General:

**1. Multi-Tenancy**

Software Multitenancy refers to a software architecture in which a single instance of a software runs on a server and serves multiple tenants.

A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture,

a software application is designed to provide every tenant a dedicated share of the instance including its data, configuration, user management,

tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures,

where separate software instances operate on behalf of different tenants.

Commentators regard multitenancy as an important feature of cloud computing

ОДин сервис(код один) - много пользователей(может быть и под каждого бд)

**2. REST Semantics**

Все является ресурсами с уникальным идентификатором (URL)

Все операции клиента с сервером stateless, т.е. сервер не должен хранить вообще никакой информации о клиенте – никакой сессии

Все запросы можно поделить на 4 типа в соответствии с CRUD, причем каждому типу сопоставляется HTTP метод – Post, Get, Put и Delete

Вся логика крутится вокруг ресурсов, а не операций

**3. Stateless Web**

Stateless - There's no memory (state) that's maintained by the program

Stateful - The program has a memory (state)

To illustrate the concept of state I'll define a function which is stateful and one which is stateless

Stateless

//The state is derived by what is passed into the function

function int addOne(int number)

{

return number + 1;

}

Stateful

//The state is maintained by the function

private int \_number = 0; //initially zero

function int addOne()

{

\_number++;

return \_number;

}

In computing, a stateless protocol is a communications protocol that treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of request and response. A stateless protocol does not require the server to retain session information or status about each communications partner for the duration of multiple requests. In contrast, a protocol which requires keeping of the internal state on the server is known as a stateful protocol.

Examples of stateless protocols include the Internet Protocol (IP) which is the foundation for the Internet, and the Hypertext Transfer Protocol (HTTP) which is the foundation of data communication for the World Wide Web.

**4. Sticky Sessions**

One solution to this problem is "sticky session" (aka "session affinity") where each user is assigned to a single server and his/her state data is contained on that server exclusively throughout the duration of the session.

Pros:

it's easy-- no app changes required.

better utilizes local RAM caches (e.g. look up user profile once, cache it, and can re-use it on subsequent visits from same user)

Cons:

if the server goes down, session is lost. (note that this is a con of storing session info locally on the web server-- not of sticky sessions per se). if what's in the session is really important to the user (e.g. a draft email) or to the site (e.g. a shopping cart) then losing one of your servers can be very painful.

depending on "sticky" implementation in your load balancer, may direct unequal load to some servers vs. others

bringing a new server online doesn't immediately give the new server lots of load-- if you have a dynamic load-balancing system to deal with spikes, stickiness may slow your ability to respond quickly to a spike. That said, this is somewhat of a corner case and really only applies to very large and sophisticated sites.

if you have relatively few users but a single user's traffic can swamp one server (e.g. complex pages with SSL, AJAX, dynamically-generated images, dynamic compression, etc.), then stickines may hurt end-user response time since you're not spreading a single user's load evenly across servers. If you have a lot of concurrent users, this is a non-issue since all your servers will be swamped!

But if you must use server-local session state, sticky sessions are definitely the way to go-- and even if you don't use server-local session state, stickiness has benefits when it comes to cache utilization (see above). Your load balancer should be able to look at HTTP cookies (not only IP address) to determine stickiness, since IP addresses can change during a single session (e.g. docking a laptop between a wired and wireless network).

Even better, don't use session state on the web server at all! If session state is very painful to lose (e.g. shopping carts), store it in a central database and clear out old sessions periodically. If session state is not critical (e.g. username/avatar URL), then stick it in a cookie-- just make sure you're not shoving too much data into the cookie.

Modern versions of Rails, by default, store session variables in a cookie for the reasons above. Other web frameworks may have a "store in cookie" and/or "store in DB" option.

**5. MVC, DI, IoC**

Инверсия управления (англ. Inversion of Control, IoC) — это принцип объектно-ориентированного программирования, при котором объекты программы не зависят от конкретных реализаций других объектов, но могут иметь знание об их абстракциях (интерфейсах) для последующего взаимодействия.

Внедрение зависимостей (англ. Dependency Injection) — это композиция структурных шаблонов проектирования, при которой за каждую функцию приложения отвечает один, условно независимый объект (сервис), который может иметь необходимость использовать другие объекты (зависимости), известные ему интерфейсами. Зависимости передаются (внедряются) сервису в момент его создания.

http://habrahabr.ru/post/62830/

https://en.wikipedia.org/wiki/Service\_locator\_pattern

http://habrahabr.ru/post/131993/

IOC реализация:

1.DI

2.Service\_locator

3.Factory

https://gist.github.com/codedokode/e1d31a31b37d5f635057

**6. Distributed vs. Replicated Cache**

https://docs.oracle.com/cd/E15357\_01/coh.360/e15723/cache\_intro.htm#COHDG319

**7. RDBMS, SQL, Transactions, ACID**

Требования ACID

Atomicity — Атомарность

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной. Поскольку на практике невозможно одновременно и атомарно выполнить всю последовательность операций внутри транзакции, вводится понятие «отката» (rollback): если транзакцию не удаётся полностью завершить, результаты всех её до сих пор произведённых действий будут отменены и система вернётся во «внешне исходное» состояние — со стороны будет казаться, что транзакции и не было. (Естественно, счётчики, индексы и другие внутренние структуры могут измениться, но, если СУБД запрограммирована без ошибок, это не повлияет на внешнее её поведение.)

Consistency — Согласованность

Основная статья: Согласованность данных

Транзакция достигающая своего нормального завершения (EOT – end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты. Это условие является необходимым для поддержки четвертого свойства.

Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисление, то система останется в некорректном состоянии и свойство согласованности будет нарушено.

Наконец, ещё одно замечание касается того, что в ходе выполнения транзакции согласованность не требуется. В нашем примере, списание и зачисление будут, скорее всего, двумя разными подоперациями и между их выполнением внутри транзакции будет видно несогласованное состояние системы. Однако не нужно забывать, что при выполнении требования изоляции, никаким другим транзакциям эта несогласованность не будет видна. А атомарность гарантирует, что транзакция либо будет полностью завершена, либо ни одна из операций транзакции не будет выполнена. Тем самым эта промежуточная несогласованность является скрытой.

Isolation — Изолированность[править | править вики-текст]

Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Изолированность — требование дорогое, поэтому в реальных БД существуют режимы, не полностью изолирующие транзакцию (уровни изолированности Repeatable Read и ниже).

Durability — Надежность[править | править вики-текст]

Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

**8. NoSQL, CAP**

<http://softwaremaniacs.org/blog/2010/01/31/brewers-cap-theorem/>

**9. ORM (Hibernate, JPA)**

ORM (англ. Object-relational mapping, рус. Объектно-реляционное отображение) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

JPA – это технология, обеспечивающая объектно-реляционное отображение простых JAVA объектов и предоставляющая API для сохранения, получения и управления такими объектами.

JPA – это спецификация (документ, утвержденный как стандарт, описывающий все аспекты технологии), часть EJB3 спецификации.

Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как что-то будет действовать. JPA также определяет интерфейсы, которые должны будут быть реализованы провайдерами. Плюс к этому JPA определяет правила о том, как должны описываться метаданные отображения и о том, как должны работать провайдеры. Дальше, каждый провайдер, реализуя JPA определяет получение, сохранение и управление объектами. У каждого провайдера реализация разная.

Реализации JPA:

Hibernate

Oracle TopLink

Apache OpenJPA

JPA состоит из трех основных пунктов:

API – интерфейсы в пакете javax.persistance. Набор интерфейсов, которые позволяют организовать взаимодействие с ORM провайдером.

JPQL – объектный язык запросов. Очень похож на SQL, но запросы выполняются к объектам.

Metadata – аннотации над объектами. Набор аннотаций, которыми мы описываем метаданные отображения. Тогда уже JPA знает какой объект в какую таблицу нужно сохранить. Метаданные можно описывать двумя способами: XML-файлом или через аннотации.

**10. Actors/Akka**

**11. Async I/O (Promises/Futures)**

<http://habrahabr.ru/post/112960/>

<http://12devs.co.uk/articles/promises-an-alternative-way-to-approach-asynchronous-javascript/>

<http://www.slideshare.net/xmlilley/mastering-async-io-in-javascript-promises-async-190913>

**12. Ruby/Python – Duck Typing, Dynamic Typing**

<http://www.voidspace.org.uk/python/articles/duck_typing.shtml>

The idea is that you don't need a type in order to invoke an existing method on an object - if a method is defined on it, you can invoke it.  
<http://rubylearning.com/satishtalim/duck_typing.html>

**13. JavaScript: jQuery – What is Selector?**

Селекторами называют строчные выражения, с помощью которых задаются условия поиска элементов DOM на странице. Вы можете находить интересующие вас элементы, используя выражения в стиле CSS 1-3, в дополнении с некоторыми другими возможностями. Селекторы позволяют находить элементы по различным признакам: значению атрибутов, содержимому элементов, родительским элементам, дочерним элементам, порядковым номерам, ну и конечно по именам классов, идентификаторов и/или тегов.

Например выражение $('div') осуществит поиск всех div-элементов на странице, $('.className') найдет все элементы с классом *className* и т. д. В данном примере селекторами являются строки 'div' и '.className', а $( ) — это функция, которая осуществляет поиск элементов по заданному селектору (а так же имеет массу [других возможностей)](http://jquery.page2page.ru/index.php5/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_jQuery). Ниже представлены все организованные в jQuery селекторы и правила их комбинирования.

При необходимости, можно [создавать собственные селекторы](http://jquery.page2page.ru/index.php5/%D0%A1%D0%BE%D0%B7%D0%B4%D0%B0%D0%BD%D0%B8%D0%B5_%D1%81%D0%BE%D0%B1%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D1%85_%D1%81%D0%B5%D0%BB%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%BE%D0%B2_jQuery).

**14. JavaScript: MVC (AngularJS, Backbone) – What is Binding?**

**15. CSS, LESS, NodeJS**

**CSS** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Cascading Style Sheets* — *каскадные таблицы стилей*) — [формальный язык](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) описания внешнего вида документа, написанного с использованием [языка разметки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8).

Преимущественно используется как средство описания, оформления внешнего вида [веб-страниц](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0), написанных с помощью [языков разметки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8) [HTML](https://ru.wikipedia.org/wiki/HTML) и [XHTML](https://ru.wikipedia.org/wiki/XHTML), но может также применяться к любым [XML-документам](https://ru.wikipedia.org/wiki/XML), например, к [SVG](https://ru.wikipedia.org/wiki/SVG) или [XUL](https://ru.wikipedia.org/wiki/XUL).

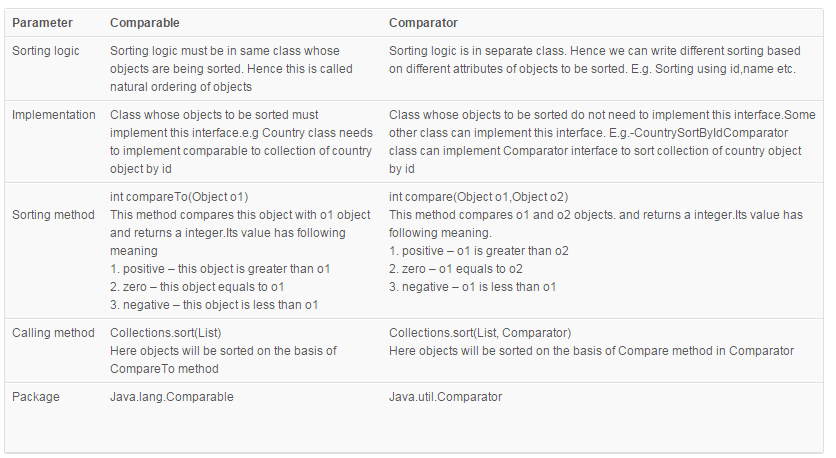
Java:

- Java’s Object: State & Behavior

- Interface, Inheritance

- By-Reference/By-Value

- TreeSet/Comparable/Comparator/NavigableSet/SortedSet



Comparable  
public class Country implements Comparable{

@Override

public int compareTo(Object arg0) {

Country country=(Country) arg0;

return (this.countryId < country.countryId ) ? -1: (this.countryId > country.countryId ) ? 1:0 ;

}}

Comparator

public class CountrySortByIdComparator implements Comparator<Country>{

@Override

public int compare(Country country1, Country country2) {

return (country1.getCountryId() < country2.getCountryId() ) ? -1: (country1.getCountryId() > country2.getCountryId() ) ? 1:0 ;

}

}

**SortedSet.** A [Set](http://docs.oracle.com/javase/6/docs/api/java/util/Set.html) that further provides a *total ordering* on its elements. The elements are ordered using their [natural ordering](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html), or by a [Comparator](http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html) typically provided at sorted set creation time.

**NavigableSet.** A [SortedSet](http://docs.oracle.com/javase/6/docs/api/java/util/SortedSet.html) extended with navigation methods reporting closest matches for given search targets. Methods lower, floor, ceiling, and higher return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element. A NavigableSet may be accessed and traversed in either ascending or descending order. The descendingSet method returns a view of the set with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. This interface additionally defines methods pollFirst and pollLast that return and remove the lowest and highest element, if one exists, else returning null. Methods subSet, headSet, and tailSet differ from the like-named SortedSet methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Subsets of any NavigableSet must implement the NavigableSet interface.

**Treeset**. A [NavigableSet](http://docs.oracle.com/javase/7/docs/api/java/util/NavigableSet.html) implementation based on a [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html). The elements are ordered using their [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html), or by a [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

- HashMap: hashCode() & equals()

- ArrayList vs. LinkedList

- ConcurrentHashMap

Hashtable полностью синхронизирован, только 1 поток за один раз может работать с картой. Очень медленная работа.

Синхронизированные обрамления коллекций synchronizedMap и synchronizedList иногда называют условно потокобезопасными - все операции в отдельности потокобезопасны, но последовательности операций, где управляющий поток зависит от результатов предыдущих операций, могут быть причиной конкуренции за данные.

Однако итераторы, реализованные в классах коллекций java.util, часто подвержены сбоям, что означает, что если один поток изменяет коллекцию в то время, когда другой пропускает её через Iterator, очередной вызов Iterator.hasNext() или Iterator.next() выбросит ConcurrentModificationException. Как и с предыдущим примером, если вы хотите предотвратить ConcurrentModificationException, вам надо целиком блокировать List в то время, когда вы выполняете повторения, заключив его внутрь блока synchronized, который синхронизируется с List l.

**Листинг 1. Типичный случай конкуренции в синхронизированных map-таблицах**

Map m = Collections.synchronizedMap(new HashMap());

List l = Collections.synchronizedList(new ArrayList());

// put-if-absent idiom - contains a race condition

// may require external synchronization

if (!map.containsKey(key))

map.put(key, value);

// ad-hoc iteration - contains race conditions

// may require external synchronization

for (int i=0; i<list.size(); i++) {

doSomething(list.get(i));

}

// normal iteration - can throw ConcurrentModificationException

// may require external synchronization

for (Iterator i=list.iterator(); i.hasNext(); ) {

doSomething(i.next());

}

Класс ConcurrentHashMap из util.concurrent (который также появится в пакете java.util.concurrent в JDK 1.5) - это потокобезопасная реализация Map, предоставляющая намного большую степень параллелизма, чем synchronizedMap. Сразу много операций чтения могут почти всегда выполняться параллельно, одновременные чтения и записи могут обычно выполняться параллельно, а сразу несколько одновременных записей могут зачастую выполняться параллельно.

Операция извлечения возвратит значение, вставленное самой последней завершившейся операцией вставки, а также может возвратить значение, добавленное операцией вставки, выполняемой в настоящее время

Итераторы могут отображать или не отображать вставки или удаления, имевшие место со времени, когда итератор был сконструирован.

ConcurrentHashMap does not allow NULL values .

Отсутствие блокировок всей таблицы на время доступа к ней

отсутствовали блокировки таблицы при выполнении операции чтения

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.   
Между хэш-кодами ключей и соответствующими им сегментами устанавливается зависимость на основе применения к старшим разрядам хэш-кода битовой маски.

- Atomics

- Concurrency: synchronized/volatile, happens-before (JMM)

Простой пример этого можно увидеть в следующем фрагменте кода:

class Reordering {

int x = 0, y = 0;

public void writer() {

x = 1;

y = 2;

}

public void reader() {

int r1 = y;

int r2 = x;

}

}

Давайте предположим, что этот код выполняется в двух потоках одновременно и чтение 'у' возвращает значение 2. Поскольку эта запись расположена после записи в 'х', программист может предположить, что чтение 'х' должно вернуть значение 1. Тем не менее, запись в 'x' и 'y', возможно, были переупорядочены. Если это имело место, то могла произойти запись в 'у', затем чтение обеих переменных, и только потом запись в 'х'.Результатом будет то, что r1 имеет значение 2, а r2 имеет значение 0.

**Комментарий переводчика**

Предполагается, что у одного и того же объекта метод reader() и метод writer() «почти одновременно» вызываются из различных потоков.

Модель памяти Java описывает то, какое поведение является законным в многопоточном коде и, как потоки могут взаимодействовать через память. Она описывает отношения между переменными в программе и низкоуровневые детали сохранения и извлечения их в и из памяти или регистров в реальной компьютерной системе. Модель определяет это таким образом, что это может быть реализовано корректно используя широкий спектр аппаратного оборудования и большое разнообразие оптимизаций компилятора.  
  
Java включает в себя несколько языковых конструкций, в том числе volatile, final и synchronized, которые предназначены, для того, чтобы помочь программисту описать компилятору требования к параллелизму в программе. Модель памяти Java определяет поведение volatile и synchronized, и, что более важно, гарантирует, что корректно синхронизированная Java-программа работает правильно на всех процессорных архитектурах.

JSR 133 определяет новую модель памяти для Java

#### Что делает синхронизация?

Синхронизация имеет несколько аспектов. Наиболее хорошо понимаемый является взаимное исключение (mutual exclusion) — только один поток может владеть монитором, таким образом синхронизации на мониторе означает, что как только один поток входит в synchronized-блок, защищенный монитором, никакой другой поток не может войти в блок, защищенный этом монитором пока первый поток не выйдет из synchronized-блока.  
  
Но синхронизация — это больше чем просто взаимное исключение. Синхронизация гарантирует, что данные записанные в память до или в синхронизированном блоке становятся предсказуемо видимыми для других потоков, которые синхронизируются на том же мониторе. После того как мы выходим из синхронизированного блока, мы освобождаем (release) монитор, что имеет эффект сбрасывания (flush) кэша в оперативную память, так что запись сделанные нашим потоком могут быть видимыми для других потоков. Прежде чем мы сможем войти в синхронизированный блок, мы захватываем (asquire) монитор, что имеет эффект объявления недействительными данных локального процессорного кэша (invalidating the local processor cache), так что переменные будут загружены из основной памяти. Тогда мы сможем увидеть все записи, сделанные видимым предыдущим освобождением (release) монитора.  
  
Обсуждая ситуацию в терминах кэшей, может показаться, что эти вопросы влияют только на многопроцессорные машины. Тем не менее, эффекты переупорядочения можно легко увидеть и на одном процессоре. Компилятор не может переместить ваш код до захвата монитора или после освобождения. Когда мы говорим, что захват и освобождение мониторов действуют на кэши, мы используем сокращение для ряда возможных эффектов.  
  
Семантика новой модели памяти накладывает частичный порядок на операции с памятью (чтение поля, запись поля, захват блокировки (lock), освобождение блокировки (unlock)) и другие операции с потоками (start(), join()). Некоторые действия, как говорят, «происходят прежде» (happens before) других. Когда одно действие «происходит прежде» (happens before) другого, первое будет гарантированно расположено до и видно второму. Правила этого упорядочения таковы:

**Комментарий переводчика**

[Частичный порядок](http://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BD%D0%BE%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%B0) — это не оборот речи, а математическое понятие.

1. Каждое действие в потоке «происходит прежде» (happens before) любого другого действия в этом потоке, которое идет «ниже» в коде этого потока.
2. Освобождение монитора «происходит прежде» (happens before) каждого последующего захвата **того же самого монитора**.
3. Запись в volatile-поле происходит «происходит прежде» (happens before) каждого последующего чтения **того же самого volatile-поля**.
4. Вызов метода start() потока «происходит прежде» (happens before) любых действий в запущенном потоке.
5. Все действия в потоке «происходят прежде» (happens before) любых действий любого другого потока, который успешно завершил ожидание на join() по первому потоку.

Это означает, что любые операции с памятью, которые были видны в потоке перед выходом из синхронизированного блока видны любому другому потоку после того, как он заходит в синхронизированный блок, защищенный тем же монитором, так как все операции с памятью «произойдут прежде» освобождения монитора, а освобождение монитора «происходит прежде» захвата.

**Комментарий переводчика**

Другим следствием является то, что следующий шаблон, который некоторые люди используют, чтобы установить барьер памяти, не работает:

synchronized (new Object()) {}

Это конструкция является на самом деле «пустышкой» (no-op), и ваш компилятор может удалить ее полностью, потому что компилятор знает, что никакой другой поток не будет синхронизироваться на том же мониторе. Вы должны установить отношение «происходит прежде» отношения для одного потока, чтобы увидеть результаты другого.  
  
**Важное примечание**: Обратите внимание, важно для обоих потоков синхронизироваться на одном и том же мониторе, чтобы установить отношение «происходит прежде» (happens before relationship) должным образом. Это не тот случай, когда все видимое потоку A, когда он синхронизируется на объекте X становится видно потоку B после того, как тот синхронизирует на объекте Y. Освобождение и захват должны «соответствовать» (то есть, быть выполнены с одним и тем же монитором), чтобы была обеспечена правильная семантика. В противном случае код содержит гонку данных (data race).

**Комментарий переводчика**

Следующая программа может как остановиться, так и не остановиться в рамках обеих моделей памяти (так как в разных потоках происходит захват и освобождение мониторов различных объектов — lockA / lockB)

public class App {

static Object lockA = new Object();

static Object lockB = new Object();

static boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

synchronized (lockA) {

run = false;

}

}

}).start();

while (true) {

synchronized (lockB) {

if (!run) break;

}

}

}

}

Что означает, что следующий код не безопасен для потоков:

myVolatileVar++;

Этот код также может быть записан следующим образом:

int temp = 0;  
synchronize( myVolatileVar ) {  
  temp = myVolatileVar;  
}  
   
temp++;  
   
synchronize( myVolatileVar ) {  
  myVolatileVar = temp;  
}

Другими словами, если volatile переменная обновляется неявно, то есть значение читается, измененяется, а затем присваивается как новое, результат будет не-потокобезопасным между двумя синхронными операциями. Вы можете выбирать, следует ли использовать синхронизацию или рассчитывать на поддержку JRE автоматической синхронизации volatile переменных. Наилучший подход зависит от вашего случая: если присвоенное значение volatile переменной зависит от её текущего значения (например, во время операции инкремента), то нужно использовать синхронизацию, если вы хотите, чтобы операция была потокобезопасной.

#### Что делает volatile?

volatile-поля являются специальными полями, которые используются для передачи состояние между потоками. Каждое чтение из volatile возвратит результат последней записи любым другим потоком; по сути, они указываются программистом как поля, для которых не приемлемо увидеть «несвежее» (stale) значение в результате кэширования или переупорядочения. Компилятору и runtime-среде запрещено размещать их в регистрах. Они также должны убедиться, что после записи в volatile данные «проталкиваются» (flushed) из кэша в основную память, поэтому они сразу же становятся видны другим потокам. Аналогично, перед чтением volatile-поля кэш должен быть освобожден, так что мы увидим значение в оперативной памяти, а не в кэше Существуют также дополнительные ограничения на изменение порядка обращения к volatile переменным.  
  
При старой модели памяти, доступ к volatile переменным не могли быть переупорядочены друг с другом, но они могли быть переупорядочены с не-volatile переменными. Это сводило на нет полезность volatile полей как средства передачи сигнала от одного потока к другому.  
  
В соответствии с новой моделью памяти, по-прежнему верно, что volatile переменные не могут быть переупорядочены друг с другом. Разница в том, что теперь уже не так легко изменить порядок между обычными полями расположенными рядом volatile. Запись в volatile поле имеет тот же эффект для памяти как и освобождение монитора (monitor release), а чтение из volatile поля имеет тот же эффект для памяти как и захват монитора (monitor acquire). В сущности, так как новая модель накладывает более строгие ограничения на изменение порядка между доступом к volatile полям и другими полями (volatile или обычным), все, что было видимо для потока A когда он писал в volatile поле f становится видимым для потока B, когда он прочтет f.

**Комментарий переводчика**

И в старой и в новой моделях памяти программа гарантированно остановится и напечатает 1 (data — volatile, run — volatile)

public class App {

static volatile int data = 0;

static volatile boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

data = 1;

run = false;

}

}).start();

while (run) {/\*NOP\*/};

System.out.println(data);

}

}

И в старой и в новой моделях памяти программа гарантированно остановится. В новой модели гарантированно напечатает 1, в старой может 0 или 1 (data — НЕ volatile, run — volatile), так в новой нельзя переносить запись в не-volatile «ниже» чем запись в volatile, а в старой — можно

public class App {

static int data = 0;

static volatile boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

data = 1;

run = false;

}

}).start();

while (run) {/\*NOP\*/};

System.out.println(data);

}

}

И в старой и в новой моделях памяти программа может НЕ остановиться (run — не volatile и может «залипнуть» в кэше). В обеих моделях если остановится, то может напечатать как 1, так и 0 (data — НЕ volatile, run — НЕ volatile), так как можно менять порядок независимых записей в не-volatile поля

public class App {

static int data = 0;

static boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

data = 1;

run = false;

}

}).start();

while (run) {/\*NOP\*/};

System.out.println(data);

}

}

Делаем запись во вторую переменную зависимой от записи в первую переменную. И в старой и в новой моделях памяти программа может НЕ остановиться (run — не volatile и может «залипнуть» в кэше). Но теперь в новой модели в случае остановки напечатает гарантированно 1

public class App {

static int data = 0;

static boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

data = 1;

run = (data != 1);

}

}).start();

while (run) {/\*NOP\*/};

System.out.println(data);

}

}

И в старой и в новой моделях памяти программа может НЕ остановиться. В обеих моделях если остановится, то может напечатать как 1, так и 0 (data — volatile, run — НЕ volatile), так как можно переносить запись в не-volatile «выше» чем запись в volatile

public class App {

static volatile int data = 0;

static boolean run = true;

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

data = 1;

run = false;

}

}).start();

while (run) {/\*NOP\*/};

System.out.println(data);

}

}

Вот простой пример того, как volatile поля могут быть использованы

class VolatileExample {

int x = 0;

volatile boolean v = false;

public void writer() {

x = 42;

v = true;

}

public void reader() {

if (v == true) {

//uses x - guaranteed to see 42.

}

}

}

Назовем один поток *писателем*, а другой — *читателем*. Запись в v в *писателе* «сбрасывает» данные x в оперативную память, а чтение v «захватывает» это значение из памяти. Таким образом, если *читатель* увидит значение true поля v, то также гарантированно увидит значение 42 в x. Это не было верно, для старой моделью памяти (в старой — можно было «спустить» запись в не-volatile «ниже» записи в volatile). Если бы v не было volatile, то компилятор мог бы изменить порядок записи в *писателе*, и *читатель* мог бы увидеть 0 в х.

**Комментарий переводчика**

Семантика volatile была существенно усиленна, почти до уровня synchronized. Каждое чтение или запись volatile действует как «половина» synchronized с точки зрения видимости.   
  
**Важное примечание:** Обратите внимание, важно чтобы оба потока сделали чтение-запись по одной и той же volatile переменной, с целью добиться установления happens before отношения. Это не тот случай, когда все, что видимо для потока А, когда он пишет volatile-поле f становится видимым для потока B после того, как он считает volatile-поле g. Чтение и запись должны относиться **к одной и той же volatile-переменной**, чтобы иметь должную семантику.

- java.util.concurrent: Concurrent Collections (CopyOnWrite e.g.), Queues, Synchronizers (Semaphore, CountDownLatch, CyclicBarrier, Exchanger, Phaser), Executors (Future & Callable), Locks, Atomics

Concurrent Collections **.**

**CopyOnWriteArrayList<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом.   
**CopyOnWriteArraySet<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Имплементация интерфейса Set, использующая за основу CopyOnWriteArrayList. В отличии от CopyOnWriteArrayList, дополнительных методов нет.

**ConcurrentMap<K, V> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Интерфейс, расширяющий Map несколькими дополнительными атомарными операциями.

**ConcurrentHashMap<K, V> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— В отличие от Hashtable и блоков synhronized на HashMap, данные представлены в виде сегментов, разбитых по hash'ам ключей. В результате, для доступ к данным лочится по сегментам, а не по одному объекту. В дополнение, итераторы представляют данные на определенный срез времени и не кидают ConcurrentModificationException. Более детально ConcurrentHashMap описан в хабратопике [тут](http://habrahabr.ru/post/132884/).

**Дополнительный конструктор**

**ConcurrentNavigableMap<K,V> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/c29/5ad/47f/c295ad47f86291abd7e8fe2b91b9621c.png**— Расширяет интерфейс NavigableMap и вынуждает использовать ConcurrentNavigableMap объекты в качестве возвращаемых значений. Все итераторы декларируются как безопасные к использованию и не кидают ConcurrentModificationException.  
  
**ConcurrentSkipListMap<K, V> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/c29/5ad/47f/c295ad47f86291abd7e8fe2b91b9621c.png**— Является аналогом TreeMap с поддержкой многопоточности. Данные также сортируются по ключу и гарантируется усредненная производительность log(N) для containsKey, get, put, remove и других похожих операций. Алгоритм работы SkipList описан на [Wiki](http://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D1%81_%D0%BF%D1%80%D0%BE%D0%BF%D1%83%D1%81%D0%BA%D0%B0%D0%BC%D0%B8) и [хабре](http://habrahabr.ru/post/139870/).  
  
**ConcurrentSkipListSet<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/c29/5ad/47f/c295ad47f86291abd7e8fe2b91b9621c.png**— Имплементация Set интерфейса, выполненная на основе ConcurrentSkipListMap.

Queues

Потокобезопасные и неблокирующие имплементации Queue на связанных нодах (linked nodes).   
  
**ConcurrentLinkedQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— В имплементации используется wait-free алгоритм от Michael & Scott, адаптированный для работы с garbage collector'ом. Этот алгоритм довольно эффективен и, что самое важное, очень быстр, т.к. построен на [CAS](http://en.wikipedia.org/wiki/Compare-and-swap). Метод size() может работать долго, т.ч. лучше постоянно его не дергать. Детальное описание алгоритма можно посмотреть тут [тут](http://www.cs.rochester.edu/u/michael/PODC96.html).  
  
**ConcurrentLinkedDeque<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Deque расшифровывается как Double ended queue и читается как «Deck». Это означает, что данные можно добавлять и вытаскивать с обоих сторон. Соответственно, класс поддерживает оба режима работы: FIFO (First In First Out) и LIFO (Last In First Out). На практике, ConcurrentLinkedDeque стоит использовать только, если обязательно нужно LIFO, т.к. за счет двунаправленности нод данный класс проигрывает по производительности на 40% по сравнению с ConcurrentLinkedQueue.

**BlockingQueue<E> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— При обработке больших потоков данных через очереди становится явно недостаточно использования ConcurrentLinkedQueue. Если потоки, разгребающие очередь перестанут справляться с наплывом данных, то можно довольно быстро схлопотать out of memory или перегрузить IO/Net настолько, что производительность упадет в разы пока не настанет отказ системы по таймаутам или из за отсутствия свободных дескрипторов в системе. Для таких случаев нужна queue с возможностью задать размер очереди или с блокировками по условиям. Тут то и появляется интерфейс BlockingQueue, открывающий дорогу к целому набору полезных классов. Помимо возможности задавать размер queue, добавились новые методы, которые реагируют по-разному на незаполнение или переполнение queue. Так, например, при добавлении элемента в переполненную queue, один метод кинет IllegalStateException, другой вернет false, третий заблокирует поток, пока не появится место, четвертый же заблокирует поток с таймаутом и вернет false, если место так и не появится. Также стоит отметить, что блокирующие очереди не поддерживают null значения, т.к. это значение используется в методе poll как индикатор таймаута.   
  
**ArrayBlockingQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Класс блокирующей очереди, построенный на классическом кольцевом буфере. Помимо размера очереди, доступна возможность управлять «честностью» блокировок. Если fair=false (по умолчанию), то очередность работы потоков не гарантируется. Более подробно о «честности» можно посмотреть в описании ReentrantLock'a.  
  
**DelayQueue<E extends Delayed> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Довольно специфичный класс, который позволяет вытаскивать элементы из очереди только по прошествии некоторой задержки, определенной в каждом элементе через метод getDelay интерфейса Delayed.  
  
**LinkedBlockingQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Блокирующая очередь на связанных нодах, реализованная на «two lock queue» алгоритме: один лок на добавление, другой на вытаскивание элемента. За счет двух локов, по сравнению с ArrayBlockingQueue, данный класс показывает более высокую производительность, но и расход памяти у него выше. Размер очереди задается через конструктор и по умолчанию равен Integer.MAX\_VALUE.  
  
**PriorityBlockingQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Является многопоточной оберткой над PriorityQueue. При вставлении элемента в очередь, его порядок определяется в соответствии с логикой Comparator'а или имплементации Comparable интерфейса у элементов. Первым из очереди выходит самый наименьший элемент.  
  
**SynchronousQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Эта очередь работает по принципу один вошел, один вышел. Каждая операция вставки блокирует «Producer» поток до тех пор, пока «Consumer» поток не вытащит элемент из очереди и наоборот, «Consumer» будет ждать пока «Producer» не вставит элемент.  
  
**BlockingDeque<E> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/c29/5ad/47f/c295ad47f86291abd7e8fe2b91b9621c.png**— Интерфейс, описывающий дополнительные методы для двунаправленной блокирующей очереди. Данные можно вставлять и вытаскивать с двух сторон очереди.  
  
**LinkedBlockingDeque<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/c29/5ad/47f/c295ad47f86291abd7e8fe2b91b9621c.png**— Двунаправленная блокирующая очередь на связанных нодах, реализованная как простой двунаправленный список с одним локом. Размер очереди задается через конструктор и по умолчанию равен Integer.MAX\_VALUE.  
  
**TransferQueue<E> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Данный интерфейс может быть интересен тем, что при добавлении элемента в очередь существует возможность заблокировать вставляющий «Producer» поток до тех пор, пока другой поток «Consumer» не вытащит элемент из очереди. Блокировка может быть как с таймаутом, так и вовсе может быть заменена проверкой на наличие ожидающих «Consumer»ов. Тем самым появляется возможность реализации механизма передачи сообщений с поддержкой как синхронных, так и асинхронных сообщений.  
  
**LinkedTransferQueue<E> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Реализация TransferQueue на основе алгоритма Dual Queues with Slack. Активно использует [CAS](http://en.wikipedia.org/wiki/Compare-and-swap) и [парковку](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/sun/misc/Unsafe.java#Unsafe.park%28boolean%2Clong%29) потоков, когда они находятся в режиме ожидания.

Synchronizers

**Semaphore http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— [Семафоры](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счетчика. Если он больше нуля, то доступ разрешается, а значение счетчика уменьшается. Если счетчик равен нулю, то текущий поток блокируется, пока другой поток не освободит ресурс. Количество разрешений и «честность» освобождения потоков задается через конструктор. Узким местом при использовании семафоров является задание количества разрешений, т.к. зачастую это число приходится подбирать в зависимости от мощности «железа».

It is different from CyclicBarrier:  
1) it cannot be reused, used only one time  
2) it does not block these N threads when they call countDown() method. Only thread on await() method waits.  
**CountDownLatch http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Позволяет одному или нескольким потокам ожидать до тех пор, пока не завершится определенное количество операций, выполняющих в других потоках. Классический пример с драйвером довольно неплохо описывает логику класса: Потоки, вызывающие драйвер, будут висеть в методе await (с таймаутом или без), пока поток с драйвером не выполнит инициализацию с последующим вызовом метода countDown. Этот метод уменьшает счетчик count down на единицу. Как только счетчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счетчик count down одноразовый и не может быть сброшен в первоначальное состояние.  
  
**CyclicBarrier http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N-потоков вызовут метод await(...) и заблокируются. После чего счетчик сбрасывается в исходное значение, а ожидающие потоки освобождаются. Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией Runnable интерфейса.  
  
**Exchanger<V> http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Как видно из названия, основное предназначение данного класса — это обмен объектами между двумя потоками. При этом, также поддерживаются null значения, что позволяет использовать данный класс для передачи только одного объекта или же просто как синхронизатор двух потоков. Первый поток, который вызывает метод exchange(...) заблокируется до тех пор, пока тот же метод не вызовет второй поток. Как только это произойдет, потоки обменяются значениями и продолжат свою работу.  
  
**Phaser http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Улучшенная реализация барьера для синхронизации потоков, которая совмещает в себе функционал CyclicBarrier и CountDownLatch, вбирая в себя самое лучшее из них. Так, количество потоков жестко не задано и может динамически меняться. Класс может повторно переиспользоваться и сообщать о готовности потока без его блокировки. Более подробно можно почитать в хабратопике [тут](http://habrahabr.ru/post/117185/).

. Основные управляющие методы:

|  |  |
| --- | --- |
| register() | зарегистрировать участника |
| arrive() | сообщить этапщику о своей готовности, *не* ожидая открытия барьера |
| arriveAndAwaitAdvance() | классическое прибытие на барьер. Точный аналог CyclicBarrier.await() |
| arriveAndDeregister() | отменить свое участие |

Executors

**Future<V> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Замечательный интерфейс для получения результатов работы асинхронной операции. Ключевым методом здесь является метод get, который блокирует текущий поток (с таймаутом или без) до завершения работы асинхронной операции в другом потоке. Также, дополнительно существуют методы для отмены операции и проверки текущего статуса. В качестве имплементации часто используется класс FutureTask.

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

**Callable<V> http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Расширенный аналог интерфейса Runnable для асинхронных операций. Позволяет возвращать типизированное значение и кидать checked exception. Несмотря на то, что в этом интерфейсе отсутсвует метод run(), многие классы java.util.concurrent поддерживают его наряду с Runnable.

**Executor http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Представляет собой базовый интерфейс для классов, реализующих запуск Runnable задач. Тем самым обеспечивается развязка между добавлением задачи и способом её запуска.  
  
**ExecutorService http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Интерфейс, который описывает сервис для запуска Runnable или Callable задач. Методы submit на вход принимают задачу в виде Callable или Runnable, а в качестве возвращаемого значения идет Future, через который можно получить результат. Методы invokeAll работают со списками задач с блокировкой потока до завершения всех задач в переданном списке или до истечения заданного таймаута. Методы invokeAny блокируют вызывающий поток до завершения любой из переданных задач. В дополнении ко всему, интерфейс содержит методы для graceful shutdown. После вызова метода shutdown, данный сервис больше не будет принимать задачи, кидая RejectedExecutionException при попытке закинуть задачу в сервис.

**Executors http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/576/c72/d73576c72fd0c51fe4fffbda47fd2791.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Класс-фабрика для создания ThreadPoolExecutor, ScheduledThreadPoolExecutor. Если нужно создать один из этих пулов, эта фабрика именно то, что нужно. Также, тут содержатся разные адаптеры Runnable-Callable, PrivilegedAction-Callable, PrivilegedExceptionAction-Callable и другие.  
  
**ThreadPoolExecutor http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Очень мощный и важный класс. Используется для запуска асинхронных задач в пуле потоков. Тем самым практически полностью отсутствует оверхэд на поднятие и остановку потоков. А за счет фиксируемого максимума потоков в пуле обеспечивается прогнозируемая производительность приложения. Как было ранее сказано, создавать данный пул предпочтительно через один из методов фабрики Executors. Если же стандартных конфигураций будет недостаточно, то через конструкторы или сеттеры можно задать все основые параметры пула. Более подробно можно ознакомиться в этом [топике](http://habrahabr.ru/post/116363/).

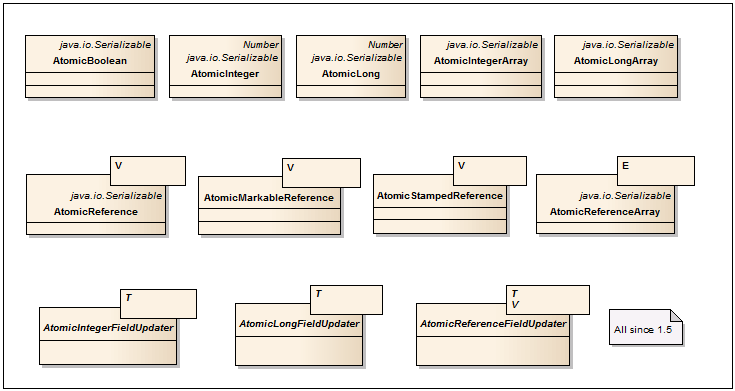
Loks

The ReentrantLock class, which implements Lock, has the same concurrency and memory semantics as synchronized, but also adds features like lock polling, timed lock waits, and interruptible lock waits. Additionally, it offers far better performance under heavy contention. (In other words, when many threads are attempting to access a shared resource, the JVM will spend less time scheduling threads and more time executing them.)

**Condition http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Интерфейс, который описывает альтернативные методы стандарным wait/notify/notifyAll. Объект с условием чаще всего получается из локов через метод lock.newCondition(). Тем самым можно получить несколько комплектов wait/notify для одного объекта.  
  
**Lock http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Базовый интерфейс из lock framework, предоставляющий более гибкий подход по ограничению доступа к ресурсам/блокам нежели при использовании synchronized. Так, при использовании нескольких локов, порядок их освобождения может быть произвольный. Плюс имеется возможность пойти по альтернативному сценарию, если лок уже кем то захвачен.  
  
**ReentrantLock http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Лок на вхождение. Только один поток может зайти в защищенный блок. Класс поддерживает «честную» (fair) и «нечестную» (non-fair) разблокировку потоков. При «честной» разблокировке соблюдается порядок освобождения потоков, вызывающих lock(). При «нечестной» разблокировке порядок освобождения потоков не гарантируется, но, как бонус, такая разблокировка работает быстрее. По умолчанию, используется «нечестная» разблокировка.  
  
**ReadWriteLock http://habrastorage.org/storage2/36b/c31/fa2/36bc31fa2bdf7a08f3a85dff79ee4eef.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Дополнительный интерфейс для создания read/write локов. Такие локи необычайно полезны, когда в системе много операций чтения и мало операций записи.  
  
**ReentrantReadWriteLock http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Очень часто используется в многопоточных сервисах и кешах, показывая очень хороший прирост производительности по сравнению с блоками synchronized. По сути, класс работает в 2-х взаимоисключающих режимах: много reader'ов читают данные в параллель и когда только 1 writer пишет данные.   
  
**ReentrantReadWriteLock.ReadLock http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Read lock для reader'ов, получаемый через readWriteLock.readLock().  
  
**ReentrantReadWriteLock.WriteLock http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Write lock для writer'ов, получаемый через readWriteLock.writeLock().  
  
**LockSupport http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Предназначен для построения классов с локами. Содержит методы для парковки потоков вместо устаревших методов Thread.suspend() и Thread.resume().

AtomicInteger uses combination of volatile & CAS for thread-safe implementation of Integer Counter.

# 6. Atomics

  
  
**AtomicBoolean**, **AtomicInteger**, **AtomicLong**, **AtomicIntegerArray**, **AtomicLongArray http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— 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](http://en.wikipedia.org/wiki/Compare-and-swap), 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.  
  
**AtomicReference http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Класс для атомарных операцией с ссылкой на объект.  
  
**AtomicMarkableReference http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Класс для атомарных операцией со следующей парой полей: ссылка на объект и битовый флаг (true/false).  
  
**AtomicStampedReference http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Класс для атомарных операцией со следующей парой полей: ссылка на объект и int значение.  
  
**AtomicReferenceArray http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Массив ссылок на объекты, который может атомарно обновляться.  
  
**AtomicIntegerFieldUpdater**, **AtomicLongFieldUpdater**,**AtomicReferenceFieldUpdater http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/d73/4a3/deb/d734a3deb05965b8207c68b307cadba8.png**— Классы для атомарного обновления полей по их именам через reflection. Смещение полей для CAS определяется в конструкторе и кешируются, т.ч. тут нет сильного падения производительности из за reflection.

- Fork Join

В java 1.7 появился новый Fork Join фреймворк для решения рекурсивных задач, работающих по алгоритмам [разделяй и влавствуй](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) или [Map Reduce](http://ru.wikipedia.org/wiki/MapReduce). Чтобы было более наглядней, можно привести визуальный пример алгоритма сортировки quicksort:  
[](http://en.wikipedia.org/wiki/Quicksort)  
Так, за счет разбиения на части, можно добиться их параллельной обработки в разных потоках. Для решения этой задачи можно использовать и обычный ThreadPoolExecutor, но за счет частого переключения контекста и отслеживания контроля исполнения все это не очень эффективно работает. Тут то нам приходит на помощь Fork Join framework в основу которого используется [work-stealing](http://classes.engineering.wustl.edu/cse566s/presentations/WorkStealing.pdf) алгоритм. Наиболее хорошо раскрывает себя в системах с большим количеством процессоров. Подробнее можно ознакомиться в блоге [тут](http://www.igvita.com/2012/02/29/work-stealing-and-recursive-partitioning-with-fork-join/) или [публикации](http://gee.cs.oswego.edu/dl/papers/fj.pdf) Doug Lea. Про производительность и масштабируемость можно почитать [тут](http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool).  
  
**ForkJoinPool http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Представляет собой точку входа для запуска корневых (main) ForkJoinTask задач. Подзадачи запускаются через методы задачи, от которой нужно отстрелиться (fork). По умолчанию создается пул потоков с количеством потоков равным количеству доступных для JVM процессоров (cores).  
  
**ForkJoinTask http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/93e/530/ead/93e530ead57ae1e70e6d067602e05095.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Базовый класс для всех Fork Join задач. Из ключевых методов можно отметить: fork() — добавляет задачу в очередь текущего потока ForkJoinWorkerThread для асинхронного выполнения; invoke() — запускает задачу в текущем потоке; join() — ожидает завершения подзадачи с возвращением результата; invokeAll(...) — объединяет все три предыдущие предыдущие операции, выполняя две или более задач за один заход; adapt(...) — создает новую задачу ForkJoinTask из Runnable или Callable объектов.  
  
**RecursiveTask http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/93e/530/ead/93e530ead57ae1e70e6d067602e05095.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Абстрактный класс от ForkJoinTask, с объявлением метода compute, в котором должна производиться асинхронная операция в наследнике.  
  
**RecursiveAction http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/93e/530/ead/93e530ead57ae1e70e6d067602e05095.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Отличается от RecursiveTask тем, не возвращает результат.  
  
**ForkJoinWorkerThread http://habrastorage.org/storage2/0fb/43e/259/0fb43e259847760ecf54027457005e30.pnghttp://habrastorage.org/storage2/968/b76/9b1/968b769b1a83359a83e5fd169d8dbde4.png**— Используется в качестве имплементации по умолчанию в ForkJoinPoll. При желании можно отнаследоваться и перегрузить методы инициализации и завершения worker потока.

# Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the [ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html) class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute [ForkJoinTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html) processes.

## Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

if (my portion of the work is small enough)

do the work directly

else

split my work into two pieces

invoke the two pieces and wait for the results

Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either [RecursiveTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveTask.html) (which can return a result) or [RecursiveAction](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/RecursiveAction.html).

After your ForkJoinTask subclass is ready, create the object that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

## Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

public class ForkBlur extends RecursiveAction {

private int[] mSource;

private int mStart;

private int mLength;

private int[] mDestination;

// Processing window size; should be odd.

private int mBlurWidth = 15;

public ForkBlur(int[] src, int start, int length, int[] dst) {

mSource = src;

mStart = start;

mLength = length;

mDestination = dst;

}

protected void computeDirectly() {

int sidePixels = (mBlurWidth - 1) / 2;

for (int index = mStart; index < mStart + mLength; index++) {

// Calculate average.

float rt = 0, gt = 0, bt = 0;

for (int mi = -sidePixels; mi <= sidePixels; mi++) {

int mindex = Math.min(Math.max(mi + index, 0),

mSource.length - 1);

int pixel = mSource[mindex];

rt += (float)((pixel & 0x00ff0000) >> 16)

/ mBlurWidth;

gt += (float)((pixel & 0x0000ff00) >> 8)

/ mBlurWidth;

bt += (float)((pixel & 0x000000ff) >> 0)

/ mBlurWidth;

}

// Reassemble destination pixel.

int dpixel = (0xff000000 ) |

(((int)rt) << 16) |

(((int)gt) << 8) |

(((int)bt) << 0);

mDestination[index] = dpixel;

}

}

...

Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

protected static int sThreshold = 100000;

protected void compute() {

if (mLength < sThreshold) {

computeDirectly();

return;

}

int split = mLength / 2;

invokeAll(new ForkBlur(mSource, mStart, split, mDestination),

new ForkBlur(mSource, mStart + split, mLength - split,

mDestination));

}

If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in a ForkJoinPool is straightforward, and involves the following steps:

1. Create a task that represents all of the work to be done.
2. // source image pixels are in src
3. // destination image pixels are in dst
4. ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
5. Create the ForkJoinPool that will run the task.
6. ForkJoinPool pool = new ForkJoinPool();
7. Run the task.
8. pool.invoke(fb);

For the full source code, including some extra code that creates the destination image file, see the [ForkBlur](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ForkBlur.java) example.

## Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the ForkBlur.java example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the [java.util.Arrays](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class for its parallelSort() methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of [Project Lambda](http://openjdk.java.net/projects/lambda/) scheduled for the Java SE 8 release. For more information, see the [Lambda Expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) section.

- NIO (Channels/Selectors)

- SoftReference, WeakReference