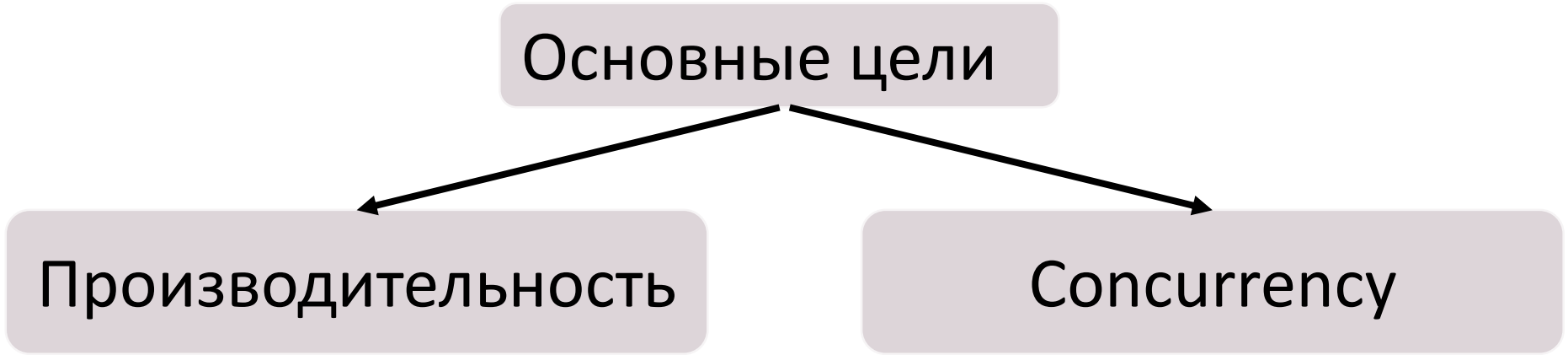
Multithreading

Многопоточность – это принцип построения программы, при котором несколько блоков кода могут выполняться одновременно.



Варианты создания нового потока

```
//Создание
class MyThread extends Thread{   public void run() {   код   }  }
//Запуск
new MyThread().start();
```

```
//Создание
class MyRunnableImpl implements Runnable{  public void run() { код } }
//Запуск
new Thread(  new MyRunnableImpl()  ).start();
```

Из за того, что в Java отсутствует множественное наследование, чаще используют 2-ой вариант.

Методы Thread

setName

getName

setPriority

getPriority

sleep

join

Concurrency / Parallelism
Asynchronous / Synchronous

Concurrency означает выполнение сразу нескольких задач. В зависимости от процессора компьютера concurrency может достигаться разными способами.

Parallelism означает выполнение 2-х и более задач в одно и то же время, т.е. параллельно. В компьютерах с многоядерным процессором concurrency может достигаться за счёт parallelism.

В синхронном программировании задачи выполняются последовательно друг за другом.

В асинхронном программировании каждая следующая задача НЕ ждёт окончания выполнения предыдущей. Асинхронное программирование помогает достичь concurrency.

Ключевое слово volatile

Ключевое слово volatile используется для пометки переменной, как хранящейся только в основной памяти «main memory».

Для синхронизации значения переменной между потоками ключевое слово volatile используется тогда, когда только один поток может изменять значение этой переменной, а остальные потоки могут его только читать.

Data race и synchronized методы

Data race – это проблема, которая может возникнуть когда два и более потоков обращаются к одной и той же переменной и как минимум 1 поток её изменяет.

Пример метода:

```
public synchronized void abc() {   method body   }
```

Понятие «монитор» и synchronized блоки

Монитор – это сущность/механизм, благодаря которому достигается корректная работа при синхронизации.
В Java у каждого класса и объекта есть привязанный к нему монитор.

Пример блока:

```
static final Object lock = new Object();
public void abc() {          method body
synchronized(lock)  {    block body   }  method body  }
```

Методы wait и notify

Для извещения потоком других потоков о своих действиях часто используются следующие методы:

wait - освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify();

notify – НЕ освобождает монитор и будит поток, у которого ранее был вызван метод wait();

notifyAll – НЕ освобождает монитор и будит все потоки, у которых ранее был вызван метод wait();

Возможные ситуации в многопоточном программировании

Deadlock – ситуация, когда 2 или более потоков залочены навсегда, ожидают друг друга и ничего не делают.

Livelock – ситуация, когда 2 или более потоков залочены навсегда, ожидают друг друга, проделывают какую-то работу, но без какого-либо прогресса.

Lock starvation – ситуация, когда менее приоритетные потоки ждут долгое время или всё время для того, чтобы могли запуститься.

Lock и ReentrantLock

Lock – интерфейс, который имплементируется классом ReentrantLock.

Также как ключевое слово synchronized, Lock нужен для достижения синхронизации между потоками.

lock()

unlock()

tryLock()

Daemon потоки

Daemon потоки предназначены для выполнения фоновых задач и оказания различных сервисов User потокам.

При завершении работы последнего User потока программа завершает своё выполнение, не дожидаясь окончания работы Daemon потоков.

setDaemon()

isDaemon()

Прерывание потоков

У нас есть возможность послать сигнал потоку, что мы хотим его прервать.
У нас также есть возможность в самом потоке проверить, хотят ли его прервать. Что делать, если данная проверка показала, что поток хотят прервать, должен решать сам программист.

interrupt()

isInterrupted()

Thread pool и ExecutorService

Thread pool – это множество потоков, каждый из которых предназначен для выполнения той или иной задачи.

В Java с thread pool-ами удобнее всего работать посредством ExecutorService.

Thread pool удобнее всего создавать, используя factory методы класса Executors:

Executors.newFixedThreadPool(int count) – создаст pool с 5-ю потоками;

Executors.newSingleThreadExecutor() – создаст pool с одним потоком.

Thread pool и ExecutorService

Метод **execute** передаёт наше задание (task) в thread pool, где оно выполняется одним из потоков.

После выполнения метода **shutdown** ExecutorService понимает, что новых заданий больше не будет и, выполнив поступившие до этого задания, прекращает работу.

Метод **awaitTermination** принуждает поток в котором он вызвался подождать до тех пор, пока не выполнится одно из двух событий: либо ExecutorService прекратит свою работу, либо пройдёт время, указанное в параметре метода awaitTermination.

ScheduledExecutorService

ScheduledExecutorService мы используем тогда, когда хотим установить расписание на запуск потоков из пула.

Данный pool создаётся, используя factory метод класса Executors:
Executors.newScheduledThreadPool(int count)

schedule

scheduleAtFixedRate

scheduleWithFixedDelay

Интерфейсы Callable и Future

Callable, также как и Runnable, представляет собой определённое задание, которое выполняется потоком.
В отличии от Runnable Callable:

- имеет return type не void;
- может выбрасывать Exception.

Метод **submit** передаёт наше задание (task) в thread pool, для выполнения его одним из потоков, и возвращает тип Future, в котором и хранится результат выполнения нашего задания.

Метод **get** позволяет получить результат выполнения нашего задания из объекта Future.

Semaphore

Semaphore – это синхронизатор, позволяющий ограничить доступ к какому-то ресурсу. В конструктор Semaphore нужно передавать количество потоков, которым Semaphore будет разрешать одновременно использовать этот ресурс.

acquire

release

CountDownLatch

CountDownLatch – это синхронизатор, позволяющий любому количеству потоков ждать пока не завершится определённое количество операций. В конструктор CountDownLatch нужно передавать количество операций, которые должны завершится, чтобы потоки продолжили свою работу.

await

countDown

getCount

Exchanger

Exchanger – это синхронизатор, позволяющий обмениваться данными между двумя потоками, обеспечивает то, что оба потока получат информацию друг от друга одновременно.

exchange

AtomicInteger

AtomicInteger – это класс, который предоставляет возможность работать с целочисленным значением int, используя атомарные операции.

- incrementAndGet
- getAndIncrement
- addAndGet
- getAndAdd
- decrementAndGet
- getAndDecrement

Коллекции для работы с многопоточностью

Synchronized collections

Получаются из традиционных коллекций благодаря их обёртыванию

Concurrent collections

Изначально созданы для работы с многопоточностью

Collections.synchronizedXYZ(коллекция)

ConcurrentHashMap

ConcurrentHashMap имплементирует интерфейс ConcurrentMap, который в свою очередь происходит от интерфейса Map.

В ConcurrentHashMap любое количество потоков может читать элементы не блокируя его.

В ConcurrentHashMap, благодаря его сегментированию, при изменении какого-либо элемента блокируется только bucket, в котором он находится.

В ConcurrentHashMap ни key, ни value не могут быть null.

CopyOnWriteArrayList

CopyOnWriteArrayList имплементирует интерфейс List.

CopyOnWriteArrayList следует использовать тогда, когда вам нужно добиться потокобезопасности, у вас небольшое количество операций по изменению элементов и большое количество по их чтению.

В CopyOnWriteArrayList при каждой операции по изменению элементов создаётся копия этого листа.

ArrayBlockingQueue

ArrayBlockingQueue – потокобезопасная очередь с ограниченным размером (capacity restricted).

Обычно один или несколько потоков добавляют элементы в конец очереди, а другой или другие потоки забирают элементы из начала очереди.

put

take