Java потоки имеют приоритеты в диапазоне от 1 до 10, где 1 - это наименьший приоритет, 10 - наивысший приоритет. По умолчанию приоритет установлен на средний уровень (Thread.NORM_PRIORITY), который равен 5. Java предоставляет следующие константы для определения приоритетов потоков:

1. Thread.MIN_PRIORITY (минимальный приоритет) = 1
2. Thread.NORM_PRIORITY (нормальный приоритет) = 5
3. Thread.MAX_PRIORITY (максимальный приоритет) = 10

Использование приоритетов позволяет определить, какой поток следует выполнить в первую очередь в случае конкуренции за ресурсы процессора. Однако стоит помнить, что приоритеты могут варьироваться в зависимости от операционной системы и не всегда гарантируют точный порядок выполнения.

# III). Как изменить приоритет потока или как установить приоритет потока?

Для изменения приоритета потока в Java вы можете использовать метод **setPriority(int value)** класса Thread. Этот метод позволяет установить приоритет потока. Вот его сигнатура:

```
final void setPriority(int value);
```

Значение приоритета должно быть целым числом в диапазоне от 1 до 10, где 1 - это наименьший приоритет, а 10 - наивысший приоритет.
Пример установки приоритета потока:

```
Thread myThread = new Thread();
myThread.setPriority(8); // Установка приоритета наивысшего уровня (8)
```

Чтобы получить текущий приоритет потока, вы можете использовать метод

**getPriority()**:

```
int priority = myThread.getPriority(); // Получение текущего приоритета
```

# 112). Если два потока имеют одинаковый приоритет, какой поток будет выполнен первым?

Когда у двух потоков одинаковый приоритет, не гарантируется, какой из них будет выполнен первым. Это зависит от планировщика потоков, и планировщик может действовать по-разному:

1. Он может выбрать любой поток из пула и запустить его до его завершения.
2. Он может предоставить равные возможности для всех потоков с помощью разделения времени (time slicing).

Точный порядок выполнения зависит от реализации планировщика потоков в конкретной системе и может быть разным на разных платформах.

# 113). Какие методы используются для предотвращения выполнения потока?

Есть три метода в классе Thread, которые могут предотвратить выполнение потока:

1. **yield()**
2. **join()**
3. **sleep()**

# 114). Объясните работу метода yield() в классе Thread?

Метод yield() в классе Thread используется для попытки перевода текущего выполняющегося потока из состояния "выполнение" в состояние "готовности". Это означает, что текущий поток добровольно уступает свое место другим потокам того же приоритета, которые могут быть готовы к выполнению. Однако нет гарантии, что это приведет к переключению контекста и выполнению другого потока, так как это зависит от планировщика потоков в JVM.
Следует помнить, что yield() - это всего лишь предложение планировщику переключиться на другой поток, и планировщик может проигнорировать это предложение. yield() может быть полезен в ситуациях, когда у вас есть несколько потоков с одинаковым приоритетом, и вы хотите дать им равные шансы на выполнение.

## 115). У потока, вызвавшего yield(), есть возможность получить ещё один шанс на выполнение?

Да, у потока, вызвавшего yield(), есть возможность получить ещё один шанс на выполнение, но это не гарантировано. Когда вы вызываете yield(), вы фактически сообщаете планировщику потоков, что текущий поток готов уступить свой интервал выполнения и позволить другим потокам с равным приоритетом выполняться. Однако, будет ли уступивший поток запланирован для выполнения сразу же или нет, зависит от алгоритма планировщика потоков и общей загрузки системы. Планировщик может решить дать уступившему потоку ещё один шанс или позволить другим потокам продолжить выполнение.

# 116). Объясните важность метода join() в классе Thread?

Метод join() в классе Thread имеет большое значение, потому что он позволяет одному потоку ожидать завершения выполнения другого потока. Предположим, у нас есть два потока, t1 и t2. Если выполняющийся поток t1 вызывает метод join() на потоке t2, то поток t1 будет ожидать, пока поток t2 не завершит выполнение. Как только поток t2 завершит выполнение, поток t1 продолжит свою работу.
Метод join() может вызывать исключение **InterruptedException**, поэтому при его использовании необходимо обрабатывать это исключение с помощью блока try-catch или объявления его в сигнатуре метода с использованием ключевого слова throws. Это важно для обеспечения корректной обработки потоковых исключений.
Метод join() имеет несколько перегрузок, которые позволяют указывать время ожидания завершения потока.

Signature :

```
public final void join() throws InterruptedException {
}
public final synchronized void join(long millis) throws InterruptedException {
}
public final synchronized void join(long millis, int nanos) throws InterruptedException {
}
```

# 117). Объясните назначение метода sleep() в Java?

Метод **sleep**() в Java используется для приостановки выполнения текущего потока на определенное количество времени. Этот метод позволяет временно приостановить выполнение потока и передать управление другим потокам. Основное назначение метода sleep() - управление временными задержками в потоках. Например, он может использоваться для создания пауз между действиями, реализации таймеров и т. д. Важно отметить, что время ожидания, указанное в методе sleep(), представляет минимальное время ожидания, но не обязательно точное время, так как точность может зависеть от реализации JVM и операционной системы.
Метод sleep() также может бросать исключение **InterruptedException**, поэтому при его использовании необходимо обрабатывать это исключение с помощью блока try-catch или объявлять его в сигнатуре метода с использованием ключевого слова throws. Это важно для обработки ситуаций, когда поток может быть прерван во время ожидания сна.

Signature :
public static native void sleep(long millis) throws InterruptedException { }
public static void sleep(long millis, int nanos) throws InterruptedException { }

## 118). Предположим, что поток заблокирован, вызовом метода sleep(). В этом потоке блокировка будет освобождена?

При вызове метода sleep() на потоке, который удерживает блокировку (lock), блокировка не будет освобождена. Поток продолжит удерживать блокировку, даже если он "спит" (ожидает) в течение определенного времени. Метод sleep() влияет только на состояние выполнения потока, но не влияет на блокировку или монитор.
Чтобы освободить блокировку, удерживаемую потоком, необходимо использовать оператор **wait**(), который приостанавливает выполнение потока и освобождает блокировку, пока другой поток не вызовет метод **notify**() или **notifyAll**() для возобновления выполнения потока.

# 119). Может ли метод sleep() перевести другой поток в спящий режим?

Метод sleep() вызывает приостановку выполнения **текущего потока**, а не другого потока. Он не может вызвать приостановку выполнения другого потока напрямую.

# 120). Объясните работу метода interrupt() в классе Thread?

Метод **interrupt**() класса Thread используется для прерывания текущего или другого потока. Это не означает, что текущий поток немедленно останавливается; это вежливый способ сказать или запросить другому потоку продолжить свою текущую работу. Поэтому мы можем не увидеть эффект вызова interrupt() немедленно.
Изначально у потока есть логическое свойство (interrupted status - статус прерывания), установленное в **false**. Поэтому, когда мы вызываем метод interrupt(), статус устанавливается в **true**. Это приводит к тому, что текущий поток продолжает свою работу и не останавливается немедленно.
Если поток находится в состоянии сна или ожидания (то есть выполнил методы wait() или sleep()), и его прервать, он останавливается и генерирует исключение **InterruptedException**. Поэтому нам нужно обрабатывать исключение InterruptedException с помощью throws или блока try/catch.