15. **Многопоточность** (экспресс-обзор)

Информатика, ИТИС, 1 курс
М.М.Абрамский
2017

Example #1 (multitasking)

- *Я печатаю документ в Power Point*
- *Я слушаю музыку*
- *Я качаю файл из интернета*

И все это происходит параллельно, не мешая друг другу.

Example #2 (сеть)

- Мы подключаемся в Интернет и заходим на какой-нибудь сайт.
  - Отправляем запросы на сервер.
- Люди на земле сидят на одних и тех же сайтах (серверах), запрашивая разные вещи, не мешая друг другу при этом.
  - Иначе я бы не мог зайти на facebook.com, пока кениец Джон Нганга не закончил бы с ним работу.
- Значит, на сервере что-то умеет независимо обрабатывать разные подключения клиентов.

Цель

- Обеспечить независимое и одновременное выполнение нескольких «программ»
- И ничего общего с ускорением!

Процессор

- До изобретения многоядерных процессоров выполнял одну операцию в каждый момент времени.

Где тогда параллельность?

Виртуальная параллельность (Метафора - книги)

- Можно читать 3 книги одну за другой
  - Это последовательное выполнение
- Можно читать по несколько страниц одной, после переходя к другой. В итоге через некоторое время все 3 книги оказываются прочитаны примерно на одинаковое количество страниц
  - Это виртуальная параллельность

Thread (тред, поток, нить)

- java.lang
- Блок кода, который может выполняться параллельно с другими такими же thread-ами.
- В любой Java-программе есть хотя бы один поток, и это...

Thread (тред, поток, нить)

- java.lang
- Блок кода, который может выполняться параллельно с другими такими же thread-ами.
- В любой Java-программе есть хотя бы один поток, и это… main!

Как работают потоки

- Пусть есть N потоков
- В каждый момент времени работает **один** поток, остальные стоят
  - **«активный** поток», «поток **перехватил управление»;**
- Переключение (Передача управления) между потоками происходит быстро и … в общем случае сильно неупорядоченно.
  - При этом поток может добровольно отпустить управление
  - А может быть насильно прерван другим, более приоритетным

  ! Что в нашей метафоре с книгами является передачей управления?

Потоки и процессы

- Поток != Процесс
  - Поток легче – его использование менее затратно. Например, процесс требует собственного адресного пространства в ОП.
  - 1 процесс может содержать несколько потоков
    - 1 программа внутри себя хочет многозадачности

Потоки и процессы

- «Слушаю музыку, работаю в ворде»
  – 2 процесса!

- «Работаю в ворде – редактирую документ, пока он отправляется на печать»
  – 1 процесс, 2 потока

Многопоточность никак не связано с ускорением

- Можно реализовывать параллельные алгоритмы,
- Но они не будут работать быстрее, чем аналогичные последовательные.

Метафора с книгами: в каком порядке вы бы их не читали, во любом случае вы должны потратить одно и то же время на все страницы всех книг.

Никак не связано с ускорением (наглядно и даже без потоков)

// Последовательно считаются n! и сумма от 1 до n.

```
for (int i = 1; i <= n; i++) {
    fact *= i;
}
for (int i = 1; i <= n; i++) {
    sum += i;
}
```

// А вот они считают типа параллельно.
// Но ясно видно, что никакого ускорения нет.

```
for (int i = 1; i <= N; i++) {
    fact *= i;
    sum += i;
}
```

Жизненный цикл потока. Метод getState() в Thread

- Создан, но не запущен (NEW)
- Запущен (RUNNABLE)
  - isAlive() возвращает true
- Приостановлен (BLOCKED, WAITING, TIMED_WAITING)
  - например, во время методов wait, sleep, join
- Закончил работу (TERMINATED)
  - isAlive() возвращает false

Реализация своих потоков

- Свой поток – наследник класса Thread
- Самое главное – реализовать run()

```
public class MyThread extends Thread {

    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(i);
        }

    }

}
```

Запуск потока

В другом месте (обычно в main):

```
// это только создание объекта для потока
MyThread t1 = new MyThread(…);

// а вот это уже запуск потока
t1.start();
```

! *посмотрите варианты конструкторов Thread*

Не путать run() и start()

- **run()** описывает поведение при запуске потока,
- **start()** запускает поток с поведением, описанным в run().
- *Нет, конечно у объекта потока можно вызвать run(), не вопрос. Но это малоэффективно…*
  - Что будет?

Пусть есть 3 потока

```
public class MyThread extends Thread {
    public void run() {
        for (int j = 0; j < 1000; j++) {
            System.out.println(getName() +
                               ": " + j);
        }
    }

    public static void main(String[] args) {
        (new MyThread()).start();
        (new MyThread()).start();
        (new MyThread()).start();
    }
}
```

Ожидание

```
Thread-0: 0
Thread-1: 0
Thread-2: 0
Thread-0: 1
Thread-1: 1
Thread-2: 1
Thread-0: 2
Thread-1: 2
Thread-2: 2
Thread-0: 3
Thread-1: 3
Thread-2: 3
...
```

Реальность

```
Thread-2: 0
Thread-2: 1
Thread-2: 2
Thread-2: 3
Thread-1: 0
Thread-1: 1
Thread-0: 0
Thread-0: 1
Thread-1: 2
Thread-2: 4
Thread-2: 5
Thread-2: 6
Thread-2: 7
Thread-1: 3
…
```

Реальность (запуск #2)

```
Thread-1: 0
Thread-0: 0
Thread-2: 0
Thread-0: 1
Thread-1: 1
Thread-0: 2
Thread-2: 1
Thread-0: 3
Thread-1: 2
Thread-0: 4
Thread-2: 2
Thread-0: 5
Thread-1: 3
…
```

Важно!

- Даже потоки с одним и тем же **run** не гарантируют синхронное выполнение
- Для разных запусков одного и того же много поточного кода передача управления происходит неупорядоченно
  - Опа! Одна и та же программа с одним и тем же вводом данных дает разный выход!
  - Надо как-нибудь **синхронизироваться!**

sleep()

- Отправляет текущий поток в ожидание на указанное в миллисекундах время.

```
public void run() {
    for (int j = 0; j < 1000; j++) {
        System.out.println(getName() + ": " + j);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

sleep() - запуск

```
Thread-2: 0
Thread-1: 0
Thread-0: 0
Thread-2: 1
Thread-1: 1
Thread-0: 1
Thread-2: 2
Thread-0: 2
Thread-1: 2
Thread-2: 3
Thread-0: 3
Thread-1: 3
Thread-0: 4
Thread-1: 4
Thread-2: 4
```

Хорошо, но очень медленно...

Передача результата в main

```
public class MyThread extends Thread {
    private int s = 0;
    public int getS() {
        return s;
    }
    public void run() {
        for (int j = 0; j < 50; j++) {
            s += j;
        }
    }
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
        int result = t1.getS();
        System.out.println(result);
    }
```

Выведется….

…o

main работает параллельно с MyThread, и забирает текущее значение s

- Можно решить проблему циклом isAlive() – *но такие циклы грузят проц.*
- Можно использовать join()

join()

```
public static void main(String[] args) {
    MyThread t1 = new MyThread();
    t1.start();
    try {
        t1.join();
        int result = t1.getS();
        System.out.println(result);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Как работает join:

Main вызвал **join** у **t1** – значит **main** ждет, пока **t1** не закончит работу. Затем **main** продолжит работу

Runnable

```
class MyThread2 implements Runnable{
    …
    public void run() {
        // Поведение потока
        …
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread2());
        t.start();
        …
    }
}
```

Runnable

```
class MyThread2 implements Runnable{
    …
    public void run() {
        // Поведение потока
        …
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyThread2());
        t.start();
        …
    }
}
```

Зачем пользователю нашего класса MyThread2 знать тонкости про то, как запускать поток через Runnable?
И еще, теперь что, все потоки – Thread, даже с разным поведением??

Прячем Thread в Runnable

```
class MyAwesomeThread implements Runnable {
    private Thread thread;
    public MyAwesomeThread() {
        // создаем поток, передавая поведение
        // нашего MyAwesomeThread
        thread = new Thread(this);
        thread.start();
    }
    public void run() {
         …
    }
}
public class Main {
    public static void main(String[] args) {
        // Поток сразу создается и запускается
        MyAwesomeThread t = new MyAwesomeThread();
}}
```

! Пройти самостоятельно

- Потоки-демоны (Daemon),
  - setDaemon(…)
- Приоритеты потоков,
  - setPriority(…)
- Другие атрибуты и методы класса Thread

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

  - yield, interrupt, …

Общие данные потоков

- Как потоки **шарят** (share) данные?
  - Через конструкторы – передаем один и тот же объект в конструкторы разных потоков.
  - ~~Все проблемы из-за шаренных объектов.~~

```
class BonusCard {
    private int bonuses;
    public int getBonuses() {return bonuses; }

    public BonusCard(int bonuses) {
        this.bonuses = bonuses;
    }

    public boolean use(int n) {
        if (bonuses >= n) {
            bonuses -= n;
            System.out.println(bonuses + " left.");
            return true;
        }
        else {
            System.out.println("OOOOOPS");
            return false;
        }
    }
}
```

Пример. Общая семейная бонусная карта

```
class Human extends Thread {
    private BonusCard bonusCard;
    private String who;
    public Human(BonusCard bc, String who) {
        this.bonusCard = bc;
        this.who = who;
    }
    public void shoppingWithBonuses(int bonuses) {
        if (bonusCard.getBonuses() >= bonuses) {
            System.out.println(who + " is gonna buy something.");
            if (bonusCard.use(bonuses))
                System.out.println(who + " bought something.");
        }
        else {
            System.out.println("Sorry, Honey, no money.");
        }
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
           shoppingWithBonuses(7);
        }
    }
}
```

И где-то в main

```
BonusCard bc = new BonusCard(150);
Human husband = new Human(bc, "husband");
Human wife = new Human(bc, "wife");
husband.start();
wife.start();
```

Запуск

- Два потока (муж и жена) тратят бонусы (общий ресурс) независимо друг от друга

```
husband is gonna buy something.
wife is gonna buy something.
143 left.
136 left.
husband bought something.
wife bought something.
husband is gonna buy something.
129 left.
husband bought something.
```

Когда бонусы близки к концу

```
…
wife bought something.
husband is gonna buy something.
wife is gonna buy something.
3 left.
OOOOOPS
husband bought something.
…
```

Сработавший OOOOPS – это форс-мажор.

Ведь по сути human проверяет карту перед использованием.

Но код **непотокобезопасен** - один поток меняет общий объект без ведома другого.

Синхронизация

- Один поток берет объект в единоличное использование (**блокирует доступ** к объекту)
- Все остальные потоки ждут, пока он не **отпустит блокировку**
- В нашем примере – поток (муж или жена) должна синхронизировать использование бонуса, чтобы другой поток не встревал между *узнаванием баланса* и *использованием*.

Монитор

- Объект, обеспечивающий синхронизацию
  - «Пускаю по одному!»
- Mutual Exclusive Lock (Mutex)
- Разновидность – *семафор*

В Java каждый объект/метод, использующий синхронизацию, фактически моделирует монитор.

Синхронизация объекта

```
public void shoppingWithBonuses(int bonuses) {
    synchronized (bonusCard) {
        if (bonusCard.getBonuses() >= bonuses) {
            System.out.println(who + " is gonna buy something.");
            if (bonusCard.use(bonuses))
                System.out.println(who + " bought something.");
        } else {
            System.out.println("Sorry, Honey, no money.");
        }
    }
}
```

Запуск

husband is gonna buy something.
143 left.
husband bought something.
wife is gonna buy something.
136 left.
wife bought something.
wife is gonna buy something.
129 left.
wife bought something.
wife is gonna buy something.
122 left.
wife bought something.

События «узнать состояние карты» и «купить» **неразрывны**.

Синхронизация метода

- Если весь функционал, который надо синхронизировать, находится в одном методе, то можно весь метод сделать синхронизированным с помощью модификатора synchronized.
- Например, метод use:

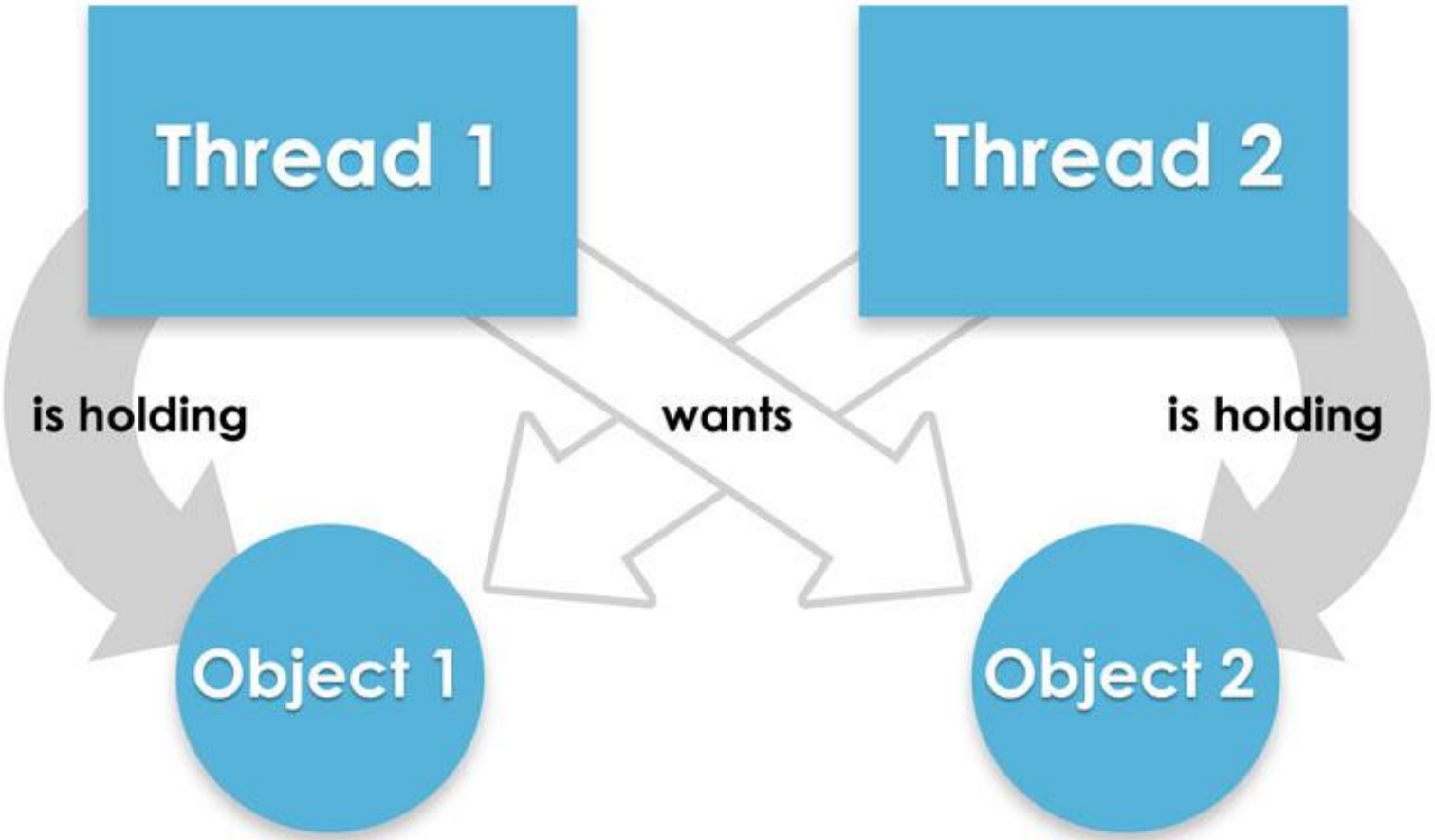
```
public synchronized boolean use(int n) {
    if (bonuses >= n) {
        bonuses -= n;
        System.out.println(bonuses + " left.");
        return true;
    }
    else {
        System.out.println("OOOOOPS");
        return false;
    }
}
```

Thread Safety (Потокобезопасность)

- Код потокобезопасен (thread-safe), если он корректно функционирует при одновременном использовании его в нескольких потоках

- Синхронизация – один из самых грубых способ сделать безопасный код
  - StringBuffer – синхронизированный изменяемый String
  - StringBuilder – несинхронизированный изменяемый String
    - » Быстрее, но ненадежнее

Но не всегда помогает. Deadlock



Картинка с docs.integral.com

Общие переменные между потоками

- Срываем покровы: JRE позволяет каждому потоку кэшировать значение общей переменной (создавать его локальную копию).
  - Главное значение переменной – master-copy
- Значит потоки не всегда читают master-copy при обращении к имени переменной, не всегда записывают master-copy при изменении переменной.
  - Чувствуете проблему?

Проблема

- Потоки меняют локальные копии переменной, не зная об изменениях друг у друга.
    - Общая переменная = 5.
    - Один поток изменил ее на 10.
    - Другой поток продолжает работать с 5 – как-то это неправильно.

volatile

- Модификатор объявление переменной
- «изменчивый»
- Мы явно говорим – эта переменная может меняться несколькими потоками! Нужно поддерживать ее актуальное значение!
  - Т.е. всегда читать master-copy, всегда записывать в master-copy

Что на самом деле означает volatile

1. Операции чтения и записи переменной – *атомарные*!
   - Нельзя вклиниться другому потоку в них
   - Исходно – не атомарны, т.к. отдельно работаем с локальным кэшем, и с общей памятью

2. Атомарность других операций не обеспечивается:
   - ++var;
   - var – volatile, но нет гарантии, что между чтением и ++ или между ++ и записью не вклинится другой поток.
     - Решение?

Синхронизация VS volatile

- Синхронизация сильнее volatile:
  – *атомизирует* весь блок синхронизации
  – работает с общей памятью напрямую

- volatile похож на синхронизацию, но только похож.

Модель «Producer–Consumer»

- Два объекта
  - Producer – производит некоторое действие, результатом которого пользуется Consumer
  - Consumer – забирает произведенный результат Producer-а, когда он готов.
- Примеров много -

Проблема – постоянный опрос Consumer-ом Producer-а

- «Ну что там, готово, нет?» – и так все время
- На уровне программирования это пустые циклы вроде:

```
while (!product.isReady());
```

которые сильно грузят процессор.

Product

```
class Product {
    private boolean status = false;
    public boolean isReady() {
        return status;
    }
    public boolean isUsed() {
        return !status;
    }
    public void produce() {
        // producing
        this.status = true;
    }
    public void use() {
        // using
        this.status = false;
    }
}
```

Producer

```
class Producer extends Thread {

    private Product product;
    public Producer(Product p) {
        this.product = p;
    }

    public void run(){
        while (true) {
            while (!product.isUsed());
            product.produce();
        }
    }
}
```

Consumer

```
class Consumer extends Thread {

    private Product product;
    public Consumer(Product p) {
        this.product = p;
    }

    public void run(){
        while (true) {
            while (!product.isReady());
            product.use();
        }
    }
}
```

Object?

Object

- wait()
  - Поток, вызвавший wait, попадает в список ожидания на этом объекте.
- notify()
  - Поток, вызвавший notify, будит из ожидания один из потоков, попавших в список ожидания.
- notifyAll()
  - Будит всех. Кто-то захватывает управление.

Producer

```
class Producer extends Thread {
    …
    public void run() {
        while (true) {
            synchronized (product) {
                while (!product.isUsed()) {
                    try {
                        // wait временно снимает блокировку
                        product.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                product.produce();
                product.notify();
            }
        }
    }
}
```

Consumer

```
class Consumer extends Thread {
    …
    public void run() {
        while (true) {
            synchronized (product) {
                while (!product.isReady()) {
                    try {
                        product.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                product.use();
                product.notify();
            }
        }
    }
}
```

This is the base

- But it is just small part of multithreading

  ! volatile

  ! **java.util.concurrent**

  http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

Volatile

- Модификатор переменной

volatile int x;

volatile-переменные всегда читаются потоками перед использованием.

  - То есть не возникает ситуации, что поток #1 прочитал значение x, передал управление потоку #2, который изменил значение x, а затем первый поток продолжил использовать старое значение x

  - Похоже на синхронизацию, но есть разница
  - **! Вторая ссылка в конце презентации**

java.util.concurrent

- Пакет для построения крутых многопоточных потокобезопасных приложений!
  - Синхронизаторы
  - Исполнители
  - Коллекции
  - и др.

Синхронизаторы

- Классы, расширяющие возможности мониторов
  - Semaphore
  - CountDownLatch
  - CyclicBarrier
  - Exchanger

Semaphore

- Монитор на несколько потоков
- Semaphore sem = new Semaphore(N);
- Передаем sem в потоки
- Поток вызывает sem.acquire()
  - Останавливается, если количество потоков, вызвавших acquire и не отпустивших его = N
  - Не останавливается, если количество потоков < N.
- Когда поток заканчивает работу с монитором, вызывается sem.release()

CountDownLatch

Поток ждет, пока не выполнится несколько событий

- CountDownLatch cdl = new CountDownLatch(N)
- *Передаем cdl в поток T*
- Вызов cdl.await() погружает поток T в ожидание
- Другие потоки в это время могут вызывать cdl.countDown(). Когда количество вызовов будет равно N, исходный поток T продолжит работу.

CyclicBarrier

Ждет, пока несколько потоков не достигнут какого-то этапа:

- См. игровые комнаты на N человек.

CyclicBarrier cb = new CyclicBarrier(N)

Потоки, владеющие cb, вызывают cb.await() и ждут. Когда количество вызвавших будет равно N, все они продолжат работу

Exchanger

Exchanger<T> - синхронизатор для обмена данными

- Один поток готовит данные, вызывает у объекта Exchanger метод exchange(T)
- Второй поток, вызывая данный метод у того же объекта Exchanger, получает данные как результат метода exchange.

Хотим!

- Чтобы потоки возвращали значение!
- Но! Мы не знаем, когда закончится поток!
- Но можно планировать то, что будет сделано, когда значение будет вычислено.

3 класса

- **ExecutorService** – умеет создавать и запускать потоки, управляя их выполнением.
- **Callable**<T> – поток, возвращающий значение типа T в будущем.
- **Future**<T> - будущее значение Callable типа T.

Пример

```
public class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        Thread.sleep(1000);
        return Thread.currentThread().getName();
    }
}
```

Пример

```
public static void main(String args[]) {
    ExecutorService executor = Executors.newFixedThreadPool(5);
    List<Future<String>> list = new ArrayList<Future<String>>();
    Callable<String> callable = new MyCallable();
    for (int i = 0; i < 100; i++) {
        Future<String> future = executor.submit(callable);
        list.add(future);
    }
    for (Future<String> fut : list) {
        try {
            System.out.println(new Date() + "::" + fut.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    executor.shutdown();
}
```

Пример

```
public static void main(String args[]) {
    ExecutorService executor = Executors.newFixedThreadPool(5);
    List<Future<String>> list = new ArrayList<Future<String>>();
    Callable<String> callable = new MyCallable();
    for (int i = 0; i < 100; i++) {
        Future<String> future = executor.submit(callable);
        list.add(future);
    }
    for (Future<String> fut : list) {
        try {
            System.out.println(new Date() + "::" + fut.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    executor.shutdown();
}
```

Concurrent Коллекции. Пример - BlockingQueue, BlockingDeque

– Если поток пытается читать данные из пустой BlockingQueue (BlockingDeque), он приостанавливается до появления данных в этой коллекции.

– Аналогично, если пытается записать данные в BlockingQueue (BlockingDeque), которая переполнена (записано максимально возможное число элементов) – ждет освобождения места для записываемых данных.

Atomic

- Типы данных, операции с которыми являются атомарными:
  - AtomicInteger
  - AtomicLong
  - AtomicIntegerArray
  - …
- Вызываемые у них методы set, get и другие ведут себя в лучших традициях синхронизации и volatile – *атомизированно*.
  - Будучи общими, могут отслеживать свои изменения разными потоками (например, ! compareAndSet)

Словарь

deadlock, join, notify, notifyAll, Runnable, sleep, Thread, Thread Safety, wait, синхронизация

Прочитать

- http://habrahabr.ru/post/164487/ (rus)
- http://habrahabr.ru/post/108016/ (rus)
- http://www.quizful.net/post/java-threads (rus)
- https://docs.oracle.com/javase/tutorial/essential/concurrency/ (eng) - java.util.concurrent
- http://www.javaworld.com/article/2077138/java-concurrency/introduction-to-java-threads.html (eng)
- http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html (eng)
- http://www.tutorialspoint.com/java/java_multithreading.htm (eng)