**1. Что такое дедлок?**  
Дедлок – это ситуация, когда два и более нитей заблокированы, ждущие друг друга. Дедлоком также называется взаимная блокировка.  
Взаимная блокировка – это ситуация в которой, два или более процесса занимая некоторые ресурсы, пытаются заполучить некоторые другие ресурсы, занятые другими процессами и ни один из процессов не может занять необходимый им ресурс, и соответственно освободить занимаемый.   
Бывают взаимные блокировки порядка синхронизации (решаются назначением порядка);  
Взаимная блокировка между объектами (различные объекты пытаются получить доступ к одним и тем же синхронизированным блокам);  
Ресурсная взаимоблокировка (при попытке получить доступ к некоторым ресурсам, которые может использовать одновременно только один поток).  
  
**2 Какие вы знаете стратегии, предотвращающие появление дедлоков?**  
Безусловно, если код написан без каких-либо ошибок, то взаимных блокировок в нем не будет. Но кто может поручиться, что его код написан без ошибок? Безусловно, тестирование помогает выявить значительную часть ошибок, но как мы уже видели ранее, ошибки в многопоточном коде нелегко диагностировать и даже после тестирования нельзя быть уверенным в отсутствии ситуаций взаимных блокировок. Можем ли мы как-то перестраховаться от блокировок? Ответ – да. Подобные техники применяются в движках баз данных, которым нередко необходимо восстанавливаться после взаимных блокировок (связанных с механизмом транзакций в БД). Интерфейс Lock и его реализации доступные в пакете java.util.concurrent.locks позволяют попытаться занять монитор, связанный с экземпляром данного класса методом tryLock (возвращает true, если удалось занять монитор).  
Также есть стратегия применения открытых вызовов, то есть вызывать методы других объектов вне синхронизированного блока.  
Ссылка на статью [www.developersonthe.net/ru/posts/post\_id/34-Vzaimnaja-blokirovkadeadlock-v-Java-i-metody-borby-s-nej/](http://www.developersonthe.net/ru/posts/post_id/34-Vzaimnaja-blokirovkadeadlock-v-Java-i-metody-borby-s-nej/)  
  
**3 Могут ли возникнуть дедлоки при использовании методов wait-notify?**  
Ответить на этот вопрос сложно лично мне, но прочитав в интернете разные дискуссии на эту тему, можно сказать следующее:  
Дедлоков можно избежать за счет разумного использования synchronized, volatile, монитора (wait(), notify(), notifyAll()), а если копать глубже, то используя классы java.utils.concurrent: вместо обычных коллекций — многопоточные варианты (ConcurrentHashMap, например); если нужен более сложный способ синхронизации потоков — различные CyclicBarrier, CountDownLatch.  
Если грамотно использовать wait – notify, то дедлоки возникнуть не должны.)))  
Вот ссылка: [www.quizful.net/interview/java/Deadlock](http://www.quizful.net/interview/java/Deadlock)  
  
**4 Что чаще используется: notify или notifyAll?**  
The java.lang.Object.notify() wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.  
This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:  
• By executing a synchronized instance method of that object.  
• By executing the body of a synchronized statement that synchronizes on the object.  
• For objects of type Class, by executing a synchronized static method of that class.  
Only one thread at a time can own an object's monitor.  
The java.lang.Object.notifyAll() wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.  
The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.  
This method should only be called by a thread that is the owner of this object's monitor.  
Это отрывки из документации. Вопрос – то по большей части риторический, смотря какое приложение, в зависимости от ситуации))) Я даже не знаю, как бы я ответил. Если у кого-то есть какие-то догадки, то прошу в комментариях оставить, буду очень рад почитать.  
  
**5 Метод wait рекомендуется использовать с конструкциями if или while?**  
Здесь отвечу просто цитатой из сайта: [www.skipy.ru/technics/synchronization.html](http://www.skipy.ru/technics/synchronization.html)  
По поводу вызова метода wait. Это уже из разряда чистой техники. Рекомендуется вызывать wait изнутри цикла while. Т.е., писать не

if (some condition){  
    obj.wait()  
}

..., а

while (some condition){  
    obj.wait()  
}

Зачем это надо. Дело в том, что notify может вызвать кто угодно. Просто по ошибке, от которой никто не застрахован. В том случае из опыта, о котором я рассказывал выше, мы взялись за переделку именно для того, чтобы избежать такой возможности. Просто спрятали объект, на котором происходит синхронизация. И доступ к нему имел только наш код. Это хорошая практика, но не всегда возможно, к сожалению. Так вот, если поток ждет выполнения некоторого условия – вариант с while надежнее. Если поток пустили по ошибке – он опять проверит условие и, если надо, будет ждать дальше.  
Кроме того, не исключена возможность и простого выхода из ожидания без вызова notify. Я честно признаюсь, что не видел этого в спецификации виртуальной машины, хотя специально искал. Но некоторые «гуру» утверждают, что VM может выйти из состояния ожидания самопроизвольно. И более того, периодически это наблюдается. Если кто-нибудь даст ссылку на соответствующую спецификацию – буду благодарен!  
  
**6 Что происходит после вызова метода notifyAll?**  
The java.lang.Object.notifyAll() wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.  
Пробуждает все нити, которые ждали на этом мониторе.  
  
**7 Какие выгоды получает объект, если он immutable?**  
Нашел комментарий на: [ru.stackoverflow.com/questions/432545/](http://ru.stackoverflow.com/questions/432545/)  
Immutable объект — это объект, состояние которого после создания невозможно изменить. В случае Java это значит что все поля экземпляра у класс отмечены как final и являются примитивами или тоже immutable типами.  
Пример:

public class ImmutablePoint {  
    private final int x;  
    private final int y;  
    private final String description;  
  
    public ImmutablePoint(int x, int y, String description) {  
        this.x = x;  
        this.y = y;  
        this.description = description;  
    }  
}

После создания экземпляра ImmutablePoint его модификация невозможна.  
Простейший пример immutable класса из JDK это String. Любые методы, которые вы вызовите на строке (например description.toLowerCase()) вернут новую строку, а не модифицируют исходную.  
Пример mutable класс из JDK — Date. Например myDate.setHours(x) модифицирует сам экземпляр myDate!  
  
Есть разница между immutable-объектом (то есть, неизменяемым), и final-ссылкой.  
Ключевое слово final для объектных типов гарантирует неизменяемость лишь ссылки, но не самого объекта. Например, если у вас есть final-ссылка на ArrayList, вы тем не менее можете добавлять в него новые элементы или изменять существующие.  
В случае же immutable-объекта объект после окончания конструктора не изменяется вообще. Одного лишь модификатора final для этого недостаточно, необходимо, чтобы все подобъекты были тоже неизменяемыми. Вы в принципе можете держать внутри ссылку на изменяемый объект, но обращаться с ним так, чтобы он не менялся.  
Использование неизменяемых объектов даёт много выгод. Например, о таком объекте намного легче судить в ситуации, когда во многих частях программы есть ссылка на него (для изменяемого объекта, любая часть программы может вызвать мутирующую функцию в практически любой момент времени и из любого потока).  
Но то, что для нас важно в контексте вопроса — неизменяемые объекты не требуют синхронизации при многопоточном доступе. Вот собственно и вся рекомендация: используйте неизменяемые объекты, и вам не придётся думать о том, что нужно, а что не нужно синхронизировать. Единственная возможная проблема — если вы внутри ещё не отработавшего конструктора публикуете ссылку на объект, через которую к нему может получить доступ кто-нибудь ещё, и увидеть объект в изменяющемся состоянии! (Это бывает не так уж и редко. Например, иногда программист хочет добавить объект в конструкторе в коллекцию всех объектов данного типа.)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Следует различать действительно неизменяемые объекты, и объекты, имеющие лишь интерфейс «только для чтения». При чтении объект тем не менее может менять свою внутреннюю структуру (например, кэшировать самый свежий запрос данных). Такие объекты не являются в строгом смысле неизменяемыми, и не могут быть использованы из разных потоков без предосторожностей. (Поэтому, если ваш объект включает другие объекты, убедитесь, что документация гарантирует их неизменяемость!)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Обратите внимание, что для полей неизменяемого объекта вы практически обязаны использовать final! Дело в так называемой безопасной публикации. Смотрите. Инструкции в Java-программе могут быть переставлены как оптимизатором, так и процессором (у Java достаточно слабая модель памяти). Поэтому, если не предпринимать специальных действий, окончание работы конструктора и присвоение значений полям может быть переставлено (но невидимо в рамках текущего потока)! Использование final гарантирует, что такого не произойдёт.  
  
В случае многопоточного программирования преимущества immutable классов очевидны: после создания объекты можно передавать другим потокам и они всегда будут в актуальном состоянии. Т.е. вам не надо проверять не устарело ли состояние вашего экземпляра и не модифицировал ли его другой поток пока вы с ним работаете. Например, у вас есть метод bill(Date endDate), в нём вы наивно проверяете соответствие endDate каким-то предварительным условиям и начинаете с ней работать. В этот момент другой поток может изменить endDate, например установит её глубоко в прошлое. Последствия могут быть самыми удивительными.  
  
**8 Что такое «thread-safe»?**  
Опять же:  
[stackoverflow.com/questions/6324085/](http://stackoverflow.com/questions/6324085/)  
Thread safe means that a method or class instance can be used by multiple threads at the same time without any problems occuring.  
Состояние потоко-безопасности подразумевает, что метод или класс может быть использован множеством нитей без проблем столкновения, то есть дедлоков.  
Consider the following method:

    private int myInt = 0;  
public int AddOne()  
{  
    int tmp = myInt;  
    tmp = tmp + 1;  
    myInt = tmp;  
    return tmp;  
}

Now thread A and thread B both would like to execute AddOne(). but A starts first and reads the value of myInt (0) into tmp. Now for some reason the scheduler decides to halt thread A and defer execution to thread B. Thread B now also reads the value of myInt (still 0) into it's own variable tmp. Thread B finishes the entire method, so in the end myInt = 1. And 1 is returned. Now it's Thread A's turn again. Thread A continues. And adds 1 to tmp (tmp was 0 for thread A). And then saves this value in myInt. myInt is again 1.  
Здесь и нить А и нить B хотят выполнить AddOne (). но А начинается первой и считывает значение myInt (0) в TMP. Теперь по некоторым причинам планировщик решает остановить поток А и отложить выполнение нити B. Поток В настоящее время также считывает значение myInt (0) в его собственной переменной TMP. Нить B завершает весь метод так, что в конце концов myInt = 1. И 1 возвращается. Поток А продолжается. И добавляет 1 к TMP (TMP 0 для нити A). А затем сохраняет это значение в myInt. myInt снова 1.  
So in this case the method AddOne was called two times, but because the method was not implemented in a thread safe way the value of myInt is not 2, as expected, but 1 because the second thread read the variable myInt before the first thread finished updating it.  
Так что в этом случае метод AddOne был вызван два раза, но так как этот метод не был реализован в потоке безопасным способом величина myInt не 2, как ожидалось, а 1, потому что второй поток чтения переменной myInt закончился перед первой нитью до его обновления.  
Creating thread safe methods is very hard in non trivial cases. And there are quite a few techniques. In Java you can mark a method as synchronized, this means that only one thread can execute that method at a given time. The other threads wait in line. This makes a method thread safe, but if there is a lot of work to be done in a method, then this wastes a lot of time. Another technique is to 'mark only a small part of a method as synchronized' by creating a lock or semaphore, and locking this small part (usually called the critical section). There are even some methods that are implemented as lockless thread safe, which means that they are built in such a way that multiple threads can race through them at the same time without ever causing problems, this can be the case when a method only executes one atomic call. Atomic calls are calls that can't be interrupted and can only be done by one thread at a time.  
  
Создание потоко — безопасных методов очень трудно. В Java вы можете пометить метод как синхронизированный, это будет означать, что только один поток может выполнить этот метод в данный момент времени. Другие нити, будут ждать в очереди. Это делает метод потоко-безопасным, но если много работы предстоит сделать в методе, то на это будет уходить много времени. Другой метод заключается в разметке лишь малой части метода, как синхронизированного 'путем создания локов(locks) или семафоров, и запирании этой небольшой части (обычно называемый критический раздел (critical section)). Есть даже некоторые методы, которые реализуются как беззамочные потокобезопасные (lockless thread safe), это означает, что они построены таким образом, что несколько потоков могут проходить через них в одно время и никогда не вызывают проблем, это может быть в случае, когда метод выполняет только один атомарный вызов. Атомарные вызовы это вызовы, которые не могут быть прерваны, и может быть реализованы только одним потоком.  
  
**9 Что такое «happens-before»?**  
Есть статья на википедии, она не конкретно про «happens-before», но все – таки.  
А так-то:   
«Выполняется прежде» (англ. happens before) — отношение строгого частичного порядка (арефлексивное, антисимметричное, транзитивное), введённое между атомарными командами (++ и — не атомарны!), придуманное Лесли Лэмпортом и не означающее «физически прежде». Оно значит, что вторая команда будет «в курсе» изменений, проведённых первой.  
  
[ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C\_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8\_Java](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8_Java)  
В частности, одно выполняется прежде другого для таких операций (список не исчерпывающий):  
• Синхронизация и мониторы:  
• Захват монитора (начало synchronized, метод lock) и всё, что после него в том же потоке.  
• Возврат монитора (конец synchronized, метод unlock) и всё, что перед ним в том же потоке.  
• Таким образом, оптимизатор может заносить строки в синхроблок, но не наружу.  
• Возврат монитора и последующий захват другим потоком.  
• Запись и чтение:  
• Любые зависимости по данным (то есть запись в любую переменную и последующее чтение её же) в одном потоке.  
• Всё, что в том же потоке перед записью в volatile-переменную, и сама запись.  
• volatile-чтение и всё, что после него в том же потоке.  
• Запись в volatile-переменную и последующее считывание её же.[4][2] Таким образом, volatile-запись делает с памятью то же, что возврат монитора, а чтение — то же, что захват.[5] А значит: если один поток записал в volatile-переменную, а второй обнаружил это, всё, что предшествует записи, выполняется раньше всего, что идёт после чтения; см. иллюстрацию.  
• Для объектных переменных (например, volatile List x;) столь сильные гарантии выполняются для ссылки на объект, но не для его содержимого.  
• Обслуживание объекта:  
• Статическая инициализация и любые действия с любыми экземплярами объектов.  
• Запись в final-поля в конструкторе[6] и всё, что после конструктора. Как исключение из всеобщей транзитивности, это соотношение happens-before не соединяется транзитивно с другими правилами и поэтому может вызвать межпоточную гонку.[7]  
• Любая работа с объектом и finalize().  
• Обслуживание потока:  
• Запуск потока и любой код в потоке.  
• Зануление переменных, относящихся к потоку, и любой код в потоке.  
• Код в потоке и join(); код в потоке и isAlive() == false.  
• interrupt() потока и обнаружение факта останова.  
  
**10 Что такое JMM?**  
Java Memory Model  
Вот ссылка:  
[docs.oracle.com/javase/specs/jls/se7/html/jls-17.html](https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html)  
И вот выдержка из нее:  
A memory model describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. The Java programming language memory model works by examining each read in an execution trace and checking that the write observed by that read is valid according to certain rules.  
Я понял на своем уровне что это набор правил:  
  
Правило № 1: однопоточные программы исполняются псевдопоследовательно. Это значит: в реальности процессор может выполнять несколько операций за такт, заодно изменив их порядок, однако все зависимости по данным остаются, так что поведение не отличается от последовательного.  
Правило № 2: нет невесть откуда взявшихся значений. Чтение любой переменной (кроме не-volatile long и double, для которых это правило может не выполняться) выдаст либо значение по умолчанию (ноль), либо что-то, записанное туда другой командой.  
И правило № 3: остальные события выполняются по порядку, если связаны отношением строгого частичного порядка «выполняется прежде» (англ. happens before).  
  
**11 Какое исключение вылетит, если вызвать wait не в блоке synchronized?**  
Вот ссылка:  
[jsehelper.blogspot.ru/2016/01/multithreading-2.html](https://jsehelper.blogspot.ru/2016/01/multithreading-2.html)  
Основная причина вызова wait и notify из статического блока или метода в том, что Java API обязательно требует этого. Если вы вызовете их не из синхронизированного блока, ваш код выбросит IllegalMonitorStateException. Более хитрая причина в том, чтобы избежать состояния гонки между вызовами wait и notify.