

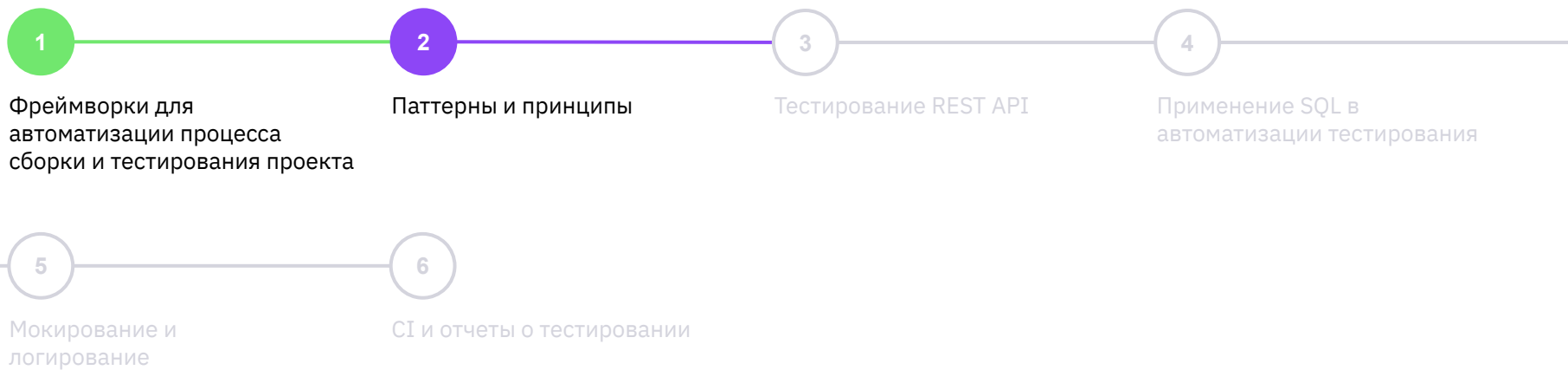
# Java Development Kit

Урок 5

Многопоточность








## План курса





## Чему посвящен урок

-  Принципы разработки многопоточных приложений
-  Java memory model
-  Процессы и потоки
-  Примитивы синхронизации
-  Коллекции и многопоточность





## Процессы



Процесс состоит из кода и данных.



Процессы работают независимо друг от друга. Они не имеют прямого доступа к общим данным в других процессах.



Операционная система выделяет ресурсы для процесса — память и время на выполнение.



Если один из процессов заблокирован, то ни один другой процесс не может выполняться, пока он не будет разблокирован.



Процесс может контролировать дочерние процессы, но не процессы того же уровня.



# Многопоточность





Два приложения работающие под управлением одной операционной системы — это два независимых процесса.

Процесс состоит из потоков. Потоки могут выполняться параллельно друг с другом





## Многопоточность

— это одновременное выполнение двух или более потоков для максимального использования центрального процессора





## Применение многопоточности

- 💡 Эффективное использование одного центрального процессора
- 💡 Оптимальное использование нескольких центральных процессоров или их ядер
- 💡 Улучшенный user experience в плане скорости ответа на запрос
- 💡 Улучшенный user experience в плане справедливости распределения ресурсов





## Статусы потоков



**New** — экземпляр потока создан, но он еще не работает.



**Running** — поток запущен и процессор начинает его выполнение.



**Suspended** — запущенный поток приостанавливает свою работу, затем можно возобновить его выполнение.



**Blocked** — поток ожидает высвобождения ресурсов или завершение операции ввода-вывода.



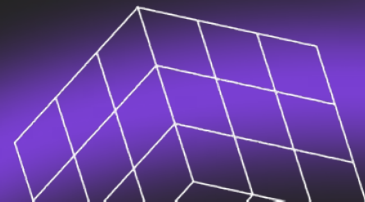
**Terminated** — поток немедленно завершает свое выполнение.



**Dead** — после того, как поток завершил свое выполнение, его состояние меняется на dead, то есть он завершает свой жизненный цикл.



Для создания потоков вы можете имплементировать интерфейс Runnable или использовать подкласс Thread





## Создание и запуск потоков

```
1 public static void main(String[] args) {
2     //Мы всегда можем получить текущий поток выполнения
3     System.out.println(Thread.currentThread().getName());
4     Runnable task = new Runnable() {
5         public void run() {
6             System.out.println(Thread.currentThread().getName());
7             System.out.println("Make some work!");
8         }
9     };
10    Thread thread = new Thread(task);
11    thread.start();
12
13 }
```



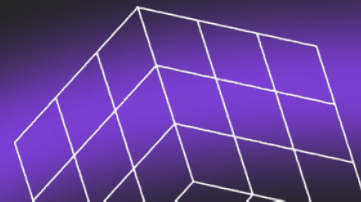
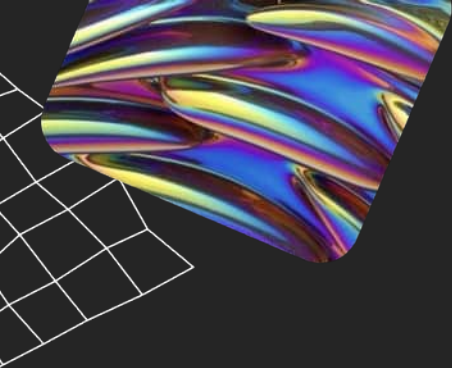
## Создание и запуск потоков

```
1 public static void main(String[] args) {
2     //Мы всегда можем получить текущий поток выполнения
3     System.out.println(Thread.currentThread().getName());
4     ExampleThread exampleThread = new ExampleThread();
5     exampleThread.start();
6
7 }
8
9 public static class ExampleThread extends Thread {
10     @Override
11     public void run() {
12         System.out.println(Thread.currentThread().getName());
13         System.out.println("Make some work");
14     }
15 }
```



Для остановки потоков вы можете  
использовать метод `stop` (не рекомендуется)  
или использовать метод `interrupt`.

Поток может остановиться сам  
с использованием метода `sleep`





## Остановка потока — interrupt()

```
1 public static void main(String[] args) {  
2     Runnable task = () → {  
3         try {  
4             Thread.sleep(10000);  
5         } catch (InterruptedException e) {  
6             System.out.println("Interrupted");  
7         }  
8     };  
9     Thread thread = new Thread(task);  
10    thread.start();  
11    thread.interrupt();  
12 }
```



## Остановка потока — interrupt()

```
1 public static void main(String []args) {  
2     Runnable task = () → {  
3         while(!Thread.currentThread().isInterrupted()) {  
4  
5             }  
6         System.out.println("Finished");  
7     };  
8     Thread thread = new Thread(task);  
9     thread.start();  
10    thread.interrupt();  
11 }
```



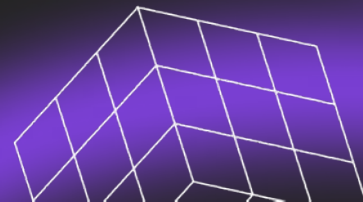
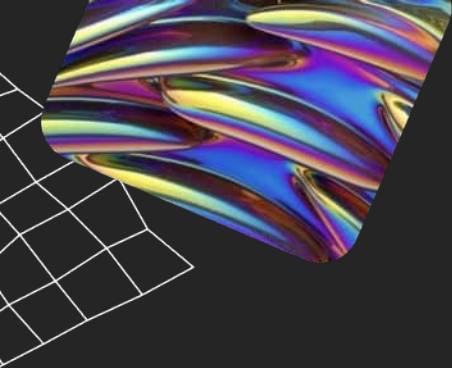
## Остановка потока — sleep()

```
1 public static void main(String[] args) {  
2     Runnable task = () → {  
3         try {  
4             Thread.currentThread().sleep(10000);  
5             System.out.println("Waked up");  
6         } catch (InterruptedException e) {  
7             e.printStackTrace();  
8         }  
9     };  
10    Thread thread = new Thread(task);  
11    thread.start();  
12    System.out.println("you here");  
13 }
```





Для управления очередностью выполнения  
потоков используется метод `join`



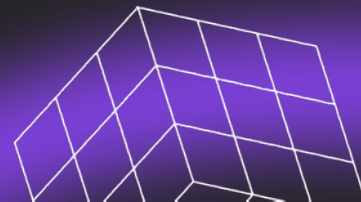
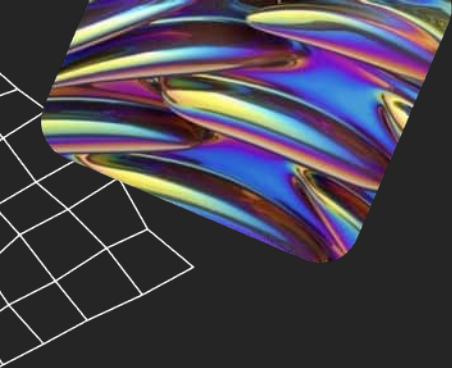


## Очередность выполнения потоков

```
1 public static void main(String []args) throws InterruptedException {  
2     Runnable task = () → {  
3         try {  
4             Thread.sleep(10000);  
5             System.out.println("work done");  
6         } catch (InterruptedException e) {  
7             System.out.println("Interrupted");  
8         }  
9     };  
10    Thread thread = new Thread(task);  
11    thread.start();  
12    thread.join();  
13    System.out.println("Finished");  
14 }
```



Процесс создания новых потоков и освобождение ресурсов являются дорогостоящей операцией. Мы можем изначально определить необходимое количество потоков, создать их и использовать для решения разных задач





## Пул потоков



Интерфейс Executor.



ExecutorService.



Класс Executors.

## Реализации пул потоков



`ThreadPoolExecutor`



`Executors.newCachedThreadPool`



`Executors.newSingleThreadExecutor`



`ScheduledThreadPoolExecutor`





# Синхронизация



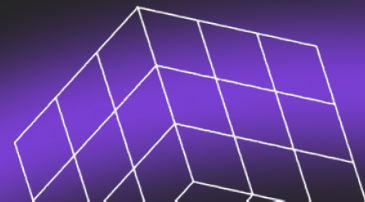


## Одновременное обращение к одному объекту

```
1 static Integer object = Integer.valueOf(0);
2
3 public static void main(String []args) throws
  InterruptedException {
4     Runnable task = () -> {
5         object = object + 1;
6         System.out.println(Thread.currentThread().getName());
7     };
8     Thread thread = new Thread(task);
9     thread.start();
10    System.out.println(Thread.currentThread().getName());
11    object = object + 1;
12    System.out.println(object.intValue());
13 }
```



Самым простым способом синхронизировать потоки (т.е. определить их поведение при работе с общим ресурсом) — это концепция «монитора» и ключевое слово `synchronized`







# Монитор

- 💡 У любого не примитивного типа есть монитор
- 💡 состоянии (locked) — признак определяющий захвачен ли монитор каким либо потоком)
- 💡 какой поток в текущий момент захватил монитор (owner);
- 💡 перечень потоков, которые не смогли захватить монитор (blocked set);
- 💡 перечень потоков у которых был вызван метод wait (wait set).



## Синхронизация

- 💡 Поток попадает в synchronized блок;
- 💡 Проверяются переменные locked и owner.
- 💡 Если эти поля false и null, соответственно, они заполняются.  
Если переменная owner не равна потоку, который хочет захватить монитор, то поток блокируется и попадает в blocked set монитора
- 💡 Поток выполнил код, который соответствует открывающей фигурной скобке synchronized блока
- 💡 После выполнил код, который соответствует закрывающейся фигурной скобке блока синхронизации,
- 💡 Переменные locked и owner в мониторе очищаются



## Синхронизация методов класса

```
1 // используется монитор объекта this
2 public synchronized void doSomething() {
3
4     // ... реализация бизнес логики метода
5 }
```



## Синхронизация другого объекта в классе

```
1 public static void main(String []args) throws InterruptedException {
2     Object objectToLock = new Object();
3
4     Runnable task = () -> {
5         synchronized (objectToLock) {
6             System.out.println(Thread.currentThread().getName());
7         }
8     };
9
10    Thread thread = new Thread(task);
11    thread.start();
12    //Если необходимо, что task выполнялся раньше используем метод
13    join()
14    //thread.join();
15    synchronized (objectToLock) {
16        for (int i = 0; i < 10; i++) {
17            Thread.currentThread().sleep(1000);
18            System.out.println("step" + i);
19        }
20        System.out.println(Thread.currentThread().getName());
21    }
```

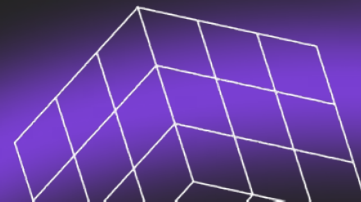
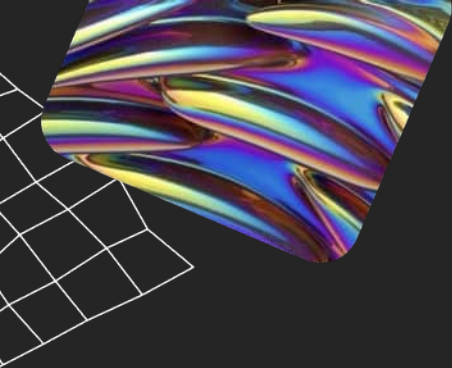


## Синхронизация статических методов класса

```
1 public static void doWork() {  
2  
3     synchronized (MyClass.class) {  
4         // ... Реализация логики  
5     }  
6 }
```



# Синхронизация и переключение между потоками ресурсоемкая операция





## Deadlock или взаимная блокировка

— это ошибка, которая происходит когда threads имеют циклическую зависимость от пары синхронизированных объектов.





# Deadlock

```
1 public static void main(String[] args) {
2     ObjectToLock objectToLockOne = new ObjectToLock();
3     ObjectToLock objectToLockTwo = new ObjectToLock();
4     getThread(objectToLockTwo, objectToLockOne);
5     getThread(objectToLockOne, objectToLockTwo);
6 }
7
8 private static void getThread(ObjectToLock objectToLockOne,
9 ObjectToLock objectToLockTwo) {
10     new Thread(new Runnable() {
11         @Override
12         public void run() {
13             System.out.println("run:" +
14 Thread.currentThread().getName());
15             try {
16                 Thread.sleep(100);
17             } catch (InterruptedException e) {
18                 e.printStackTrace();
19             }
20             objectToLockTwo.stepOne(objectToLockOne);
21         }
22     }).start();
23 }
24
25 static class ObjectToLock {
26     public synchronized void stepOne(ObjectToLock object) {
27         System.out.println("stepOne:" +
28 Thread.currentThread().getName());
29         object.stepTwo(this);
30     }
31     public synchronized void stepTwo(ObjectToLock object) {
32         System.out.println("stepTwo:" +
33 Thread.currentThread().getName());
34         object.toString();
35     }
36 }
```





## Управление переключение потоков

💡 `wait()`

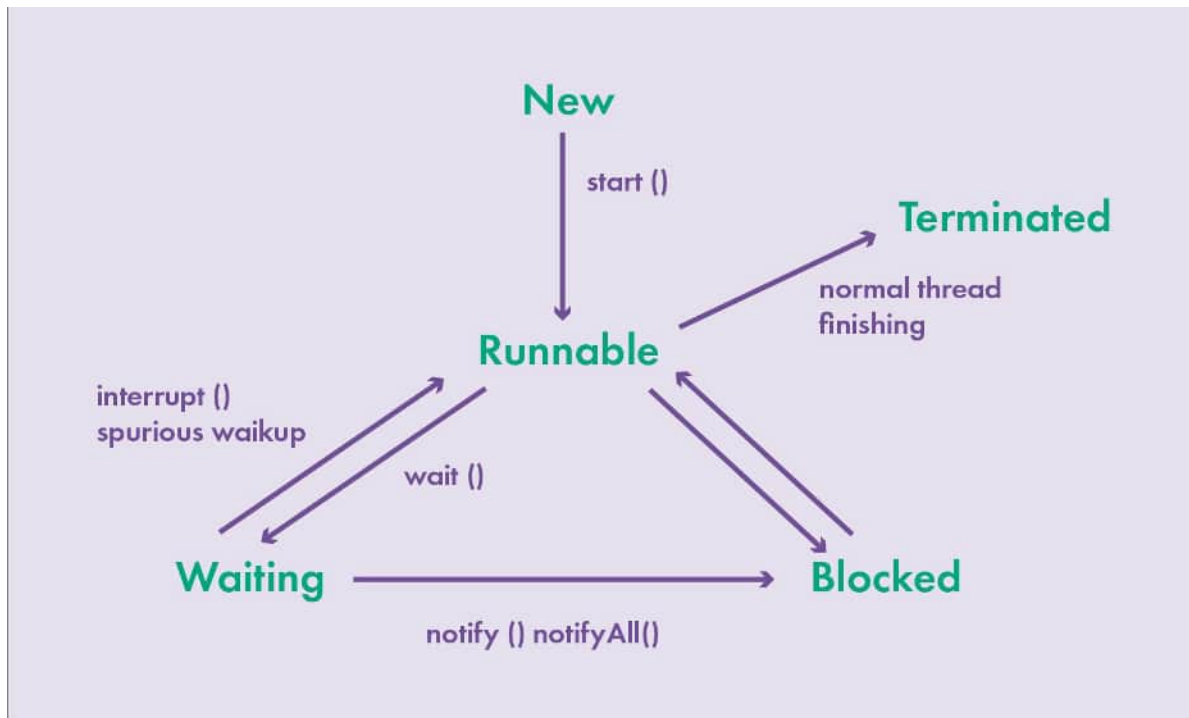
💡 `notify()`

💡 `notifyAll()`





## Управление переключение потоков



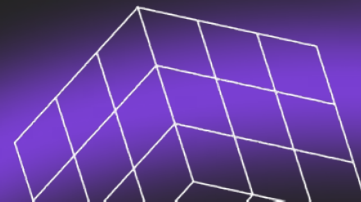


# Java memory model





Модель памяти Java (Java Memory Model, JMM) описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а для Java в целом





## Типы памяти



Heap – это регион памяти, где хранятся объекты Java.



Stack – это область памяти, где хранятся локальные переменные и стек вызовов методов.



Method Area – это область памяти, где хранятся информация о классах и методах JVM. Здесь также хранятся константы и статические переменные.



Program Counter Register – это регистр, который указывает на следующую инструкцию, которую нужно выполнить в текущем потоке.



Native Method Stack – это стек, используемый для выполнения нативного кода.



## Проблема

— видимость изменений, которые произвел  
поток над общими переменными





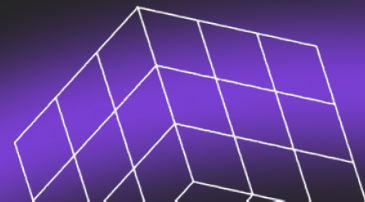
## Проблема

— состояние гонки при чтении, проверке  
и записи общих переменных





Ключевое слово `volatile` указывает, что взаимодействие с переменной в памяти должно происходить минуя кэши процессора, т. е. напрямую.







## Volatile гарантирует



Если первый поток записывает в `volatile` переменную, а затем второй поток читает значение из этой переменной, тогда все переменные, видимые потоку A перед записью в переменную `volatile`, также будут видны потоку B, после того как он прочитал переменную `volatile`.



Если поток A считывает переменную `volatile`, то все переменные, видимые потоку A при чтении переменной `volatile`, также будут перечитаны из основной памяти.



В Java нам доступны классы атомарных переменных:  
`AtomicInteger` `AtomicLong`, `AtomicBoolean`, `AtomicReference`.





## Алгоритм создания immutable объектов



Проверьте отсутствие mutable методов — т.е. классе не должно быть ни одного публичного метода, который могут бы изменить состояние объекта.



Все поля следует объявить как `private final` — это гарантирует, что если поле ссылается на примитивный тип оно никогда не измениться, если на ссылочный тип то ссылка не может быть изменена.



Если метод вашего класса возвращает изменяемый объект, то возвращайте его копию.



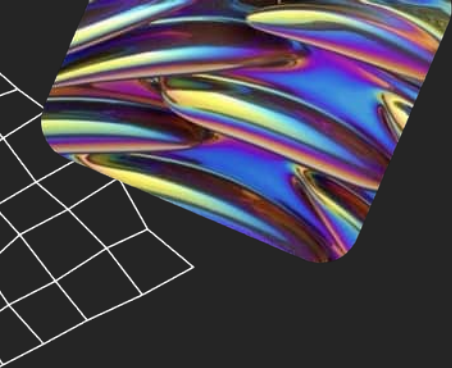
Если при вызове конструктора вы передаете в него изменяемый объект, который должен быть присвоен полю, то создавайте его копию.



Объявите ваш класс `final`



Класс ThreadLocal представляет  
хранилище тред-локальных  
переменных.





# ThreadLocal

```
1 private static Integer counter = 0;
2
3 public static void main(String []args) throws InterruptedException
4 {
5     new Thread(new ThreadTask()).start();
6     new Thread(new ThreadTask()).start();
7 }
8
9 public static class ThreadTask implements Runnable {
10     ThreadLocal<Integer> threadCounter = new ThreadLocal<>();
11     public void run() {
12         for (int i=0; i < 10; i++) {
13             counter++;
14             if (threadCounter.get() != null) {
15                 threadCounter.set(threadCounter.get() + 1);
16             } else {
17                 threadCounter.set(0);
18             }
19         }
20         System.out.println("Counter: " + counter);
21         System.out.println("threadLocalCounter: " +
22             threadCounter.get());
23     }
24 }
```

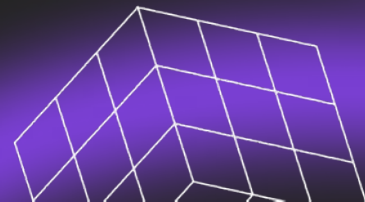


# Примитивы синхронизации





Semaphore один из примитивов синхронизации,  
позволяющий определить  $N$  потоков, которым позволено  
исполнять критическую секцию кода.





## Exchanger

— точка синхронизации, позволяющая двум  
потокам обмениваться значениями







## CountDownLatch

— это счетчик значение которого уменьшается  
каждый раз, когда поток использует счетчик  
(поток при этом блокируется)





## CyclicBarrier

— аналог CountdownLatch, но можно  
использовать повторно





## ReentrantLock

— высокоуровневые механизмы блокировки, как более удобная альтернатива `synchronized`





## Интерфейс Lock

💡 **void lock()** — захват блокировки (если доступна).

💡 **void lockInterruptibly()** — аналог `lock()`, но позволяет прервать блокированный поток и восстановить выполнение через `InterruptedException`;

💡 **boolean tryLock()** — неблокирующий вариант метода `lock()`

💡 **boolean tryLock(long timeout, TimeUnit timeUnit)** — то же, что `tryLock()`, за исключением того, что метод ждет определенное время, перед тем остановить попытку получения блокировки;

💡 **void unlock()** — отпускает блокировку.



# Коллекции





Можно превратить в синхронизированную  
обычную коллекцию, вызвав соответствующий  
ее типу метод `Collections.synchronized*`()





## CopyOnWriteArrayList



Copy on write содержит volatile массив элементов и при КАЖДОМ изменении коллекции создается новая локальная копия массива для изменений. После модификации измененная такая копия становится текущей.



Массив элементов используется с ключевым словом volatile для того, чтобы все потоки сразу увидели изменения в массиве.



Если для чтения коллекции используется Iterator и при попытке вызвать remove() при обходе коллекции будет сгенерировано UnsupportedOperationException



Для операций записи внутри класса CopyOnWriteArrayList создается блокировка, чтобы в конкретный момент времени только один поток мог изменять copy on write структуру данных.



`CopyOnWriteArraySet` используется `CopyOnWriteArrayList`,  
и для него характерны те же свойства.

Если необходимо иметь отсортированное множество элементов,  
следует использовать `ConcurrentSkipListSet`, который реализует  
интерфейсы `SortedSet` и `NavigableSet`.







## ConcurrentHashMap

— содержит массив бакетов, внутри каждого из которых находится либо связанный список, либо бинарное дерево. Внутри ConcurrentHashMap состоит из массива сегментов, каждый из которых содержит отдельную HashMap с массивом бакетов





## BlockingQueue

— интерфейс потокобезопасных очередей,  
в которую несколько потоков могут записывать  
данные и читать их оттуда.





## Подведем итоги

- Научились создавать и запускать потоки
- Определили принципы и механизмы разработки многопоточных приложений
- Научились создавать immutable объекты
- Научились работать с примитивами синхронизации и синхронизированными коллекция
- Изучили принципы работы Java с памятью



# Чек-лист подготовки к семинару

Перед семинаром рекомендуем вам:





**Спасибо за внимание**

