1. Как пользоваться интерфейсом Comparable?

В интерфейсе Comparable объявлен всего один метод *compareTo(Object obj)*, предназначенный для реализации упорядочивания объектов класса. Его удобно использовать при сортировке упорядоченных списков или массивов объектов.  
  
Данный метод сравнивает вызываемый объект с obj. В отличие от метода equals, который возвращает true или false, compareTo возвращает:

* 0, если значения равны;
* Отрицательное значение, если вызываемый объект меньше параметра;
* Положительное значение, если вызываемый объект больше параметра.

Прежде всего он удобен для сортировки упорядоченных списков (java.util.List) и массивов объектов. Если список/массив содержит элементы, реализующие этот интерфейс, то они могут быть отсортированы автоматически методами java.util.Collections.sort(List)/Arrays.sort(Object[]).  
  
С интерфейсом Comparable связано понятие натурального упорядочивания, потому как он устанавливает натуральный порядок следования экземпляров любого класса, реализующего этот интерфейс. Иначе говоря, порядок (x, y) соответствует выполнению условия x.compareTo(y) <= 0.  
  
Правила реализации Comparable, а вернее, его метода compareTo(Object) следующие (x и y – экземпляры класса, реализующего Comparable):

* x.compareTo(y) возвращает -1 или 1, если x должен находиться, соответственно, раньше или позже y. Если метод возвращает 0, то порядки (x, y) и (y, x) эквивалентны.
* Если sign(a) – функция, возвращающая -1,0,1 для а, соответственно, меньше 0, равного 0 и больше 0, то должно выполняться равенство sign(x.compareTo(y))==-sign(y.compareTo(x)). Что логично: если x идет раньше y, то y должен идти позже x, и наоборот.
* Если x.compareTo(y) > 0 и y.compareTo(z) > 0, то x.compareTo(z) > 0 – соотношение транзитивности неравенств.
* Если x.compareTo(y) == 0, то sign(x.compare(z)) == sign(y.compareTo(z)), для любых z.
* Вызов x.compareTo(null) должен бросать исключение NullPointerException. В этом есть расхождение с логикой реализации equals (напомню, x.equals(null) возвращает false).
* Если y по своему типу не может быть сравнен с x, то вызов x.compareTo(y) должен бросать исключение ClassCastException.
* (x.compareTo(y) == 0) == x.equals(y), т.е. вызов x.compareTo(y) должен возвращать 0 тогда и только тогда, когда x.equals(y) возвращает true. Это правило непротиворечивости, и его очень важно учитывать.

**Источники:**  
[echuprina.blogspot.ru/2012/02/comparable-comparator.html](http://echuprina.blogspot.ru/2012/02/comparable-comparator.html)  
[www.skipy.ru/technics/objCompTh.html#comparable](http://www.skipy.ru/technics/objCompTh.html#comparable)

2. Как пользоваться интерфейсом Comparator?

В интерфейсе Comparator объявлено два метода compare(Object obj1, Object obj2) и equals(Object obj).  
При использовании интерфейса Comparator, логика сравнения пары объектов не прячется внутрь класса/объекта, а реализуется в отдельном классе.  
  
Метод compare(x,y) в точности соответствует по своей сути вызову x.compareTo(y). Точно так же должны выполняться все правила, что и правила для реализации метода compareTo(Object) интерфейса Comparable.  
Comparator может использоваться в любом месте, где нужна сортировка. При этом, во-первых, появляется необходимая гибкость – возможность реализации нескольких правил сортировки. А во-вторых, сортируемые объекты могут не реализовывать интерфейс Comparable. В случае, если они его все-таки реализуют, Comparator имеет приоритет.  
  
Интерфейс Comparator определяет еще и метод equals(Object), как это ни парадоксально. Этот метод служит для сравнения самих экземпляров интерфейса Comparator и должен возвращать true только в том случае, если сравниваемые объекты обеспечивают одинаковый порядок сортировки. Однако всегда безопасно оставлять исходную реализацию Object.equals(Object) нетронутой  
  
**Источник:**   
[echuprina.blogspot.ru/2012/02/comparable-comparator.html](http://echuprina.blogspot.ru/2012/02/comparable-comparator.html)  
[www.skipy.ru/technics/objCompTh.html#comparable](http://www.skipy.ru/technics/objCompTh.html#comparable)

3. Какие методы есть у класса Collections?

**public static <T> boolean addAll(Collection<? super T> c, T… elements)**  
Метод добавляет элементы массива elements в коллекцию *Collection<? super T> c*. Элементы могут быть указаны по одиночке, либо как массив. Когда элементы указанны по отдельности данный метод предоставляет возможность удобно добавить все элементы в имеющуюся коллекцию:

Collections.addAll(flavors, "Peaches 'n Plutonium", "Rocky Racoon");

**public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)  
public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)**  
  
Оба метода ищут в списке переданном в параметре объект переданный в параметре используя алгоритм двоичного поиска. Возвращают индекс элемента, если такой элемент в списке есть, иначе индекс первого элемента списка большего *key*, если все элементы меньше *key*, возвращает *list.size()*.   
  
Перед использованием данных методов списки должны быть отсортированы. В первом случае отсортированы по возрастанию в «естественном» порядке следования элементов списка (такой же, как при использовании *Collections.sort(list)*). Во втором случае список должен быть отсортирован по возрастанию в порядке следования, который обеспечивает переданный компаратор (такой же порядок, как при использовании *Collections.sort(list, c)*[здесь «с» — компаратор из параметров описываемого метода])  
  
**public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type)**  
  
Преамбула:  
Механизм дженериков в языке обеспечивает проверку типов во время компиляции. Обычно этого и достаточно, однако бывают случаи, когда все-таки нет. К примеру мы нашу коллекцию передаем в код библиотеки, куда-нибудь на сторону, нам неизвестную, и нам очень сильно хочется, чтоб код этой «third-party library» не вставил в нашу коллекцию элемент неправильного типа. Это вот возможная проблема номер 1.   
Возможная проблема номер 2 следующая. Предположим наша программа выдает нам ClassCastException, который оповещает нас о том, что в коллекцию был вставлен элемент неправильного типа. К сожалению данное исключение может вылететь в любой момент, после того, как неправильный элемент был вставлен, и обычно предоставляет нам совсем немного или вообще ноль информации об источнике проблемы.   
Конец преамбулы.   
  
Используя метод метод *checkedCollection* мы можем избавить себя от проблемы один и два, т.к. этот метод создает коллекцию проверяемую на этапе выполнения.   
Решение проблемы номер два, с помощью данного метода:   
К примеру мы имеем вот это, и у нас вываливается ClassCastException.

Collection<String> c = new HashSet<String>();

Код выше можно временно заменить на:

Collection<String> c = Collections.checkedCollection(  
         new HashSet<String>(), String.class);

При запуске программы снова мы локализуем строку кода, которая вставляет элемент неправильного типа в нашу коллекцию.   
  
Родственные на мой взгляд методы: 

public static <E> List<E> checkedList(List<E> list,Class<E> type)  
public static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType,Class<V> valueType)  
public static <E> Set<E> checkedSet(Set<E> s,Class<E> type)  
public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m,Class<K> keyType,Class<V> valueType)  
public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s,Class<E> type)

**public static <T> void copy(List<? super T> dest,List<? extends T> src)**  
Метод копирует элементы src в dest. индексы у копированных элементов будут совпадать.   
  
**public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)  
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)  
public static <T> T min(Collection<? extends T> coll,Comparator<? super T> comp)  
public static <T> T max(Collection<? extends T> coll,Comparator<? super T> comp)**  
  
методы возвращают минимальный\максимальный элемент в коллекции с точки зрения «естественного порядка»(интерфейс Comparable) либо порядка переданного компаратора.  
  
**public static boolean disjoint(Collection<?> c1,Collection<?> c2)**  
Возвращает true если у коллекций нет одинаковых элементов.   
  
**<T> List <T> emptyList(), <K,V> Map <K,V> emptyMap(),  
<T> Set <T> emptySet()** – возвращают пустой список, карту отображения  
и множество соответственно;  
  
**<T> void fill(List<? super T> list, T obj)** – заполняет список заданным элементом;  
  
**int frequency(Collection<?> c, Object o)** – возвращает количество вхождений в коллекцию заданного элемента;  
  
**<T> List <T> nCopies(int n, T o)** – возвращает список из *n* заданных элементов;  
  
**<T> boolean replaceAll(List<T> list, T oldVal, T newVal)** – заменяет все заданные элементы новыми;  
  
**void reverse(List<?> list)** – “переворачивает” список;  
  
**void rotate(List<?> list, int distance)** – сдвигает список циклически на заданное число элементов;  
  
**void shuffle(List<?> list)** – перетасовывает элементы списка;  
  
**<T> Set <T> singleton(T o), singletonList(T o), singletonMap(K key, V value)** – создают множество, список и карту отображения, состоящие из одного элемента;  
  
**<T extends Comparable<? super T>> void sort(List<T> list),  
<T> void sort(List<T> list, Comparator<? super T> c)** – сортировка списка, естественным порядком и используя Comparator соответственно;  
  
**void swap(List<?> list, int i, int j)** – меняет местами элементы списка стоящие на заданных позициях.  
  
Источники:  
[crypto.pp.ua/2010/06/klass-collections-v-java/](http://crypto.pp.ua/2010/06/klass-collections-v-java/)  
[docs.oracle.com/javase/7/docs/api/java/util/Collections.html](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html)

4. Какие методы есть у класса Arrays?

Полный перечень методов класса Arrays можно увидеть в [документации](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html). В данном конспекте приведу лишь некоторые из них. [переводил методы из документации, и к сожалению потерял большую часть своего перевода. Обидно, и тратить время на тоже самое не хочется, так что вставлю нагугленное]  
  
**public static <T> List<T> asList(T… a)**  
формирует список на основе массива. Массив при этом используется для внутреннего представления списка. Таким образом сохраняется связь между списком и исходным массивом:  
  
изменения в массиве отразятся на списке:

String[] a = { "foo", "bar", "baz"};  
List<String> list = Arrays.asList(a);  
System.out.println(list); // [foo, bar, baz]  
  
a[0] = "aaa";  
System.out.println(list); // [aaa, bar, baz]

изменения в списке отразятся на массиве:

String[] a = { "foo", "bar", "baz"};  
List<String> list = Arrays.asList(a);  
System.out.println(list); // [foo, bar, baz]  
  
list.set(0, "bbb");  
System.out.println(Arrays.toString(a)); // [bbb, bar, baz]

Если массив содержит объекты, очевидно, и массив и список будут ссылаться на одни и те же экземпляры:

Object[] a = { new Object(), new Object(), new Object()};  
List<Object> list = Arrays.asList(a);  
System.out.println(a[0] == list.get(0)); // true

**int binarySearch(параметры)** – перегруженный метод организации бинарного поисказначения в массивах примитивных и объектных типов. Возвращает позицию первого совпадения;  
  
**void fill(параметры)** – перегруженный метод для заполнения массивов значениями различных типов и примитивами;  
  
**void sort(параметры)** – перегруженный метод сортировки массива или его части с использованием интерфейса Comparator и без него;  
  
**static <T> T[] copyOf(T[] original, int newLength)** –заполняет массив определенной длины, отбрасывая элементы или заполняя null при необходимости;  
  
**static <T> T[] copyOfRange(T[] original, int from, int to)** – копирует заданную область массива в новый массив;  
  
**<T> List<T> asList(T… a)** – метод, копирующий элементы массива в объект типа List<T>.  
  
Источник:  
[crypto.pp.ua/2010/06/klass-arrays-v-java/](http://crypto.pp.ua/2010/06/klass-arrays-v-java/)

5. Как называется сортировка, которая используется при вызове Collections.sort()?

[Из документации:](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List))  
Реализация является адаптированным вариантом сортировки списка для Python Тима Петерса (TimSort). Данная реализация сбрасывает список в массив, сортирует массив, затем проходит по списку и перезагружает каждый элемент списка из соответствующего элемента массива. Это позволяет избежать сложности n\*n log(n), которая возникла бы при попытки отсортировать связный список напрямую  
  
[Из вики:](https://ru.wikipedia.org/wiki/Timsort)  
Timsort — гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году Тимом Петерсом. В настоящее время Timsort является стандартным алгоритмом сортировки в Python, OpenJDK 7 и реализован в Android JDK 1.5. Основная идея алгоритма в том, что в реальном мире сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. На таких данных Timsort существенно быстрее многих алгоритмов сортировки.

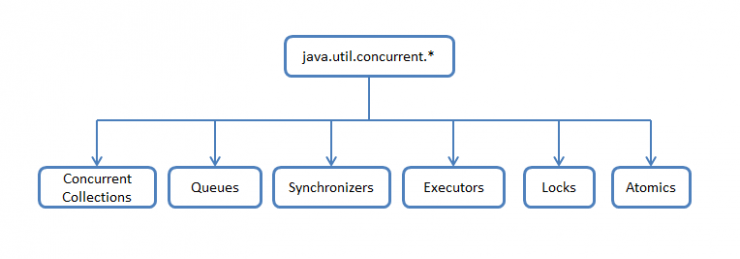
10. Что такое итератор?

Представленный в релизе JDK 1.2 языка Java интерфейс *java.util.Iterator*обеспечивает итерацию контейнерных классов. Каждый Iterator реализует методы *next()* и *hasNext()* и дополнительно может поддерживать метод *remove()*. Итераторы создаются соответствующими контейнерными классами, как правило методом *iterator()*.  
  
Метод *next()* переводит итератор на следующее значение и возвращает указываемое значение итератору. При первоначальном создании итератор указывает на специальное значение, находящееся перед первым элементом, поэтому первый элемент можно получить только после первого вызова *next()*. Для определения момента, когда все элементы в контейнере были перебраны, используется тестовый метод *hasNext()*. Следующий пример демонстрирует простое использование итераторов:

Iterator iter = list.iterator();  
//Iterator<MyType> iter = list.iterator(); в J2SE 5.0  
while (iter.hasNext())  
    System.out.println(iter.next());

Для коллекции типов, поддерживающей подобное, метод итератора *remove()* удаляет последний 'посещенный' элемент из контейнера. Почти все остальные типы модификации контейнера во время итерации являются небезопасными.  
  
Кроме того, для *java.util.List* существует *java.util.ListIterator* со схожим API, но позволяющем прямую и обратную итерации, обеспечивая определение текущего индекса в списке и переход к элементу по его позиции.

6. Что такое канкаренси?

Concurrency – это библиотека классов в Java, в которой собрали специальные классы, оптимизированные для работы из нескольких нитей. Эти классы собраны в пакете java.util.concurrent. Их можно схематично поделить по функциональному признаку следующим образом:  
  
  
**Concurrent Collections** — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам.  
  
**Queues** — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'a.  
  
**Synchronizers** — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.  
  
**Executors** — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.   
  
**Locks** — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.  
  
**Atomics** — классы с поддержкой атомарных операций над примитивами и ссылками.  
  
Источник:   
[habrahabr.ru/company/luxoft/blog/157273/](https://habrahabr.ru/company/luxoft/blog/157273/)

7. Какие классы из «канкаренси» ты знаешь?

Ответ на этот вопрос отлично изложен в [этой статье](https://habrahabr.ru/company/luxoft/blog/157273/). Смыла перепечатывать всю её сюда я не вижу, поэтому приведу описания только тех классов, с которыми имел честь вскользь ознакомиться.   
  
**ConcurrentHashMap<K, V>** — В отличие от Hashtable и блоков synhronized на HashMap, данные представлены в виде сегментов, разбитых по hash'ам ключей. В результате, для доступа к данным лочится по сегментам, а не по одному объекту. В дополнение, итераторы представляют данные на определенный срез времени и не кидают ConcurrentModificationException.   
  
**AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray**— Что если в классе нужно синхронизировать доступ к одной простой переменной типа int? Можно использовать конструкции с synchronized, а при использовании атомарных операций set/get, подойдет также и volatile. Но можно поступить еще лучше, использовав новые классы Atomic\*. За счет использования CAS, операции с этими классами работают быстрее, чем если синхронизироваться через synchronized/volatile. Плюс существуют методы для атомарного добавления на заданную величину, а также инкремент/декремент.

8. Как устроен класс ConcurrentHashMap?

К моменту появления ConcurrentHashMap Java-разработчики нуждались в следующей реализации хэш-карты:

* Потокобезопасность
* Отсутствие блокировок всей таблицы на время доступа к ней
* Желательно, чтобы отсутствовали блокировки таблицы при выполнении операции чтения

Основные идеи реализации ConcurrentHashMap следующие:   
  
**1. Элементы карты**  
  
В отличие от элементов HashMap, Entry в ConcurrentHashMap объявлены как volatile. Это важная особенность, также связанная с изменениями в [JMM](https://habrahabr.ru/post/133981/).

static final class HashEntry<K, V> {  
    final K key;  
    final int hash;  
    volatile V value;  
    final HashEntry<K, V> next;  
  
    HashEntry(K key, int hash, HashEntry<K, V> next, V value) {  
        this .key = key;  
        this .hash = hash;  
        this .next = next;  
        this .value = value;  
     }  
  
    @SuppressWarnings("unchecked")  
    static final <K, V> HashEntry<K, V>[] newArray(int i) {  
        return new HashEntry[i];  
    }  
}

**2. Хэш-функция**  
  
В ConcurrentHashMap также используется улучшенная функция хэширования.  
Напомню, какой она была в HashMap из JDK 1.2:

static int hash(int h) {  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}

Версия из ConcurrentHashMap JDK 1.5:

private static int hash(int h) {  
    h += (h << 15) ^ 0xffffcd7d;  
    h ^= (h >>> 10);  
    h += (h << 3);  
    h ^= (h >>> 6);  
    h += (h << 2) + (h << 14);  
    return h ^ (h >>> 16);  
}

В чём необходимость усложнения хэш-функции? Таблицы в хэш-карте имеют длину, определяемую степенью двойки. Для хэш-кодов, двоичные представления которых не различаются в младшей и старшей позиции, мы будем иметь коллизии. Усложнение хэш-функции как раз решает данную проблему, уменьшая вероятность коллизий в карте.  
  
**3. Сегменты**  
  
Карта делится на N различных сегментов (16 по умолчанию, максимальное значение может быть 16-битным и представлять собой степень двойки). Каждый сегмент представляет собой потокобезопасную таблицу элементов карты. Увеличение количества сегментов будет способствовать тому, что операции модификации будут затрагивать различные сегменты, что уменьшит вероятность блокировок во время выполнения.  
  
**4. ConcurrencyLevel**  
  
Данный параметр влияет на использование картой памяти и количество сегментов в карте.  
Количество сегментов будет выбрано как ближайшая степень двойки, большая чем concurrencyLevel. Занижение concurrencyLevel ведёт к тому, что более вероятны блокировки потоками сегментов карты при записи. Завышение показателя ведёт к неэффективному использованию памяти.  
  
Если лишь один поток будет изменять карту, а остальные будут производить чтение — рекомендуется использовать значение 1.  
  
**Итого**  
  
Итак, основные преимущества и особенности реализации ConcurrentHashMap:

* Карта имеет схожий с hashmap интерфейс взаимодействия
* Операции чтения не требуют блокировок и выполняются параллельно
* Операции записи зачастую также могут выполняться параллельно без блокировок
* При создании указывается требуемый concurrencyLevel, определяемый по статистике чтения и записи
* Элементы карты имеют значение value, объявленное как volatile

Источник: [habrahabr.ru/post/132884/](https://habrahabr.ru/post/132884/)

9. Что такое класс Lock?

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете *java.util.concurrent.locks*.  
  
Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на поток на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.  
  
Классы блокировок реализуют интерфейс Lock, который определяет следующие методы:

* **void lock():** ожидает, пока не будет получена блокировка
* **boolean tryLock():** пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна
* **void unlock():** снимает блокировку
* **Condition newCondition():** возвращает объект Condition, который связан с текущей блокировкой

Организация блокировки в общем случае довольно проста: для получения блокировки вызывается метод lock(), а после окончания работы с общими ресурсами вызывается метод unlock(), который снимает блокировку.  
  
Объект Condition позволяет управлять блокировкой.  
  
Как правило, для работы с блокировками используется класс **ReentrantLock** из пакета *java.util.concurrent.locks.* Данный класс реализует интерфейс Lock.  
  
Рассмотрим использование Java Lock API на примере небольшой программы:  
И так, пусть у нас есть класс Resource с парочкой потокобезопасных методов и методов, где потокобезопасность не требуется.

public class Resource {  
   
    public void doSomething(){  
        // пусть здесь происходит работа с базой данных   
    }  
       
    public void doLogging(){  
        // потокобезопасность для логгирования нам не требуется  
    }  
}

А теперь берем класс, который реализует интерфейс Runnable и использует методы класса Resource.

public class SynchronizedLockExample implements Runnable{  
   
    // экземпляр класса Resource для работы с методами  
    private Resource resource;  
       
    public SynchronizedLockExample(Resource r){  
        this.resource = r;  
    }  
       
    @Override  
    public void run() {  
        synchronized (resource) {  
            resource.doSomething();  
        }  
        resource.doLogging();  
    }  
}

А теперь перепишем приведенную выше программу с использованием Lock API вместо ключевого слова synchronized.

import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
   
// класс для работы с Lock API. Переписан с приведенной выше программы,  
// но уже без использования ключевого слова synchronized  
public class ConcurrencyLockExample implements Runnable{  
   
    private Resource resource;  
    private Lock lock;  
       
    public ConcurrencyLockExample(Resource r){  
        this.resource = r;  
        this.lock = new ReentrantLock();  
    }  
       
    @Override  
    public void run() {  
        try {  
            // лочим на 10 секунд  
            if(lock.tryLock(10, TimeUnit.SECONDS)){  
            resource.doSomething();  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }finally{  
            //убираем лок  
            lock.unlock();  
        }  
        // Для логгирования не требуется потокобезопасность  
        resource.doLogging();  
    }  
   
}

Как видно из программы, мы используем метод tryLock(), чтобы убедиться в том, что поток ждет только определенное время. Если он не получает блокировку на объект, то просто логгирует и выходит.  
  
Еще один важный момент. Необходимо использовать блок try-finally, чтобы убедиться в том, что блокировка будет снята, даже если метод doSomething() бросит исключение.  
  
Источники:  
[metanit.com/java/tutorial/8.9.php](http://metanit.com/java/tutorial/8.9.php)  
[prologistic.com.ua/java-lock-teoriya-i-primer-ispol-zovaniya-concurrency-lock.html](http://prologistic.com.ua/java-lock-teoriya-i-primer-ispol-zovaniya-concurrency-lock.html)

11. Что такое mutex?

Мютекс – это специальный объект для синхронизации нитей/процессов. Он может принимать два состояния – занят и свободен. Если упростить, то мютекс – это boolean-переменная, которая принимает два значения: занят(true) и свободен(false).  
  
Когда нить хочет монопольно владеть некоторым объектом, она помечает его мютекс занятым, а когда закончила работу с ним – помечает его мютекс свободным.  
  
Мютекс прикреплен к каждому объекту в Java. Прямой доступ к мютексу есть только у Java-машины. От программиста он скрыт.

12. Что такое монитор?

Монитор – это специальный механизм (кусок кода) – надстройка над мютексом, который обеспечивает правильную работу с ним. Ведь мало пометить, что объект – занят, надо еще обеспечить, чтобы другие нити не пробовали воспользоваться занятым объектом.  
  
В Java монитор реализован с помощью ключевого слова synchronized.  
  
Когда мы пишем блок synchronized, то компилятор Java заменяет его тремя кусками кода:

1. В начале блока synchronized добавляется код, который отмечает мютекс как занятый.
2. В конце блока synchronized добавляется код, который отмечает мютекс как свободный.
3. Перед блоком synchronized добавляется код, который смотрит, если мютекс занят – то нить должна ждать его освобождения.