Java. Многопоточность.

1. 1. Многопоточные приложения в Java. Виталий Унгурян unguryan@itstep.org
2. [2.](https://image.slidesharecdn.com/11-170322121329/95/java-2-638.jpg?cb=1508227507)Процессы Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создаёт отдельный процесс для каждой вкладки, что даёт ему некоторые преимущества, вроде независимости вкладок друг от друга).
3. [3.](https://image.slidesharecdn.com/11-170322121329/95/java-3-638.jpg?cb=1508227507)Процессы Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен.
4. [4.](https://image.slidesharecdn.com/11-170322121329/95/java-4-638.jpg?cb=1508227507)Потоки Поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.
5. [5.](https://image.slidesharecdn.com/11-170322121329/95/java-5-638.jpg?cb=1508227507)Многопоточность Многопоточность операционной системы – возможность одновременного выполнения более чем одной программы. Число одновременно выполняющихся процессов не ограничено количеством процессоров.
6. [6.](https://image.slidesharecdn.com/11-170322121329/95/java-6-638.jpg?cb=1508227507)Для чего нужна многопоточность?
7. [7.](https://image.slidesharecdn.com/11-170322121329/95/java-7-638.jpg?cb=1508227507)Для чего нужна многопоточность? Посчитайте сумму чисел на экране 62 41 36 54 50 10 52 92 44 76 58 83 50 66 31 805
8. [8.](https://image.slidesharecdn.com/11-170322121329/95/java-8-638.jpg?cb=1508227507)Для чего нужна многопоточность? Посчитайте сумму своей группы чисел на экране A 73 37 59 3 24 B 35 39 65 68 71 C 23 55 97 67 16 732
9. [9.](https://image.slidesharecdn.com/11-170322121329/95/java-9-638.jpg?cb=1508227507)Потоки
10. [10.](https://image.slidesharecdn.com/11-170322121329/95/java-10-638.jpg?cb=1508227507)Многопоточность Многопоточные программы расширяют идею многозадачности. Индивидуальные приложения могут выполнять множество задач в одно и то же время. Каждая задача называется потоком – thread (нить).
11. [11.](https://image.slidesharecdn.com/11-170322121329/95/java-11-638.jpg?cb=1508227507)Многопоточность Существенная разница между многими процессами и многими потоками заключается в следующем: Каждый процесс имеет собственный набор переменных, потоки могут разделят одни и те же данные.
12. [12.](https://image.slidesharecdn.com/11-170322121329/95/java-12-638.jpg?cb=1508227507)Многопоточность Потоки являются более «легковесными», чем процессы. Пример многопоточных приложений – браузер, web-сервер, программы с графическим пользовательским интерфейсом.
13. [13.](https://image.slidesharecdn.com/11-170322121329/95/java-13-638.jpg?cb=1508227507)Реализация интерфейса Runnable Самый простой способ создания потока заключается в определении класса, который реализует интерфейс Runnable. Runnable определяет всего один абстрактный метод – run(). При выходе из метода run() поток завершает свое действие.
14. [14.](https://image.slidesharecdn.com/11-170322121329/95/java-14-638.jpg?cb=1508227507)Наследование класса Thread Для создания потока необходимо расширить класс Thread и переопределить метод run(). Как и в случае с реализацией интерфейса в теле метода run() реализуется работа потока, и при выходе из метода поток прекращает свою работу.
15. [15.](https://image.slidesharecdn.com/11-170322121329/95/java-15-638.jpg?cb=1508227507)Какую реализацию выбрать? Реализация интерфейса Runnable используется в случаях, когда класс уже наследует какой-либо родительский класс, и тем самым не позволяет расширить класс Thread. Да и вообще реализация интерфейсов считается хорошим тоном программирования в java.
16. [16.](https://image.slidesharecdn.com/11-170322121329/95/java-16-638.jpg?cb=1508227507)Завершение процесса В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но ещё выполняются порождённые им потоки, система будет ждать их завершения.
17. [17.](https://image.slidesharecdn.com/11-170322121329/95/java-17-638.jpg?cb=1508227507)Демоны Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.
18. [18.](https://image.slidesharecdn.com/11-170322121329/95/java-18-638.jpg?cb=1508227507)Метод Thread.sleep() Thread.sleep() — статический метод класса Thread, который приостанавливает выполнение потока, в котором он был вызван.
19. [19.](https://image.slidesharecdn.com/11-170322121329/95/java-19-638.jpg?cb=1508227507)Метод Thread.sleep() Во время выполнения метода sleep() система перестаёт выделять потоку процессорное время, распределяя его между другими потоками.
20. [20.](https://image.slidesharecdn.com/11-170322121329/95/java-20-638.jpg?cb=1508227507)Определение состояния потока Чтобы определить состояние потока используется метод isAlive().
21. [21.](https://image.slidesharecdn.com/11-170322121329/95/java-21-638.jpg?cb=1508227507)Метод Thread.sleep() Метод sleep() может выполняться либо заданное кол-во времени (миллисекунды или наносекунды) либо до тех пор пока он не будет остановлен прерыванием (в этом случае он сгенерирует исключение InterruptedException).
22. [22.](https://image.slidesharecdn.com/11-170322121329/95/java-22-638.jpg?cb=1508227507)Прерывание работы потока В Java существуют (существовали) средства для принудительного завершения потока. В частности метод Thread.stop() завершает поток незамедлительно после своего выполнения.
23. [23.](https://image.slidesharecdn.com/11-170322121329/95/java-23-638.jpg?cb=1508227507)Прерывание работы потока Однако этот метод, а также Thread.suspend(), приостанавливающий поток, и Thread.resume(), продолжающий выполнение потока, были объявлены устаревшими и их использование отныне крайне нежелательно.
24. [24.](https://image.slidesharecdn.com/11-170322121329/95/java-24-638.jpg?cb=1508227507)Прерывание работы потока Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за своё завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода run(), (main() — для главного потока) либо по сигналу из другого потока.
25. [25.](https://image.slidesharecdn.com/11-170322121329/95/java-25-638.jpg?cb=1508227507)Прерывание работы потока Класс Thread содержит в себе скрытое булево поле, которое называется флагом прерывания. Установить этот флаг можно вызвав метод interrupt() потока. Проверить же, установлен ли этот флаг, можно двумя способами. Первый способ — вызвать метод bool isInterrupted() объекта потока, второй — вызвать статический метод bool Thread.interrupted().
26. [26.](https://image.slidesharecdn.com/11-170322121329/95/java-26-638.jpg?cb=1508227507)Метод yield() Статический метод Thread.yield() заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще.
27. [27.](https://image.slidesharecdn.com/11-170322121329/95/java-27-638.jpg?cb=1508227507)Метод join() В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.
28. [28.](https://image.slidesharecdn.com/11-170322121329/95/java-28-638.jpg?cb=1508227507)Приоритеты потоков Каждый поток в системе имеет свой приоритет. Приоритет – это некоторое число в объекте потока, более высокое значение которого означает больший приоритет. Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда их более привилегированные собратья простаивают.
29. [29.](https://image.slidesharecdn.com/11-170322121329/95/java-29-638.jpg?cb=1508227507)Race condition (англ. race condition) — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. Своё название ошибка получила от похожей ошибки проектирования электронных схем.
30. [30.](https://image.slidesharecdn.com/11-170322121329/95/java-30-638.jpg?cb=1508227507)Конкуренция Конкуренция (contention) — ситуация, когда несколько потоков одновременно пытаются владеть одним и тем же ресурсом, который предназначен для монопольного использования
31. [31.](https://image.slidesharecdn.com/11-170322121329/95/java-31-638.jpg?cb=1508227507)Race condition
32. [32.](https://image.slidesharecdn.com/11-170322121329/95/java-32-638.jpg?cb=1508227507)Race condition
33. [33.](https://image.slidesharecdn.com/11-170322121329/95/java-33-638.jpg?cb=1508227507)Синхронизация Для предотвращения состояния гонки используются приемы синхронизации, позволяющие правильно упорядочить операции, выполняемые разными потоками.
34. [34.](https://image.slidesharecdn.com/11-170322121329/95/java-34-638.jpg?cb=1508227507)Монитор Монитор - это средство обеспечения контроля за доступом к ресурсу.
35. [35.](https://image.slidesharecdn.com/11-170322121329/95/java-35-638.jpg?cb=1508227507)Блокировки В java роль монитора выполняют объекты. У монитора может быть максимум один владелец в каждый текущий момент времени.
36. [36.](https://image.slidesharecdn.com/11-170322121329/95/java-36-638.jpg?cb=1508227507)Монитор Следовательно, если кто-то использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой, желающий использовать тот же ресурс, должен подождать освобождения монитора, захватить его и только потом начать использовать ресурс.
37. [37.](https://image.slidesharecdn.com/11-170322121329/95/java-37-638.jpg?cb=1508227507)Монитор Если поток использует ресурс и захватил монитор для обеспечения единоличного доступа, то другой поток, желающий использовать тот же ресурс, должен подождать освобождения монитора.
38. [38.](https://image.slidesharecdn.com/11-170322121329/95/java-38-638.jpg?cb=1508227507)Deadlock Предположим, что поток один уже захватил монитор на некотором объекте A и для продолжения работы ему нужно захватить монитор на объекте X.В втором же потоке ситуация ровно обратная – он уже захватил монитор на объекте A и ему нужен монитор объекта B.
39. [39.](https://image.slidesharecdn.com/11-170322121329/95/java-39-638.jpg?cb=1508227507)Взаимные блокировки В результате оба потока будут ждать, пока нужный монитор освободится. Как вы сами прекрасно понимаете, ждать они будут до бесконечности. Эта ситуация и называется взаимной блокировкой – deadlock. Способов определить случившуюся взаимную блокировку в Java нет. Виртуальная машина никак не отслеживает подобных ситуаций.
40. [40.](https://image.slidesharecdn.com/11-170322121329/95/java-40-638.jpg?cb=1508227507)Livelock Система не «застревает» (как в обычной взаимной блокировке), а занимается бесполезной работой, её состояние постоянно меняется — но, тем не менее, она «зациклилась», не производит никакой полезной работы. Жизненный пример: двое встречаются лицом к лицу. Каждый пытается посторониться, но оба сдвигаются в одну и ту же сторону и так бесконечно.
41. [41.](https://image.slidesharecdn.com/11-170322121329/95/java-41-638.jpg?cb=1508227507)Livelock Метод пытается выполнить какую-либо работу, используя 2 внешних объекта. Сперва он получает блокировку по одному из объектов, а затем проверяет, свободен ли второй объект. Если объект свободен - получает блокировку по нему и выполняет работу, если занят - освобождает первый объект и ждёт, когда они оба освободятся.
42. [42.](https://image.slidesharecdn.com/11-170322121329/95/java-42-638.jpg?cb=1508227507)Livelock 2 потока одновременно вызывают этот метод. Поток 1 блокирует первый объект. Поток 2 блокирует второй объект. Оба проверяют, свободен ли второй ресурс - обнаруживают, что он занят и освобождают занятый ресурс. Оба потока обнаруживают, что оба ресурса свободны и начинают процесс блокировки сначала.
43. [43.](https://image.slidesharecdn.com/11-170322121329/95/java-43-638.jpg?cb=1508227507)Livelock
44. [44.](https://image.slidesharecdn.com/11-170322121329/95/java-44-638.jpg?cb=1508227507)Starvation Множество потоков постоянно находятся в борьбе за один критический ресурс и все ждут, пока кто-то один освободит этот ресурс.
45. [45.](https://image.slidesharecdn.com/11-170322121329/95/java-45-638.jpg?cb=1508227507)Starvation Потом из ждущих по какому-либо алгоритму выбирается только кто-то один, кто следующим захватит ресурс. Таким образом, может найтись один такой поток, который никогда не получит доступ к ресурсу, потому что другие потоки постоянно захватывают ресурс раньше него.
46. [46.](https://image.slidesharecdn.com/11-170322121329/95/java-46-638.jpg?cb=1508227507)Синхронизация данных Одной из проблем при работе с данными из разных потоков является их синхронизация. И даже не столько в случае одновременного изменения. Чтение данных в одном потоке после изменения их другим потоком тоже может давать неверный результат.
47. [47.](https://image.slidesharecdn.com/11-170322121329/95/java-47-638.jpg?cb=1508227507)Синхронизация данных Происходит это потому, что поток держит у себя локальные копии данных, которые использует для чтения. И обновление их происходит в тот момент времени, который сочтёт для этого удобным виртуальная машина. Это происходит если мы не предпринимаем никаких усилий по синхронизации.
48. [48.](https://image.slidesharecdn.com/11-170322121329/95/java-48-638.jpg?cb=1508227507)synchronized-блок Object sync = new Object(); ... synchronized(sync){ ... }
49. [49.](https://image.slidesharecdn.com/11-170322121329/95/java-49-638.jpg?cb=1508227507)Synchronized-метод (не статик) public synchronized void someMethod(){ // code } эквивалентно public void someMethod(){ synchronized(this){ // code } }
50. [50.](https://image.slidesharecdn.com/11-170322121329/95/java-50-638.jpg?cb=1508227507)Synchronized-static-метод public class SomeClass{ public static synchronized void someMethod(){ //code } } public class SomeClass{ public static void someMethod(){ synchronized(SomeClass.class){ //code } }
51. [51.](https://image.slidesharecdn.com/11-170322121329/95/java-51-638.jpg?cb=1508227507)Synchronized Если два не статических метода объявлены как synchronized, то в каждый момент времени из разных потоков на одном объекте может быть вызван только один из них. Поток, который вызывает метод первым, захватит монитор, и второму потоку придется ждать.
52. [52.](https://image.slidesharecdn.com/11-170322121329/95/java-52-638.jpg?cb=1508227507)Synchronized Синхронизация на уровне метода имеет очевидный недостаток – метод может быть долгим. Если, скажем, нужно менять данные в некой реализации Runnable и при этом синхронизировать метод run – ничего хорошего не получится. Ибо run захватит монитор при старте и любой вызов любого синхронизированного метода из любого другого потока будет блокирован.
53. [53.](https://image.slidesharecdn.com/11-170322121329/95/java-53-638.jpg?cb=1508227507)Synchronized У синхронизированного метода ключевое слово synchronized фигурирует в сигнатуре, что сразу дает разработчику много информации о поведении данного метода. Если же синхронизация будет внутри, блоком от начала до конца метода – о том, что она есть, нужно будет упоминать в комментариях к методу.
54. [54.](https://image.slidesharecdn.com/11-170322121329/95/java-54-638.jpg?cb=1508227507)Ключевое слово volatile Поле объекта можно объявить с модификатором volatile. Этот модификатор вынуждает потоки отключить оптимизацию доступа и использовать единственный экземпляр переменной (не кэшировать).
55. [55.](https://image.slidesharecdn.com/11-170322121329/95/java-55-638.jpg?cb=1508227507)volatile Операции чтения/записи volatile переменной не являются атомарными. Результат операции записи значения в volatile переменную одним потоком, становится виден всем другим потокам, которые используют эту переменную для чтения из нее значения.
56. [56.](https://image.slidesharecdn.com/11-170322121329/95/java-56-638.jpg?cb=1508227507)Синхронизация данных Если переменная является ссылкой на объект – синхронизировано будет исключительно значение этой ссылки.
57. [57.](https://image.slidesharecdn.com/11-170322121329/95/java-57-638.jpg?cb=1508227507)Синхронизация данных JRE неявно обеспечивает синхронизацию при доступе к volatile-переменным, но : чтение volatile-переменной и запись в volatile-переменную синхронизированы, а неатомарные операции ― нет
58. [58.](https://image.slidesharecdn.com/11-170322121329/95/java-58-638.jpg?cb=1508227507)Синхронизация данных Синхронизация – это всегда некоторые дополнительные издержки. При большой интенсивности использования синхронизированных методов исключительно для доступа к данным эти издержки могут дать отрицательный эффект и вызвать желание избавиться от синхронизации.
59. [59.](https://image.slidesharecdn.com/11-170322121329/95/java-59-638.jpg?cb=1508227507)Приёмы синхронизации По уровню, на котором производится синхронизация потоков, ее можно разделить на две области – системная и программная. Системная осуществляется с использованием средств (мониторов и пр.) виртуальной машины, в то время как в программной роль объектов, на которых происходит синхронизация, играют Java-объекты.
60. [60.](https://image.slidesharecdn.com/11-170322121329/95/java-60-638.jpg?cb=1508227507)Системная синхронизация с использованием wait/notify Берётся некий объект. Поток, который ждёт выполнения каких-либо условий, вызывает у этого объекта метод wait, предварительно захватив его монитор. На этом его работа приостанавливается.
61. [61.](https://image.slidesharecdn.com/11-170322121329/95/java-61-638.jpg?cb=1508227507)Системная синхронизация с использованием wait/notify Другой поток может вызвать на этом же самом объекте метод notify (опять же, предварительно захватив монитор объекта), в результате чего, ждущий на объекте поток "просыпается" и продолжает свое выполнение.
62. [62.](https://image.slidesharecdn.com/11-170322121329/95/java-62-638.jpg?cb=1508227507)notifyAll notifyAll служит для одной цели – собщает ВСЕМ ждущим потокам, что они могут продолжить работу (очередность пробуждения зависит от ОС). В то время как вызов notify подействует только на один.
63. [63.](https://image.slidesharecdn.com/11-170322121329/95/java-63-638.jpg?cb=1508227507)thread-safe Объект является потоко- безопасным (thread-safe), если его методы могут вызываться из различных потоков без ущерба для его состояния. Никаких усилий по синхронизации при этом прилагать не требуется, об этом заботится сам объект.
64. [64.](https://image.slidesharecdn.com/11-170322121329/95/java-64-638.jpg?cb=1508227507)Программная синхронизация – шаблоны и библиотеки Вся программная синхронизация неразрывно связана с одним единственным именем – Doug Lea (Дуг Ли). В 1999 году вышла его книга Concurrent Programming in Java™: Design Principles and Pattern, в которой собрано множество различных шаблонов программной синхронизации.
65. [65.](https://image.slidesharecdn.com/11-170322121329/95/java-65-638.jpg?cb=1508227507)Программная синхронизация – шаблоны и библиотеки Реализации синхронизации оказались настолько удачными, что с версии Java 1.5 они включены в ядро java в виде пакетов: java.util.concurrent, java.util.concurrent.atomic и java.util.concurrent.locks.
66. [66.](https://image.slidesharecdn.com/11-170322121329/95/java-66-638.jpg?cb=1508227507)java.util.concurent
67. [67.](https://image.slidesharecdn.com/11-170322121329/95/java-67-638.jpg?cb=1508227507)Атомарные типы Atomics Atomics — классы с поддержкой атомарных операций над примитивами и ссылками. AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray За счёт использования CAS, операции с этими классами работают быстрее, чем если синхронизироваться через synchronized/volatile.
68. [68.](https://image.slidesharecdn.com/11-170322121329/95/java-68-638.jpg?cb=1508227507)Атомарные типы Atomics AtomicReference — Класс для атомарных операцией с ссылкой на объект. AtomicMarkableReference — Класс для атомарных операцией со следующей парой полей: ссылка на объект и битовый флаг (true/false).
69. [69.](https://image.slidesharecdn.com/11-170322121329/95/java-69-638.jpg?cb=1508227507)Атомарные типы Atomics AtomicStampedReference — Класс для атомарных операцией со следующей парой полей: ссылка на объект и int значение. AtomicReferenceArray — Массив ссылок на объекты, который может атомарно обновляться.
70. [70.](https://image.slidesharecdn.com/11-170322121329/95/java-70-638.jpg?cb=1508227507)Атомарные типы Atomics AtomicIntegerFieldUpdater, AtomicLongFieldUpdater,AtomicReference FieldUpdater — 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.
71. [71.](https://image.slidesharecdn.com/11-170322121329/95/java-71-638.jpg?cb=1508227507)Concurrent Collections Concurrent collections – это набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета.
72. [72.](https://image.slidesharecdn.com/11-170322121329/95/java-72-638.jpg?cb=1508227507)Concurrent Collections Вместо базовой обертки 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.
73. [73.](https://image.slidesharecdn.com/11-170322121329/95/java-73-638.jpg?cb=1508227507)Concurrent Collections
74. [74.](https://image.slidesharecdn.com/11-170322121329/95/java-74-638.jpg?cb=1508227507)CopyOnWrite коллекции Перед каждой модификацией коллекция копирует свое содержимое в новый массив, чтобы операции чтения содержимого массива выполнялись без синхронизации (так как они никогда не работают с изменяемыми данными).
75. [75.](https://image.slidesharecdn.com/11-170322121329/95/java-75-638.jpg?cb=1508227507)CopyOnWrite коллекции CopyOnWriteArrayList<E> — Потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом. CopyOnWriteArraySet<E> — Имплементация интерфейса Set, использующая за основу CopyOnWriteArrayList. В отличии от CopyOnWriteArrayList, дополнительных методов нет.
76. [76.](https://image.slidesharecdn.com/11-170322121329/95/java-76-638.jpg?cb=1508227507)Blocking Queues BlockingQueue (блокирующая очередь) - это очередь, которая блокирует поток, в двух случаях: •поток пытается получить элементы из пустой очереди •поток пытается положить элементы в полную очередь
77. [77.](https://image.slidesharecdn.com/11-170322121329/95/java-77-638.jpg?cb=1508227507)Blocking Queues
78. [78.](https://image.slidesharecdn.com/11-170322121329/95/java-78-638.jpg?cb=1508227507)Blocking Queues ArrayBlockingQueue<E> — 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.
79. [79.](https://image.slidesharecdn.com/11-170322121329/95/java-79-638.jpg?cb=1508227507)Blocking Queues SynchronousQueue<E> — 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.
80. [80.](https://image.slidesharecdn.com/11-170322121329/95/java-80-638.jpg?cb=1508227507)Blocking Queues DelayQueue<E extends Delayed> — Довольно специфичный класс, который позволяет вытаскивать элементы из очереди только по прошествии некоторой задержки, определённой в каждом элементе через метод getDelay интерфейса Delayed.
81. [81.](https://image.slidesharecdn.com/11-170322121329/95/java-81-638.jpg?cb=1508227507)Blocking Queues LinkedBlockingQueue<E> — 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.
82. [82.](https://image.slidesharecdn.com/11-170322121329/95/java-82-638.jpg?cb=1508227507)Blocking Queues PriorityBlockingQueue<E> — Является многопоточной оберткой над PriorityQueue. При вставке 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.
83. [83.](https://image.slidesharecdn.com/11-170322121329/95/java-83-638.jpg?cb=1508227507)Non-Blocking Queues ConcurrentLinkedQueue<E> — 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. Метод size() может работать долго, поэтому лучше постоянно его не дергать.
84. [84.](https://image.slidesharecdn.com/11-170322121329/95/java-84-638.jpg?cb=1508227507)Non-Blocking Queues ConcurrentLinkedDeque<E> — Данные в этой очереди можно добавлять и получать 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.
85. [85.](https://image.slidesharecdn.com/11-170322121329/95/java-85-638.jpg?cb=1508227507)Concurrent Collections
86. [86.](https://image.slidesharecdn.com/11-170322121329/95/java-86-638.jpg?cb=1508227507)Scalable Maps ConcurrentMap<K, V> — Интерфейс, расширяющий Map несколькими дополнительными атомарными операциями. 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 доступ к данным закрывается по сегментам, а не по одному объекту.
87. [87.](https://image.slidesharecdn.com/11-170322121329/95/java-87-638.jpg?cb=1508227507)Scalable Maps ConcurrentSkipListMap<K, V> — 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. ConcurrentSkipListSet<E> — Имплементация Set интерфейса, выполненная на основе ConcurrentSkipListMap.
88. [88.](https://image.slidesharecdn.com/11-170322121329/95/java-88-638.jpg?cb=1508227507)Синхронизаторы Синхронизаторы – вспомогательные утилиты для синхронизации потоков, которые дают возможность разработчику регулировать и/или ограничивать работу потоков и предоставляют более высокий уровень абстракции, чем основные примитивы языка (мониторы).
89. [89.](https://image.slidesharecdn.com/11-170322121329/95/java-89-638.jpg?cb=1508227507)Semaphore Семафоры (Semaphore) чаще всего используются для ограничения количества потоков при работе с аппаратными ресурсами или файловой системой. Доступ к общему ресурсу управляется с помощью счётчика
90. [90.](https://image.slidesharecdn.com/11-170322121329/95/java-90-638.jpg?cb=1508227507)Semaphore
91. [91.](https://image.slidesharecdn.com/11-170322121329/95/java-91-638.jpg?cb=1508227507)CountDownLatch Защёлки — позволяют одному или нескольким потокам ожидать до тех пор, пока не завершится определённое количество операций, выполняющих в других потоках.
92. [92.](https://image.slidesharecdn.com/11-170322121329/95/java-92-638.jpg?cb=1508227507)CountDownLatch
93. [93.](https://image.slidesharecdn.com/11-170322121329/95/java-93-638.jpg?cb=1508227507)CountDownLatch Как только счётчик становится равным нулю, все ожидающие потоки в await продолжат свою работу, а все последующие вызовы await будут проходить без ожиданий. Счётчик count down одноразовый и не может быть сброшен в первоначальное состояние.
94. [94.](https://image.slidesharecdn.com/11-170322121329/95/java-94-638.jpg?cb=1508227507)Циклический барьер Циклический барьер — может использоваться для синхронизации заданного количества потоков в одной точке. Барьер достигается когда N- потоков вызовут метод await(...) и заблокируются. После чего счётчик сбрасывается в исходное значение, а ожидающие потоки освобождаются.
95. [95.](https://image.slidesharecdn.com/11-170322121329/95/java-95-638.jpg?cb=1508227507)CyclicBarrier Дополнительно, если нужно, существует возможность запуска специального кода до разблокировки потоков и сброса счетчика. Для этого через конструктор передается объект с реализацией Runnable интерфейса.
96. [96.](https://image.slidesharecdn.com/11-170322121329/95/java-96-638.jpg?cb=1508227507)CyclicBarrier
97. [97.](https://image.slidesharecdn.com/11-170322121329/95/java-97-638.jpg?cb=1508227507)Обменник Exchanger<V> — 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.
98. [98.](https://image.slidesharecdn.com/11-170322121329/95/java-98-638.jpg?cb=1508227507)Exchanger Exchanger<V> — 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.
99. [99.](https://image.slidesharecdn.com/11-170322121329/95/java-99-638.jpg?cb=1508227507)Phaser Phaser — 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 пере использоваться и сообщать о готовности потока без его блокировки.
100. [100.](https://image.slidesharecdn.com/11-170322121329/95/java-100-638.jpg?cb=1508227507)Phaser
101. [101.](https://image.slidesharecdn.com/11-170322121329/95/java-101-638.jpg?cb=1508227507)ExecutorService ExecutorService - исполняет асинхронный код в одном или нескольких потоках. Создание объекта ExecutorService'а делается либо вручную через конкретные имплементации (ScheduledThreadPoolExecutor или ThreadPoolExecutor) или используя фабрики класса Executors. ExecutorService service = Executors.newFixedThreadPool(2);
102. [102.](https://image.slidesharecdn.com/11-170322121329/95/java-102-638.jpg?cb=1508227507)Thread pool