Лекция 6

Взаимоблокировка потоков

Если имеются два потока и два объекта, подлежащих блокированию, возникает опасность возникновения взаимоблокировки — каждый из потоков владеет блокировкой одного объекта и ожидает освобождения другого объекта.

Если объект X обладает synchronized-методом, который вызывает synchronized-метод объекта Y, а Y, в свою очередь, также имеет синхронизированный метод, обращающийся к synchronized-методу объекта X, два потока могут находиться в состоянии ожидания взаимного завершения, чтобы овладеть блокировкой, и ни один из них не окажется способен продолжить работу.

Такая ситуация называется клинчем.

Рассмотрим пример:

```
class Friends {
  private Friends partner;
  private String name;
  public Friends(String name){ this.name = name; }
  public synchronized void hug(){
    Systern.out.println(Thread.currentThread().getName()+
        "в" + name + ".hug() пытается  вызвать" + partner.name + ".hugBack()") ;
    partner.hugBack();
  }
  private synchronized void hugBack(){
   System.out.println(Thread.currentThread().getName()+ " в  " + name +
                                                          ".hugBack()");
   }

  public void becomeFriend(Friends partner)  {
        this.partner = partner;
  }
}
```

Далее возможен следующий сценарий:

```
public static void main(string[]  args)  {
   final Friends jareth = new Friends("jareth");
   final Friends cory = new Friends("cory");
   jareth.becomeFriend(cory);
   cory.becomeFriend(jareth);
    new Thread(new Runnable(){
      public void run(){ jareth.hug();}
     },“thread1”).start();
    new Thread(new Runnable(){
      public void run(){ cory.hug();}
    },“thread2”).start();
}
```

Таким образом имеется следующий сценарий:

1) Thread 1 вызывает synchronized метод **jareth.hug();** теперь thread 1 владеет блокировкой объекта jareth.
2) Thread 2 вызывает synchronized-метод **cory.hug();** теперь thread 2 владеет блокировкой объекта cory;
3) **jareth.hug()** вызывает synchronized-метод **cory.hugBack();** thread 1 приостанавливает выполнение, переходя в стадию ожидания возможности захвата блокировки cory (которой в данный момент владеет thread 2);

4) наконец,cory.hug() вызывает synchronized-метод **jareth.hugBack();** thread 2 приостанавливает выполнение, переходя в стадию ожидания возможности захвата блокировки jareth (которой в данный момент владеет thread 1).

Программа после запуска успеет вывести:

Thread 1 в jareth.hug() пытается вызвать **cory.hugBack()**

Thread 2 в cory.hug() пытается вызвать **jareth.hugBack()**

После программа зависает.

Разумеется, не исключено, что повезет, и один из потоков сумеет выполнить код hug целиком еще до момента старта второго потока.

Завершение выполнения потока

О потоке, приступившем к работе, говорят как о действующем (alive), и метод isAlive такого потока возвращает значение true.

Поток продолжает оставаться действующим до тех пор, пока не будет остановлен в результате возникновения одного из трех возможных событий:

- метод run завершил выполнение нормальным образом;
- работа метода run прервана;
- вызван метод destroy объекта потока.

Возврат из метода run посредством return или в результате естественного завершения кода — это нормальный способ окончания выполнения потока.

Вызов метода **destroy** объекта потока — это совершенно радикальный шаг.

В этом случае поток "умирает" внезапно, независимо от того, что именно он выполняет в данный момент, и не освобождает ни одной из захваченных блокировок, поэтому остальные потоки могут остаться в состоянии бесконечного ожидания.

Многими java машинами метод destroy не поддерживается, и его вызов влечет выбрасывание исключения типа NoSuchMethodError, способного остановить работу потока-инициатора, а не того потока, завершение которого предусматривалось.

Корректное завершение работы потока

Может возникнуть ситуация, когда поток создается для достижения определенной цели, а затем его выполнение необходимо прервать прежде, чем он решит поставленную задачу.

Для завершения потока вызывается метод interrupt, и код соответствующего потока должен сам следить за событием прерывания и отвечать за его выполнение.

Рассмотрим пример:

Поток 1

thread2.interrupt() ;

Поток 2

```
while(! interrupted())   {
 // что-то делается
  .....................
}
```

Рассмотрим пример:

```
class Main extends Thread {
  static int howOften;
  Thread th;
  String word;
  Main(String whatToSay) { word = whatToSay; }
    void setThread(Thread t){th=t;};
    public void run() {
        while(! interrupted()){
          th.interrupt();
          for (int i = 0; i < howOften; i++) {
             System.out.println(word);
          }
        }
    }

    public static void main(String[] args) {
      howOften = 2;
      Thread cur = currentThread();
       cur.setPriority(Thread.MAX_PRIORITY);
            Main th1=new Main("Did");
            Main th2=new Main("Did Not");
            th1.setThread(th2);
            th2.setThread(th1);
            th1.start();
            th2.start();
     }
}
```

Метод interrupt сам по себе не принуждает поток прекращать свою деятельность, хотя часто прерывает спящий режим или ожидание потока, выполняющего соответственно функции sleep или wait.

К механизму прерывания работы потока имеют отношения следующие методы:

1) **interrupt() -** посылает потоку уведомление о прерывании. Т.е. устанавливает флаг прерывания. Флаг устанавливается только при запущенном потоке.
2) **isInterrupted() -** проверяет, была ли прервана работа потока вызовом метода interrupt. Состояние прерывания не очищается.

3) **interrupted()** - статический метод, проверяющий, выполнялось ли прерывание текущего потока, и очищающий "состояние прерывания" потока.

Последнее может быть очищено только самим потоком — "внешних" способов отмены уведомления о прерывании, посланного потоку, не существует.

Прерывание посредством метода interrupt обычно не воздействует на работоспособность потока, но некоторые методы, такие как sleep и wait, будучи прерванными, выбрасывают исключение типа interruptedException.

Другими словами, если поток в момент прерывания его работы с помощью interrupt выполняет один из этих методов, они генерируют исключение InterruptedException.

В этом случае состояние прерывания потока очищается, поэтому код, обрабатывающий исключение InterruptedException, обычно должен выглядеть следующим образом:

```
void tick(int count,long pauseTime){
    try {
      for (int i  =0; i< count;i++)  {
       System.out.println('…');
       Thread.sleep(pauseTime);
      }
    }
   catch(interruptedException e){
     Thread.currentThread().interrupt();
   }
}
```

Метод tick выводит на экран символ точки count раз, "засыпая" после каждой операции на период времени, равный значению pauseTime, выраженному в миллисекундах.

Если работа потока прерывается посредством interrupt в момент выполнения им метода tick, метод sleep выбрасывает исключение типа InterruptedException.

Управление передается из цикла for в предложение catch, где уведомление о прерывании потока посылается заново.

Ожидание завершения работы потока

Поток способен ждать завершения работы другого потока, используя одну из разновидностей метода **join.**

Рассмотрим пример:

```
class CalcThread extends Thread {
   private double  result;
   public void run()  { result = calculate(); }
   public double getResult(){return  result; }
   public double calculate() {
   //   ...   вычислить значение поля  result
   }
}
```

```
class  showjoin  {
 public static void main(string[]   args){
   CalcThread calc = new CalcThread();
   calc.start() ;
   ……..
   try{
     calc.join();
     System.out.println (“The result is " +calc.getResult());
   }
   catch (InterruptedException e){
     System.out.println(“the thread is interrupted");
   }
 }
 ……
}
```

Выход из join определенно означает, что работа метода CalcThread.run завершена и значение result может быть использовано в текущем потоке.

Метод join имеет три формы:

1. **public final void join(long millis)**
   throws InterruptedException

Ожидает завершения выполнения потока или истечения заданного периода времени, выраженного в миллисекундах, в зависимости от того, что произойдет раньше.

Нулевое значение параметра означает задание бесконечного промежутка времени.

Если работа потока прерывается в момент ожидания, выбрасывается исключение типа InterruptedException.

2. **public final void join(long millis, int nanos)**
throws InterruptedException

Более чувствительная версия метода.

Величина интервала ожидания складывается из двух составляющих: millis (выраженной в миллисекундах) и nanos (в наносекундах). Вновь, суммарное нулевое значение параметра означает бесконечное ожидание. Величина nanos должна находиться в промежутке 0-999999.

3. **public final void join()**
throws InterruptedException

Метод аналогичен первому варианту при условии join(0).

Внутренняя реализация метода join может быть выражена в следующих терминах:

while(isAlive()) wait() ;

Потоки – демоны.

Существуют два вида потоков — пользовательские (user) и потоки - демоны (daemon).

Наличие пользовательских потоков сохраняет приложение в работающем состоянии.

Когда выполнение последнего из пользовательских потоков завершается, деятельность всех демонов прерывается и приложение финиширует.

Прерывание работы демонов похоже на вызов метода destroy — оно происходит внезапно и не оставляет потокам никаких шансов для выполнения завершающих операций, — поэтому демоны ограничены в выборе функциональных возможностей.

Для придания потоку статуса демона необходимо вызвать метод **setDaemon(true)**, определенный в классе Thread.

Проверить принадлежность потока к категории демонов можно с помощью метода **isDaemon().**

По умолчанию статус демона наследуется потоком от потока-"родителя" в момент создания и после старта не может быть изменен; попытка вызова setDaemon(true) во время работы потока приводит к выбрасыванию исключения типа **IllegalThreadStateException**.

Метод main по умолчанию порождает потоки со статусом пользовательских.

Рассмотрим пример:

```
class T extends Thread {
    public void run() {
      try {
        if (isDaemon()){
           System.out.println("старт потока-демона");
            sleep(1000);
            } else {
               System.out.println("старт обычного потока");
                  sleep(10);
            }
       }
       catch (InterruptedException e) {
         System.out.print("Error" + e);
        }
        finally {
            if (!isDaemon())
                System.out.println("завершение работы обычного потока");
           else
                System.out.println("завершение работы потока-демона");
       }
  }
}
```

```
public class DemoDaemonThread {
   public static void main(String[] args)
                        throws InterruptedException {
         T tr = new T();
         T trdaemon = new T();
          trdaemon.setDaemon(true);
          trdaemon.start();
          tr.start();
    }
}
```

Квалификатор volatile

Язык гарантирует, что операции чтения и записи любых значений, кроме относящихся к типам long или double, всегда выполняются атомарным образом — соответствующая переменная в любой момент времени будет содержать только то значение, которое сохранено определенным потоком, но не некую смесь результатов нескольких различных операций записи.

Однако, атомарный доступ не гарантирует, что поток всегда сможет считать самую последнюю версию значения, сохраненного в переменной.

Квалификатор volatile сообщает компилятору, что значение переменной может быть изменено в непредсказуемый момент.

Рассмотрим пример:

```
int currentvalue = 5;
 for  (;;)   {
  display.showValue(currentvalue);
  Thread.sleep(1000);  // заснуть на одну секунду
}
```

Если метод showValue сам по себе не обладает возможностью изменения значения currentvalue, компилятор волен выдвинуть предположение о том, что внутри цикла for это значение можно трактовать как неизменное, и использовать одну и ту же константу 5 на каждой итерации цикла при вызове showValue.

Но если содержимое поля currentvalue в ходе выполнения цикла подвержено обновлению посредством других потоков, предположение компилятора окажется неверным.

Поэтому правильно объявить переменную currentvalue, используя volatile

```
volatile int currentvalue = 5;
```

Класс ThreadGroup

Потоки могут объединяться в группы потоков по соображениям улучшения управляемости и безопасности.

Одна группа потоков может принадлежать другой группе, составляя иерархию с основной группой на верхнем уровне.

Потоки, относящиеся к группе, могут управляться единовременно другими словами можно прервать работу сразу всех потоков группы либо установить для них единое максимальное значение приоритета выполнения.

Объекты групп могут быть использованы также для задания верхней границы значений приоритетов потоков, относящихся к группе.

После вызова метода **setMaxPriority** с передачей ему соответствующего наибольшего допустимого значения приоритета любая попытка задания значения, превышающего установленный порог, сводится к повышению приоритета потока только до величины максимального уровня.

Рассмотрим пример:

```
static synchronized void
    maxThread(Thread thr, int priority){
        ThreadGroup grp = thr.getThreadGroup();
        thr.setPriority(priority);
        grp.setMaxPriority(thr.getPriority()- 1); //Max
                                        приоритет в группе.
}
```

Класс ThreadGroup поддерживает следующие конструкторы и методы:

1. **public ThreadGroup(String name)**

Создает новый объект класса ThreadGroup, принадлежащий той группе потоков, к которой относится и поток-"родитель".

Как и в случае объектов потоков, имена групп не используются исполняющей системой непосредственно, но в качестве параметра name имени группы может быть передано значение null.

2. **public ThreadGroup(ThreadGroup parent, String name)**

Создает новый объект класса ThreadGroup с указанным именем name в составе "родительской" группы потоков parent.

Если в качестве parent передано значение null, выбрасывается исключение типа NullPointerException.

3. **public final String getName()**

Возвращает строку имени текущей группы потоков.

4. **public final ThreadGroup getParent()**

Возвращает ссылку на объект "родительской" группы потоков либо null, если такового нет (последнее возможно только для группы потоков верхнего уровня иерархии).

5. **public final void setDaemon(boolean daemon)**

придает текущему объекту группы потоков статус принадлежности к категории групп-демонов

6. **public final boolean isDaemon()**

Возвращает статус принадлежности текущего объекта группы потоков к категории групп-демонов

7. **public final void setMaxPriority(int maxpri)**

Устанавливает верхнюю границу приоритетов выполнения для текущей группы потоков.

8. **public final int getMaxPriority()**

Возвращает ранее заданное значение верхней границы приоритетов выполнения для текущей группы потоков.

9. **public final boolean parentOf(ThreadGroup g)**

Проверяет, является ли текущая группа "родительской" по отношению к группе g либо совпадает с группой g

10. **public final void checkAccess()**

Выбрасывает исключение типа SecurityException, если текущему потоку не позволено воздействовать на параметры группы потоков; в противном случае просто возвращает управление.

11. **public final void destroy()**

Уничтожает объект группы потоков.

Группа не должна содержать потоков, иначе метод выбрасывает исключение типа IllegalThreadStateException.

Если в составе группы имеются другие группы, они также не должны содержать потоков.

Не уничтожает объекты потоков, принадлежащих группе.

12. **public int activeCount()**

Возвращает приблизительное количество действующих (активных) потоков группы, включая и те потоки, которые принадлежат вложенным группам.

Количество нельзя считать точным, поскольку в момент выполнения метода оно может измениться, — одни потоки "умирают", а другие создаются.

Поток считается действующим, если метод isAlive соответствующего объекта Thread возвращает значение true.

13. **public int enumerate(**
Thread[] threadsInGroup, boolean recurse)

Заполняет массив threadsInGroup ссылками на объекты действующих потоков группы, принимая во внимание размер массива, и возвращает количество сохраненных ссылок.

Если значение параметра **recurse** равно false, учитываются только те потоки, которые принадлежат непосредственно текущей группе, а в противном случае — еще и потоки, относящиеся ко всем вложенным группам.

14. **public int enumerate(**
Thread[ ] threadsInGroup)

метод аналогичен предыдущему при условии enumerate(threadsInGroup true).

15. **public int activeGroupCount()**

подобен методу activeCount, но подсчитывает количество групп, включая вложенные

16. **public int enumerate(**
ThreadGroup[ ] groupsInGroup,
boolean recurse)

Подобен соответствующему варианту метода enumerate для подсчета потоков, но заполняет массив groupsInGroup ссылками на объекты вложенных групп потоков.

17. **public int enumerate(**
ThreadGroup[ ] groupsInGroup)

Метод аналогичен предыдущему при условии enumerate(groupsInGroup, true).

18. **public void uncaughtException(**
Thread thr, Throwable exc)

Вызывается, когда поток thr в текущей группе генерирует исключение exc, которое далее не обрабатывается.

В классе Thread существует два статических метода, позволяющих обрабатывать данные о группе, к которой принадлежит текущий поток.

1. **public static int activeCount()**

Возвращает количество действующих потоков в группе, к которой относится текущий поток.

2. **public static int enumerate(**
Thread[ ] threadsInGroup)

Метод аналогичен вызову enumerate (threadsInGroup) объекта группы, которой принадлежит текущий поток.

Метод stop()

Вызов метода stop приводит к возникновению в соответствующем потоке асинхронного исключения типа ThreadDeath.

Объекты типа ThreadDeath могут быть отловлены точно так же, как и другие, а если исключение не подвергается обработке, последствия такого бездействия со временем приведут к аварийному завершению работы потока.

Переменные ThreadLocal

Класс ThreadLocal предоставляет возможность иметь единую логическую переменную, обладающую независимыми значениями в контексте каждого отдельного потока.

В составе объекта ThreadLocal есть методы set и get, которые позволяют соответственно присваивать и считывать значения переменной для текущего потока.

Рассмотрим пример:

```
public class SomeBuilderDemo {
 public static class SomeBuilder {
   private ThreadLocal<Integer> counter =
                                                  new ThreadLocal<Integer>();
   public void build() {
    System.out.println("Thread " +
      Thread.currentThread().getName() + " Build some structure");
    if (counter.get() == null)
       counter.set(0);
    counter.set(counter.get() + 1);
    try {
      Thread.sleep(100);
    }catch (InterruptedException e) { e.printStackTrace(); }
   }
   public int getCount() {return counter.get();}
}
```

```
public static class SomeBuilderThread extends Thread {
    private SomeBuilder builder;
    public SomeBuilderThread(SomeBuilder builder) {
        this.builder = builder;
    }
    public void run() {
        for (int i = 0; i < Math.random() * 10; i++) {
                builder.build();
        }
        System.out.println("My name is " + getName() +
                "and I built " + builder.getCount() + " things");
    }
}
```

```
 public static void main(String[] args) {
    SomeBuilder builder = new SomeBuilder();
    Thread thread1 = new SomeBuilderThread(builder);
    Thread thread2 = new SomeBuilderThread(builder);
     try {
         thread1.start();
          thread2.start();
          thread1.join();
          thread2.join();
       } catch (InterruptedException e) {
          e.printStackTrace();
       }
  }
}
```

Результат работы:

```
Thread Thread-1 Build some structure...
Thread Thread-0 Build some structure...
Thread Thread-1 Build some structure...
Thread Thread-0 Build some structure...
Thread Thread-0 Build some structure...
Thread Thread-1 Build some structure...
Thread Thread-0 Build some structure...
Thread Thread-1 Build some structure...
Thread Thread-1 Build some structure...
My name is Thread-0 and I built 4 things
Thread Thread-1 Build some structure...
Thread Thread-1 Build some structure...
My name is Thread-1 and I built 7 things
```

В строках

```
if (counter.get() == null)
    counter.set(0);
```

производится ее инициализация. **Важно! т.к. ThreadLocal-переменные изолированы в потоках, то инициализация такой переменной должна происходить в том же потоке, в котором она будет использоваться.**

Ошибкой является инициализация такой переменной - вызов метода **set()** - в главном потоке приложения, т.к. в данном случае значение, переданное в методе **set(),** будет "захвачено" для главного потока, и при вызове метода **get()** в целевом потоке будет возвращен null.

Когда поток прекращает существование, значения, установленные для этого потока в переменных ThreadLocal, недостижимы и могут быть уничтожены сборщиком мусора, если какие-либо ссылки на них отсутствуют.

Отладка потоков

В составе класса Thread есть несколько методов, которые можно использовать в процессе отладки кода многопоточного приложения.

1. **public String toString()**

Возвращает строковое представление содержимого объекта потока, включающее его наименование, значение приоритета выполнения и имя соответствующей группы потоков.

2. **public static void dumpStack()**

Выводит на консоль данные трассировки для текущего потока.

В классе ThreadGroup существую также методы для отладки потоков

1. **public String toString()**

Возвращает строковое представление содержимого объекта группы потоков, включающее его наименование и значение приоритета выполнения.

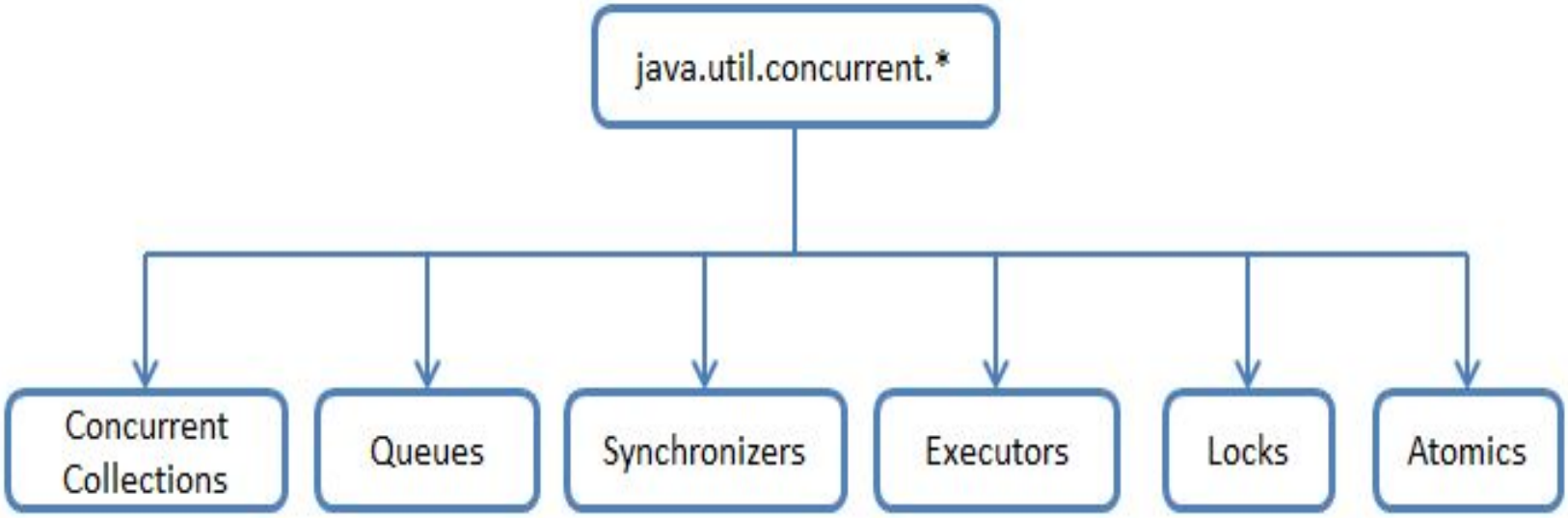
2. **public void list()**

Выводит на консоль информацию об объекте ThreadGroup, включающую результаты вызовов toString для каждого из потоков, принадлежащих группе, и всех вложенных групп.

Потоки в J2SE 5.0

Добавлены пакеты классов **java.util.concurrent.locks**, **java.util.concurrent**, **java.util.concurrent.atomic,** возможности которых обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков параллельных (concurrent) классов, вызов утилит синхронизации, использование семафоров, ключей и atomic-переменных.

Схематично concurrent выглядит следующим образом.



Concurrent Collections — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам.

Queues — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer» 'а.

Synchronizers — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.

Executors — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.

Locks — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.

Atomics — классы с поддержкой атомарных операций над примитивами и ссылками.

Concurrent Collections

CopyOnWriteArrayList<E> — Потокобезопасный аналог ArrayList, реализованный с CopyOnWrite алгоритмом.

Главная идея copy-on-write — при копировании областей данных создавать реальную копию только когда ОС обращается к этим данным с целью записи.

CopyOnWriteArraySet<E> — Имплементация интерфейса Set, использующая за основу CopyOnWriteArrayList.

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, доступ к данным лочится по сегментам, а не по одному объекту.

В дополнение, итераторы представляют данные на определенный срез времени и не кидают ConcurrentModificationException.

ConcurrentNavigableMap<K,V> — 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> — 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.

ConcurrentSkipListSet<E> — Имплементация Set интерфейса, выполненная на основе ConcurrentSkipListMap.

Queues

ConcurrentLinkedQueue<E> — В имплементации используется wait-free алгоритм от Michael & Scott, адаптированный для работы с garbage collector'ом.

Этот алгоритм довольно эффективен и очень быстр, т.к. построен на CAS.

Метод size() может работать долго, т.ч. лучше постоянно его не дергать.

ConcurrentLinkedDeque<E> — 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> — При обработке больших потоков данных через очереди становится явно недостаточно использования ConcurrentLinkedQueue.

Если потоки, разгребающие очередь перестанут справляться с наплывом данных, то может довольно быстро вылететь outofmemory 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.

Для этого создан 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> — 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.

DelayQueue<E extends Delayed> —Класс, который позволяет вытаскивать элементы из очереди только по прошествии некоторой задержки, определенной в каждом элементе через метод getDelay интерфейса Delayed.

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.

PriorityBlockingQueue<E> — Является многопоточной оберткой над PriorityQueue.

При вставлении элемента в очередь, его порядок определяется в соответствии с логикой Comparator'a или имплементации Comparable интерфейса у элементов. Первым из очереди выходит самый наименьший элемент.

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.

BlockingDeque<E> — Интерфейс, описывающий дополнительные методы для двунаправленной блокирующей очереди. Данные можно вставлять и вытаскивать с двух сторон очереди.
LinkedBlockingDeque<E> — 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> — 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> — Реализация TransferQueue на основе алгоритма Dual Queues with Slack.

Активно использует CAS и парковку потоков, когда они находятся в режиме ожидания.

Synchronizers

Semaphore — Семафоры 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».

Рассмотрим пример:

```
import java.util.concurrent.Semaphore;
public class Main {
  public static void main(String args[]) throws Exception {
//true –гарантия того, что первый поток который вызвал acquire
    получит доступ к блокируемому объекту
   Semaphore sem = new Semaphore(1, true);

   Thread thrdA = new Thread(new MyThread(sem, "Message 1"));
   Thread thrdB = new Thread(new MyThread(sem, "Message 2"));

   thrdA.start();
   thrdB.start();

   thrdA.join();
   thrdB.join();
  }
}
```

```
class MyThread implements Runnable {
 Semaphore sem;
 String msg;
 MyThread(Semaphore s, String m) {
   sem = s;
   msg = m;
 }
 public void run() {
   try {
     sem.acquire();
     System.out.println(msg);
     Thread.sleep(10);
     sem.release();
   } catch (Exception exc) {
       System.out.println("Error Writing File");
   }
 }
}
```

CountDownLatch — 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.

Рассмотрим пример:

```
public class HungryStudent implements Runnable {
   static CountDownLatch c;
   public void run() {
   try {
     c.await();
      System.out.println("Студент поел");
    } catch (InterruptedException e) {}
   }
  public static void main(String[] args) {
     int n = 5;
     new Thread(new HungryStudent()).start();
     c = new CountDownLatch(n);
     for (int i = 0; i < n; i++)
      new Thread(new Kock()).start();
  }
}
public class Kock implements Runnable {
   static public int count=0;
   private int id=count++;
   public void run() {
     System.out.println("Готова еда от Кок№"+id);
     HungryStudent.c.countDown();
   }
}
```

CyclicBarrier- приостанавливает все потоки, которые вызывают его метод await до тех пор, пока их не наберётся нужно количество, указанное в конструкторе.

Также в конструкторе можно передать объект, реализующий интерфейс Runnable, который будет выполнен по достижению размера очереди потоков определённого количества.

Рассмотрим пример.

```
class MyThread extends Thread{
    private CyclicBarrier queue;
    public MyThread(CyclicBarrier queue){
        this.queue=queue;
    }
    public void run(){
        System.out.println("Thread call await");
        try {
            queue.await();
        System.out.println("Threads running");
        } catch (InterruptedException ex) {

        } catch (BrokenBarrierException ex) { //Данное exception
вызывается если барьер был сломан посредством вызова reset()
        }
    }
}
```

```
class Hello implements Runnable{
    public void run(){
        System.out.println("Hello");
    }
}
public class Main {
public static void main(String[] args) {
        Hello h=new Hello();
        CyclicBarrier queue=new CyclicBarrier(5,h);
        for(int i=0; i<7;i++){
            new MyThread(queue).start();
        }
    }
}
```

Exchanger<V> — 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 — 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.

Executors

Future<V> — 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.

RunnableFuture<V> — Если Future — это интерфейс для Client API, то интерфейс RunnableFuture уже используется для запуска асинхронной части.

Успешное завершение метода run() завершает асинхронную операцию и позволяет вытаскивать результаты через метод get.

Callable<V> — 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.

FutureTask<V> — Имплементация интерфейса Future/RunnableFuture. Асинхронная операция принимается на вход одного из конструкторов в виде Runnable или Callable объектов.

Сам же класс FutureTask предназначен для запуска в worker потоке, например через new Thread(task).start(), или через ThreadPoolExecutor.

Результаты работы асинхронной операции вытаскиваются через метод get(...).

Delayed — Используется для асинхронных задач, которые должны начаться в будущем, а также в DelayQueue.

Позволяет задавать время до начала асинхронной операции.

ScheduledFuture<V> — Маркерный интерфейс, объединяющий Future и Delayed интерфейсы.

RunnableScheduledFuture<V> — Интерфейс, объединяющий RunnableFuture и ScheduledFuture.

Дополнительно можно указывать является ли задача одноразовой или же должна запускаться с заданной периодичностью.

Executor Services

Executor — Представляет собой базовый интерфейс для классов, реализующих запуск Runnable задач.

Тем самым обеспечивается развязка между добавлением задачи и способом её запуска.

ExecutorService — 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.

ScheduledExecutorService — В дополнении к методам ExecutorService, данный интерфейс добавляет возможность запускать отложенные задачи.

AbstractExecutorService — Абстрактный класс для построения ExecutorService'а. Имплементация содержит базовую имплементацию методов submit, invokeAll, invokeAny.

От этого класса наследуются ThreadPoolExecutor, ScheduledThreadPoolExecutor и ForkJoinPool.

ThreadPoolExecutor & Factory

xecutors — Класс-фабрика для создания ThreadPoolExecutor, ScheduledThreadPoolExecutor.

Также, тут содержатся разные адаптеры Runnable-Callable, PrivilegedAction-Callable, PrivilegedExceptionAction-Callable и другие.

ThreadPoolExecutor —Используется для запуска асинхронных задач в пуле потоков.

Тем самым практически не тратится время 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.

ScheduledThreadPoolExecutor — В дополнении к методам ThreadPoolExecutor, позволяет запускать задачи после определенной задержки, а также с некоторой периодичностью, что позволяет реализовать на базе этого класса Timer Service.

ThreadFactory — По умолчанию, ThreadPoolExecutor использует стандартную фабрику потоков, получаемую через Executors.defaultThreadFactory(). Если нужно что-то больше, например задание приоритета или имени потока, то можно создать класс с реализацией этого интерфейса и передать его в ThreadPoolExecutor.

RejectedExecutionHandler — Позволяет определить обработчик для задач, которые по каким то причинам не могут быть выполнены через ThreadPoolExecutor.

Такой случай может произойти, когда нет свободных потоков или сервис выключается или выключен (shutdown).

Несколько стандартных имплементаций находятся в классе ThreadPoolExecutor:

CallerRunsPolicy — запускает задачу в вызывающем потоке;
AbortPolicy — кидает exception;

DiscardPolicy — игнорирует задачу;

DiscardOldestPolicy — удаляет самую старую незапущенную задачу из очереди, затем пытается добавить новую задачу еще раз.

Fork Join

В java 1.7 появился новый Fork Join фреймворк для решения рекурсивных задач, работающих по алгоритмам разделяй и влавствуй или Map Reduce.

ForkJoinPool — 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 — 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 — Абстрактный класс от ForkJoinTask, с объявлением метода compute, в котором должна производиться асинхронная операция в наследнике.

RecursiveAction — Отличается от RecursiveTask тем, что не возвращает результат.

ForkJoinWorkerThread — Используется в качестве имплементации по умолчанию в ForkJoinPoll.

При желании можно отнаследоваться и перегрузить методы инициализации и завершения worker потока.

Completion Service

CompletionService — Интерфейс сервиса с развязкой запуска асинхронных задач и получением результатов.

Так, для добавления задач используются методы submit, а для вытаскивания результатов *завершенных* задач используются блокирующий метод take и неблокирующий poll.

ExecutorCompletionService — По сути является враппером над любым классом, реализующим интерфейс Executor, например ThreadPoolExecutor или ForkJoinPool.

Используется преимущественно тогда, когда хочется абстрагироваться от способа запуска задач и контроля за их исполнением.

Если есть завершенные задачи — вытаскиваем их, если нет — ждем в take пока что-нибудь не завершится.

В основе сервиса по умолчанию используется LinkedBlockingQueue, но может быть передана и любая другая имплементация BlockingQueue.

Locks

Condition — 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 — 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.

Класс reentrantLock

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.

```
Lock lock = new reentrantLock();
lock.lock();
try {
    // update object state
} finally { lock.unlock(); }
```

Задача Producer-Consumer

Имеются один или несколько производителей, генерирующих данные некоторого типа (записи, символы и т.п.) и помещающих их в буфер, а также единственный потребитель, который извлекает помещенные в буфер элементы по одному.

Требуется защитить систему от перекрытия операций с буфером, т.е. обеспечить, чтобы одновременно получить доступ к буферу мог только один процесс (производитель или потребитель).

Рассмотрим решение этой задачи:

```
public class ProducerConsumer {
 final Lock lock = new ReentrantLock();
 final Condition emty = lock.newCondition();
 final Condition filled = lock.newCondition();
 private int variable = -1;
public int consume() throws InterruptedException {
  lock.lock();
  try {
   while (variable == -1) { filled.await(); }
   try {
    System.out.println("Consumer: " + variable);
     return variable;
   } finally {
     variable = -1;
     emty.signalAll();
   }
  } finally { lock.unlock(); }
 }
```

```
public void produce(int value) throws InterruptedException {
  lock.lock();
  try {
    while (variable != -1) { emty.await(); }
    this.variable = value;
    System.out.println("Producer: " + variable);
    filled.signalAll();
  } finally { lock.unlock(); }
}
```

```
public static void main(String[] args) {
  final ProducerConsumer producerConsumer = new ProducerConsumer();
  final Thread consumer = new Thread() {
          public void run() {
              while(true) {
                   try { producerConsumer.consume(); }
                   catch (InterruptedException e) { break; }
              }
              System.out.println("Consume Done.");
           }
    };
    consumer.start();
    final Random random = new Random();
    try { for (int i = 0; i < 10; i++) {
          final int value = random.nextInt();
          producerConsumer.produce(value); }
     } catch (InterruptedException e) {
           System.out.println("Produce interrupted.");
     } consumer.interrupt();
 }
}
```

ReadWriteLock — Дополнительный интерфейс для создания read/write локов.

Такие локи полезны, когда в системе много операций чтения и мало операций записи.

ReentrantReadWriteLock — 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 — Read lock для reader'ов, получаемый через readWriteLock.readLock().

ReentrantReadWriteLock.WriteLock — Write lock для writer'ов, получаемый через readWriteLock.writeLock().

LockSupport — Предназначен для построения классов с локами. Содержит методы для парковки потоков вместо устаревших методов Thread.suspend() и Thread.resume().

AbstractOwnableSynchronizer — Базовый класс для построения механизмов сихнронизации.

Содержит всего одну пару геттер/сеттер для запоминания и чтения эксклюзивного потока, который может работать с данными.

AbstractQueuedSynchronizer — Используется в качестве базового класса для механизма синхронизации в FutureTask, CountDownLatch, Semaphore, ReentrantLock, ReentrantReadWriteLock.

Может применяться при создании новых механизмов синхронизации, полагающихся на одиночное и атомарное значение int.

AbstractQueuedLongSynchronizer — Разновидность AbstractQueuedSynchronizer, которая поддерживает атомарное значение long.

Atomics

Атомарные переменные представлены в виде классов, которые реализуют механизм, гарантирующий, что операции с этими переменными будут выполненяться как атомарная операция.

Операция присваивания (например, getAndSet() в AtomicInteger) в атомарной переменной выглядит так:

1. Читается текущее значение переменной.
   Это значение записывается в локальную переменную.
2. Выполняется попытка заменить старое значение на новое.
   Если старое значение такое же, как во время чтения его в локальную переменную, то выполняется присваивание переменной нового значения.
   Если значение изменилось, то метод начинает работать снова, с пункта 1.

AtomicBoolean, **AtomicInteger**, **AtomicLong**, **AtomicIntegerArray**, **AtomicLongArray** —За счет использования CAS, операции с этими классами работают быстрее, чем если синхронизироваться через synchronized/volatile.

Плюс существуют методы для атомарного добавления на заданную величину, а также инкремент/декремент.

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).

AtomicStampedReference — Класс для атомарных операцией со следующей парой полей: ссылка на объект и int значение.

AtomicReferenceArray — Массив ссылок на объекты, который может атомарно обновляться.

AtomicIntegerFieldUpdater, **AtomicLongFieldUpdater**,**AtomicReferenceFieldUpdater** — Классы для атомарного обновления полей по их именам через reflection.

Пример.

```
import java.util.concurrent.atomic.AtomicInteger;
public class TaskAtomic implements Runnable {
   private AtomicInteger number;
   public TaskAtomic() {
      this.number = new AtomicInteger();
   }
   public void run() {
    for (int i = 0; i < 1000000; i++) {
      number.set(i);
    }
  }
}
```

Библиотека Stream

java.util.stream введен для поддержки распараллеливания вычислений в потоках. Таким образом теперь потоки делятся на последовательные и параллельные.

Прежде всего рассмотрим последовательные потоки.

Stream представляет набор различных элементов: промежуточных и конечных. Конечные элементы возвращают либо void либо не-stream результат.

Промежуточные возвращают stream.

Методы stream могут принимать в качестве параметров lambda функции.

Рассмотрим пример:

```
List<String> myList = Arrays
    .asList("a1", "a2", "b1", "c2", "c1");
    myList.stream()
            .filter(s -> s.startsWith("c"))
            .map(String::toUpperCase)
            .sorted()
            .forEach(System.out::println);
```

На экране получим

C1

C2

Можно также использовать of для связки последовательности ссылок объектов:

```
Stream.of("a1", "a2", "a3")
        .findFirst()
        .ifPresent(System.out::println);
```

На экране

a1

Существуют также специальные виды stream типа **IntStream, LongStream** и **DoubleStream**

Рассмотрим пример:

```
IntStream.range(1, 4)
        .forEach(System.out::println);
```

На экране

1

2

3

Примитивные потоки поддерживают дополнительные терминальные функции avarage() и sum()

Пример:

```
Arrays.stream(new int[] {1, 2, 3})
        .map(n -> 2 * n + 1)
        .average()
        .ifPresent(System.out::println);
```

На экране

5.0

Для примитивных стримов существуют операторы **mapToInt()**, **mapToLong()** и **mapToDouble**

Пример.

```
Stream.of("a1", "a2", "a3")
        .map(s -> s.substring(1))
        .mapToInt(Integer::parseInt)
        .max()
        .ifPresent(System.out::println);
```

На экране

3

Примитивные стримы можно преобразовать в объектные стримы.

Рассмотрим пример:

```
IntStream.range(1, 4)
                .mapToObj(i -> "a" + i)
                .forEach(System.out::println);
```

На экране

a1

a2

a3

Важнейшая характеристика промежуточных операторов- использование ленивых вычислений.

Пример:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> { System.out.println("filter: " + s);
     return true; });
```

В данном случае ничего на консоль выведено не будет.

Выполнение всех операций произойдет только тогда, когда будет вызван терминальный оператор:

```
Stream.of("d2", "a2", "b1", "b3", "c")
  .filter(s -> { System.out.println("filter: " + s);
   return true; })
  .forEach(s -> System.out.println("forEach: " + s));
```

На экране

filter: d2

forEach: d2

filter: a2

forEach: a2

filter: b1

forEach: b1

filter: b3

forEach: b3

filter: c

forEach: c

Стримы в Java не допускают повторное использование:

```
Stream<String> stream =
        Stream.of("d2", "a2", "b1", "b3", "c")
                .filter(s -> s.startsWith("a"));
stream.anyMatch(s -> true); // ok
stream.noneMatch(s -> true); // exception
```

Для преодоления этой трудности необходимо создать новую потоковую цепочку для каждого терминального оператора

Рассмотрим пример:

```
Supplier<Stream<String>> streamSupplier =
            () -> Stream.of("d2", "a2", "b1", "b3", "c")
             .filter(s -> s.startsWith("a"));
streamSupplier.get().anyMatch(s -> true); // ok
streamSupplier.get().noneMatch(s -> true); // ok
```

Стримы поддерживают большое количество различных операторов.

Рассмотрим пример:

```
class Person {
   String name;
   int age;
   Person(String name, int age) {
     this.name = name;
     this.age = age;
   }
   public String toString() { return name; }
}
List<Person> persons = Arrays.asList(
   new Person("Max", 18),
   new Person("Peter", 23),
   new Person("Pamela", 23),
   new Person("David", 12));
```

```
List<Person> filtered =
   persons .stream()
            .filter(p -> p.name.startsWith("P"))
            .collect(Collectors.toList());
System.out.println(filtered);
```

Collect преобразовывает последовательность объектов в List, Set или Map.

```
Map<Integer, List<Person>> personsByAge =
   persons .stream()
            .collect(Collectors.groupingBy(p -> p.age));
personsByAge .forEach((age, p) ->
                System.out.format("age %s: %s\n", age, p));
```

На экране

age 18: [Max]

age 23: [Peter, Pamela]

age 12: [David]

Parallel Streams

Паралельные стримы выполняются в многопоточной среде.

Рассмотрим пример:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
        .parallelStream()
        .filter(s -> { System.out.format("filter: %s [%s]\n", s,
                                        Thread.currentThread().getName());
         return true; })
        .map(s -> { System.out.format("map: %s [%s]\n", s,
                                        Thread.currentThread().getName());
         return s.toUpperCase(); })
       .forEach(s -> System.out.format("forEach: %s [%s]\n", s,
                                        Thread.currentThread().getName()));
```

Для изменения количества потоков выполнения в параметрах JVM необходимо указать

-Djava.util.concurrent.ForkJoinPool.common.parallelism=5