

# Многопоточность

## Java Development Kit



# Оглавление

<b>Введение</b>	<b>3</b>
<b>Термины, используемые в лекции</b>	<b>3</b>
<b>Многопоточность</b>	<b>3</b>
Создание и управление потоками	5
Синхронизация	7
<b>Java memory model</b>	<b>8</b>
Race condition	8
Valiotil	8
Атомарные переменные	8
Неизменяемые объекты	9
ThreadLocal	10
<b>Примитивы синхронизации</b>	<b>10</b>
Semaphore	10
Exchanger	11
CowntDownLatch	11
CyclicBarrier	12
ReentrantLock	12
Коллекции в многопоточной среде	14
Синхронизированные коллекции	14
List	14
Set	15
Map	15
Queue	16
<b>Подведем итоги</b>	<b>17</b>
<b>Подготовка к семинару</b>	<b>17</b>
<b>Что можно почитать еще?</b>	<b>17</b>
<b>Используемая литература</b>	<b>17</b>

# Введение

На предыдущей лекции мы познакомились с Java Collection Framework и с основными интерфейсами для организации хранения списков объектов (List, Set, Queue, Map) а также с основными соответствующих реализациями.

Сегодня мы будем говорить с вами о многопоточности. На текущий момент времени вы разрабатывали только однопоточные приложения — весь код выполняется последовательно шаг за шагом. Многопоточность же позволяет нам решить сразу две основные задачи:

- ☐ Одновременное выполнение нескольких не связанных действий;
- ☐ Параллельные вычисления в рамках одной задачи.

В рамках текущего урока мы рассмотрим механизмы создания потоков, основные примитивы синхронизации и принципы создания потокобезопасных приложений. Рассмотрим способы организации памяти Java и как это влияет на выполнение многопоточного приложения.

## Термины, используемые в лекции

**CPU (central processing unit)** — центральное обрабатывающее устройство, часто просто процессор;

**Многопоточность (Multithreading)** — свойство платформы или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

**Deadlock** — взаимная блокировка двух потоков

**Race Condition** — состояние гонки, при котором потоки выполняются в неверном порядке

**Volatile** — инструкция, запрещающая копировать переменную в stack процессора

**ThreadLocal** — переменная, уникальная для каждого потока

**UX (user experience)** — восприятие и ответные действия пользователя, возникающие в результате использования продукции, системы или услуги.

## Многопоточность

Для начала давайте дадим определение понятию многопоточного приложения. Любое java приложение состоит из одного процесса и нескольких потоков (до этого момента мы с вами писали приложения состоящие только из одного потока).

Давайте определим назначение процесса:

- Каждый процесс состоит данных и кода, который их обрабатывает. Сам процесс создается операционной системой при запуске приложения, что является ресурсоемкой задаче. Также процесс обладает собственным виртуальным адресным пространством.
- Процессы работают независимо друг от друга, т.е. не имеют прямого доступа к данным других процессов.
- Операционная система выделяет ресурсы (память, время выполнения) для процесса.
- Если один из запущенных процессов заблокирован, то ни один другой процесс не может выполняться, пока он не будет снята блокировка.
- Процесс может инициировать создание и контролировать дочерние процессы, но он не может управлять процессами того же уровня.

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

Любой процесс состоит из потоков. Потоки (также их можно назвать легковесными процессами) выполняются параллельно друг с другом — они используют адресное пространство процесса, которое делят между собой.

Если говорить о многопоточности, то это одновременное выполнение двух или более потоков, с целью использования центрального процессора (CPU — central processing unit) максимально эффективно. Каждый поток работает параллельно и не требует отдельной области памяти, к тому же, переключение контекста между потоками занимает меньше времени. Многопоточность применяется для следующих целей:

- Эффективное использование одного CPU несколькими потоками: допустим один из запущенных потоков обрабатывает http запрос , в это время другой поток может использовать CPU для выполнения своих задач. Если же у сервера несколько процессоров (или несколько ядер), многопоточность позволяет запускать потоки на отдельных ядрах CPU.
- Оптимальное использование нескольких центральных процессоров (или несколько ядер одного процессора): многопоточность позволяет запускать потоки на отдельных ядрах CPU, чтобы задействовать все процессоры или и ядра - один поток может запускаться максимум на одном CPU.
- Улучшенный UX в плане скорости ответа на запрос: допустим, пользователь нажимает на элемент в графическом интерфейсе (это действие направляет запрос по сети серверу) — вопрос, какой поток выполняет этот запрос. В однопоточном приложении, этот же поток будет отвечать за любой вид взаимодействия пользователя с приложением - это значит, что пока запрос не будет обработан, мы не сможем продолжить взаимодействие с программой. В многопоточном приложении мы можем создать фоновый поток, который будет обрабатывать запрос, в то время как поток отвечающий за интерфейс мог продолжить реагировать на другие действия.
- Улучшенный UX в плане распределения ресурсов: т.е. справедливое распределение ресурсов сервера между пользователями. Представьте web приложение, которое принимает запросы от клиентов, при этом используется один поток для обработки всех запросов. Пока этот поток обрабатывает один из запросов пользователя, остальные будут находиться в очереди, ожидания завершения обработки предыдущего. В многопоточном приложении, ни один из пользовательских запросов, выполняемый собственным потоком, не сможет полностью захватить вычислительные ресурсы процессора.

Потоки процесса могут находиться в следующих состояниях:

- New – экземпляр потока создан, но он не запущен.
- Running — экземпляр потока запущен, процессор выделил ресурсы и начинается его выполнение. Во время выполнения потока может перейти в состояние Runnable, Dead или Blocked.
- Suspended — запущенный поток приостанавливает свою работу (выполнение потока можно возобновить) - поток продолжит работать с того места, где его он был остановлен.
- Blocked — поток ожидает высвобождения ресурсов или завершения операции ввода-вывода. Находясь в этом состоянии поток не потребляет процессорное время.

- Terminated — поток немедленно завершает свою работу (выполнение потока нельзя возобновить). Поток может быть завершен, если код потока был выполнен (т.е. поток выполнил свою задачу) или если во время выполнения потока было выброшено необработанное исключение.
- Dead — финальное состояние жизненного цикла потока: после того, как поток завершил свое выполнение, его состояние меняется на dead.

## Создание и управление потоками

### Создание и запуск потока

И так мы познакомились с основными определениями используемыми для описания многопоточности. Теперь давайте посмотрим как создать новый поток в Java. Существует два основных способа:

- Создать класс и имплементировать интерфейс Runnable. Нам необходимо реализовать единственный метод run при имплементации интерфейса Runnable, который должен содержать код, выполняющийся в потоке. Объект Runnable передается конструктору класса Thread. Этот способ считается наиболее гибким и применяется для высокоуровневых API управления потоками.

```
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 }
```

- Использовать подкласс Thread. Класс Thread сам имплементирует интерфейс Runnable: обратите внимание его метод run не выполняет никакой работы (необходимо переопределить поведение метода в наследнике). Можно объявить подкласс Thread, предоставляя собственную реализацию метода run. Данный способ больше подходит для “простых” приложений.

```

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 }

```

Обратите внимание, что новый поток создается только если вызвать метод `start`, который в свою очередь вызывает метод `run` - при попытке вызвать метод `run` напрямую, код выполняется в том же потоке. Так же при запуске JVM, создается главный поток с именем `main` и еще несколько служебных потоков.

## Остановка потока

Теперь мы умеем создавать и запускать потоки, но как мы можем выбранный поток остановить? Вы можете обратить внимание, что класс `Java Thread` содержит метод `stop()` (помечен как `deprecated`, т.е. не рекомендованный к использованию). Метод `stop()` не дает гарантий относительно состояния, в котором поток остановили. Таким образом, все объекты `Java`, к которым у потока был доступ во время его выполнения, останутся в неизвестном состоянии (изменения состояния объектов не будут зафиксированы и видны другим потокам), что может привести к ошибкам — например при выполнении потока создается ресурс (например подключение к базе данных), который не будет закрыт и как следствие мы получим утечку памяти.

Вместо `deprecated` метода `stop()` вам следует использовать `interrupt()`. Если метод `stop()`, принудительно останавливал поток, то `interrupt()` только предлагает потоку остановить свое выполнение путем установки флага `interrupted` в `true` внутри потока — решение об остановке принимает сам код потока. Данный флаг отображает статус прерывания (значение по умолчанию `false`). Если поток прерывается другим потоком то происходит следующее:

- ☐ Если поток ожидает выполнения прерываемого метода блокирования (`Thread.sleep()`, `Thread.join()` или `Object.wait()`), то процесс ожидание

прерывается и выбрасывает `InterruptedException` — после этого флаг `interrupted` устанавливается в значение `false`.

```
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 }
```

- ☐ Если поток не ожидает выполнения прерываемого метода блокирования, флаг `interrupted` устанавливается в значение `true` — теперь код потока должен обработать переменную в реализации метода `run`

```
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 }
```

Также любой поток может остановиться сам — для этого необходимо вызвать `static` метод `Thread.sleep()` (самый простой способ взаимодействия с другими потоками). В операционной системе, с установленной JVM, имеется свой планировщик потоков, называемый `Thread Scheduler`. Данный планировщик принимает решение, какой поток и когда необходимо запускать. Метод `Thread.sleep()` может принимать в качестве параметра количество миллисекунд — время на которое поток попытается заснуть, возобновлением выполнения — обратите внимание, что абсолютная точность не гарантирована.



```

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 }

```

## Очередность выполнения потоков

Теперь давайте представим, что в рамках решения задачи нам необходимо гарантировать, что поток не начнет свое выполнения пока не будет завершен другой поток. Для этого используется метод `join()` экземпляра класса `Thread` — он объединяет начало выполнения одного потока с завершением выполнения другого. Если метод `join()` вызывается на одном из потоков, то текущий `Thread` выполняющийся в этот момент блокируется до момента времени, пока поток, для которого вызван метод `join` не закончит свое выполнение.

Метод `join()` может (не обязательно) принимать в качестве параметра количество миллисекунд — количество времени ожидания. Если в качестве значения времени ожидания указать 0, то такой поток будет «ждать вечно».

```

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 }

```

Также обратите внимание на статический метод `Thread.yield()`, который заставляет CPU переключиться на обработку других потоков системы. Метод может быть

полезным, если, когда поток, для которого вызван метод `yield()` ожидает наступления события, а проверка наступления происходила как можно чаще.



#### **Посмотрите демонстрацию в видеолекции:**

- Создание и запуск потоков;
- Демонстрация API класс Thread.

### **Пул потоков**

Процесс создания новых потоков и освобождение ресурсов являются дорогостоящей операцией. Мы можем изначально определить необходимое количество потоков, создать их и использовать для решения задач — в java для этого используются пулы потоков и очереди задач, из которых выбираются задачи для потоков. Пул потоков — это по сути контейнер, в котором находятся потоки, и после выполнения одной из задач они самостоятельно переходить к следующей.

Вы можете использовать такой контейнер для контроля создания и управления потоками — это экономит ресурсы связанные с процессом создания новых потоков. Рассмотрим классы и интерфейсы, которые отвечают за создание и управление пулом потоков (Executor Framework in Java):

- ☐ Интерфейс `Executor`. Объекты, которые реализуют интерфейс `Executor`, могут выполнять `Runnable`-задачу (Интерфейс имеет один метод `void execute(Runnable command)`).
- ☐ `ExecutorService`. Интерфейс `ExecutorService` наследуется от интерфейса `Executor` и предоставляет возможности для выполнения заданий `Callable`.
- ☐ Класс `Executors`. Утилитарный класс `Executors` создает классы, которые реализуют интерфейсы `Executor` и `ExecutorService`.

Теперь давайте посмотрим на основные реализации интерфейсов `Executor` и `ExecutorService`:

- `ThreadPoolExecutor` — пул потоков содержит фиксированное количество потоков - количество потоков определяется через конструктор.
- `Executors.newCachedThreadPool()` - возвращает пул потоков, если количество потоков в пуле недостаточно, то в нем будет создан новый поток.

- `Executors.newSingleThreadExecutor()` — пул потоков, который гарантирует, что в нем может быть только один поток.
- `ScheduledThreadPoolExecutor` — пул потоков используется для запуска периодических задач или задач, которые должны запуститься только раз по истечении некоторого промежутка времени.



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация работы с пулом потоков

## Синхронизация

Итак, мы научились создавать и запускать потоки. Теперь давайте обсудим, что происходит когда два и более потока начинают конкурировать за ресурсы (т.е. пытаются в параллели получить доступ к одному и тому же объекту или ресурсу).

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу или подключение к базе данных. Если одновременно несколько потоков обратятся к такому ресурсу на запись (т.е. с целью изменения состояния), то результаты выполнения программы будут сложно предсказуемыми или это может привести к ошибке выполнения.

```
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 }
```

Попробуйте несколько раз запустить данный код, чтобы убедиться, что последовательность запуска потоков отличается каждый раз.

Самым простым способом синхронизировать потоки (т.е. определить их поведение при работе с общим объектом) — это концепция «монитора» и ключевое слово `synchronized` (обратите внимание, что у любого наследника класса `Object` есть свой собственный «монитор» — именно поэтому нельзя синхронизировать примитивные типы). Монитор характеризуется следующей информацией:

- состоянии (`locked`) — признак, что монитор захвачен потоком;
- владелец (`owner`) — каким потоком захвачен монитор в текущий момент;
- перечень потоков, которые не смогли захватить монитор (`blocked set`), так как монитор захвачен другим потоком;
- перечень потоков у которых был вызван метод `wait` (`wait set`).

Для синхронизации потоков используется ключевое слово `synchronized` (обратите внимание, что всегда синхронизируется именно объект — только у него есть монитор). Давайте посмотрим, что происходит, когда поток пытается захватить монитор объекта:

- Поток попадает в синхронизированный блок кода (`synchronized` блок);
- В синхронизированном объекте проверяются переменные `locked` и `owner` монитора.
- 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 начал выполнять код (соответствует открывающей фигурной скобке `synchronized` блока)
- Поток завершил выполнение кода (соответствует закрывающейся фигурной скобке блока синхронизации)
- Переменные `locked` и `owner` монитора очищаются.

Теперь давайте посмотрим как мы можем синхронизировать соответствующий объект. Во-первых, можно синхронизировать методы самого класса. В этом случае объектом синхронизации является сам объект — `this`:

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

Во вторых, можно синхронизировать другой объект в целевом 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 выполнился 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    }
22 }

```

В третьих, в случае синхронизации статических методов мы используем монитор самого класса:

```

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

```

Примечание. Почему же тогда просто не сделать все методы синхронизированными? Проблема в том, что синхронизация и переключение между потоками ресурсоемкая операция (время выполнения увеличивается и приложение начинает работать медленнее), поэтому использовать данные механизмы нужно **осторожно, только там, где между потоками существует конкуренция за ресурсы**.

Более того, неверное проектирование многопоточного исполнения программ, может привести к Deadlock. Deadlock (взаимная блокировка потоков) — это ошибка в многопоточном программировании, которая происходит когда несколько потоков имеют циклическую зависимость от пары синхронизированных объектов. Пусть один thread входит в монитор объекта A, а другой — объекта B. Если thread в объекте A пытается вызвать любой синхронизированный метод объекта B, а объект



В в то же самое время пытается вызвать любой синхронизированный метод объекта А, то потоки будут заблокированы в процессе ожидания “навечно”.

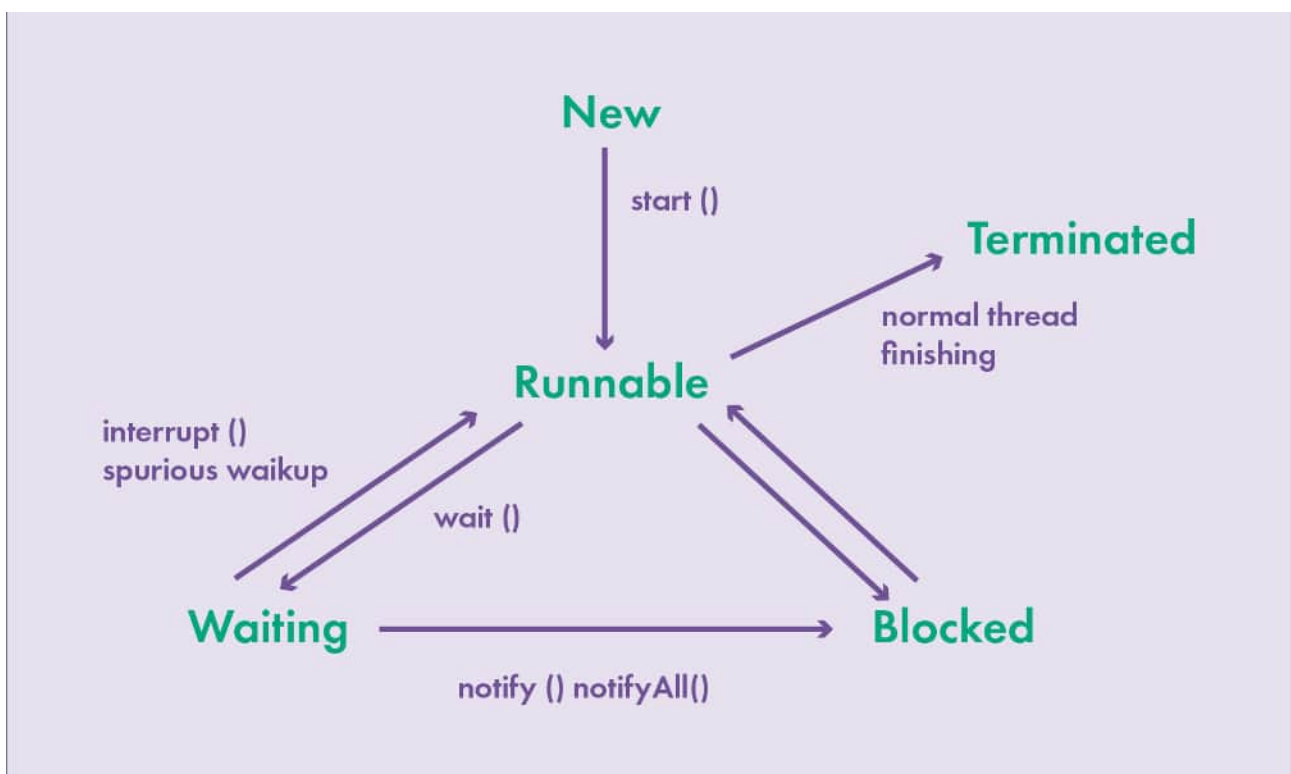
```
1 public static void main(String[] args) {
2     ObjectToLock objectToLockOne = new ObjectToLock();
3     ObjectToLock objectToLockTwo = new ObjectToLock();
4     getThread(objectToLockTwo, objectToLockOne);
5     getThread(objectToLockOne, objectToLockTwo);
6 }
7
8 private static void getThread(ObjectToLock 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 }
```

И так мы теперь можем создать и запустить поток, а также синхронизировать потоки при использовании общих ресурсов, но в процессе работы приложения, довольно часто могут возникать ситуации, что поток ожидает события, необходимого для продолжения выполнения своей работы (например ответ от web сервера на http запрос) — такой поток блокирует работу всего приложения — в этом случае логичным решением является уступить ресурс, другому потоку, который в текущий момент времени может выполнять свои задачи. У класса Object (а значит у

всех не примитивных типов), есть следующие методы, которые позволяют управлять переключением потока:

- `wait()` — После вызова этого метода поток попадает в `wait set` монитора, сам же монитор освобождается (переменные `locked` и `owner` в мониторе очищаются)
- `notify()` — Для того чтобы потоки, которые находятся в `wait set`, продолжили свое выполнение, другой поток должен захватить монитор и вызвать методы `notify()`. После вызова метода `notify()` из `wait set` выбирается произвольный поток и переводится в `blocked set`. После того как этот поток выйдет из `synchronized` блока, нотифицированные потоки будут по одному захватывать монитор и продолжать выполнение
- `notifyAll()` — аналогично `notify()`, но все потоки из `wait set` переводятся в `blocked set`

На схемы мы видим переходы потока из одного состояния в другое.



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация проблемы конкуренции;
- Демонстрация применения синхронизации.

- Демонстрация Deadlock

## Java memory model

Без преувеличения многопоточность одна из самых сложных и комплексных тем в программировании. Пока мы познакомились, только с базовыми механизмами создания и управления потоками, но прежде чем перейти к изучению паттернов и механизмов, применяемых в многопоточном программировании, нам необходимо понять, причины не очевидного поведения таких программ. Для этого нам предстоит узнать как Java работает с памятью и как это влияет на многопоточность.

Java Memory Model, JMM или модель памяти Java описывает поведение потоков в среде исполнения Java. Модель памяти — это часть семантики языка Java, которая определяет на что может и на что не должен рассчитывать разработчик при работе с потоками. Разумеется JMM посвящена большая глава в спецификации Java - сегодня мы рассмотрим только основные моменты.

Первое что нам необходимо понять, как Java структурирует выделенную ей для работы память. JVM использует следующие типы памяти:

- ☐ Heap – это регион памяти, где хранятся объекты Java. Куча является общей для всех потоков, ее содержимое управляется сборщиком мусора.

Куча содержит все объекты, созданные в вашем приложении, независимо от того, какой поток создал объект (к этому относятся и обертки примитивных типов).

- ☐ Stack – это область памяти, где хранятся локальные переменные и стек вызовов методов. Для каждого потока создается отдельный стек в JVM.

Стек содержит все локальные переменные для каждого метода. Соответствующий поток может получить доступ только к своему стеку - локальные переменные (т.е. переменные инициализированные в методе), невидимы для других потоков, кроме потока, который их создал. Представим, что два потока выполняют один и тот же код, т.е. вызвали один и тот же метод, всё равно все локальные переменные, будут созданы своих собственных стеках. Таким образом, каждый поток имеет свою версию локальной переменной соответствующего метода.

- Все локальные переменные примитивных типов (boolean, byte, short, char, int, long, float, double) полностью хранятся в стеке потоков и не видны другим потокам.



- Локальная переменная также может быть ссылкой на объект. В этом случае ссылка (локальная переменная) хранится в стеке потоков, но сам объект хранится в куче.
- Статические переменные класса также хранятся в куче вместе с определением класса.

К объектам в куче могут обращаться любые потоки, имеющие ссылку на соответствующий объект. Когда поток имеет доступ к объекту, он также может получить доступ к переменным-членам этого объекта. Если два потока вызывают метод для одного и того же объекта одновременно, они оба будут иметь доступ к переменным-членам объекта, но каждый поток будет иметь свою собственную копию локальных переменных..

- ☐ 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.

Итак мы разобрали, как Java управляет памятью, теперь давайте обсудим, что может произойти, когда объекты и переменные хранятся в различных областях памяти – возникают следующие проблемы:

- ☐ Видимость изменений, которые произвел поток над общими переменными.

Пусть два или более потока работают с общим объектом (без использования volatile-объявления или синхронизации), то изменения в этом объекта, сделанные одним из потоком, могут быть невидимы для други.

Представьте: общий объект изначально хранится в Heap. Поток, выполняющийся на CPU, считывает общий объект в кэш этого же CPU. Там он вносит изменения в объект. Пока кэш CPU не был сброшен в основную память, измененная версия общего объекта не видна другим потокам. Таким образом, каждый поток может получить свою собственную копию общего объекта, каждая копия будет находиться в отдельном кэше CPU.

- ☐ Состояние гонки при чтении, проверке и записи общих переменных.

Пусть два или более потоков совместно используют один объект и более одного потока одновременно меняют состояние этого объекта (т.е. обновляют переменные), тогда может возникнуть состояние гонки или race condition. По сути

это нарушения алгоритма выполнения задач потоками - один поток начинает свое выполнение раньше, чем это необходимо.

Представьте: поток X считывает переменную объекта в кэш своего процессора. Представьте также, что поток Y делает то же самое, но в кэш другого процессора. Теперь поток X прибавляет 1 к значению переменной count, и поток Y делает то же самое. Теперь переменная была увеличена дважды.

Если бы эти операции были выполнены последовательно (как и было задумано), переменная была бы увеличена дважды.

Тем не менее, обе операции были выполнены одновременно без использования синхронизации. Независимо от того, какой из потоков, записывает свою версию переменной в основную память, новое значение будет только на 1 больше исходного.



**Посмотрите демонстрацию в видеолекции:**

- Видимость изменений
- Race condition

## Volatile

Ключевое слово `volatile` указывает, что взаимодействие потоков с этой переменной должно происходить минуя кэш процессора, т. е. напрямую (т.е. запрещено копировать переменную из Heap).

Когда потоки используют переменные (не `volatile`), они могут копировать значение этих переменных в кэш CPU для улучшения производительности. Если вы используете процессор с несколькими ядрами, при этом каждый из потоков выполняется на отдельном ядре, то одна и та же переменная может находиться в разном состоянии на каждом ядре CPU. В результате мы получим несколько копий одной и той же переменной: копии в кэше каждого ядра процессора и копия переменной в Heap. При использовании не `volatile` переменных нельзя знать наверняка, когда JVM читает значение переменной из главной памяти и когда записывается значение в главную память.

Примечание. Помните, что объявление переменной как `volatile` достаточно, только когда один поток изменяет переменную, а другой поток читает ее значение - Если

два потока одновременно меняют состояние переменной, то `volatile` уже недостаточно — все равно может быть `race condition`.

И так ключевое слово `volatile` гарантирует нам следующее:

- Если поток X пишет в `volatile` переменную, а затем второй поток Y читает значение, тогда все переменные, видимые потоку X перед записью в переменную `volatile`, также будут видны потоку Y после того как он прочитал переменную `volatile`.
- Если поток X читает переменную `volatile`, то все переменные, видимые потоку X при чтении переменной `volatile`, также будут пересчитаны из основной памяти.



**Посмотрите демонстрацию в видеолекции:**

- Применение `volatile`

## Атомарные переменные

И так, теперь давайте рассмотрим ситуацию, когда два или более потоков пытаются изменить общий разделяемый ресурс: одновременно выполнять операции чтения и записи. Чтобы избежать состояния гонки, мы можем использовать `synchronized`-методы, `synchronized`-блоки (`volatile` нам в данном случае не подойдет). Без использования механизма синхронизации, результат может быть непредсказуемым, а значение не будет иметь никакого смысла для нашей программы. Но как было уже сказано выше, процесс синхронизации дорогостоящая операция, требующая существенных ресурсов CPU. Также этот способ блокирующий — одновременно выполняется только один поток, что сильно влияет на производительность системы в целом.

Для решения этой проблемы мы можем использовать неблокирующие алгоритмы.. Эти алгоритмы называются `compare and swap (CAS)` и базируются на том, что современные процессоры поддерживают некоторые операции на уровне машинных инструкций. В Java нам доступны классы атомарных переменных: `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference`.

Алгоритм `compare and swap` работает следующим образом: есть ячейка памяти, текущее значение в ней и то значение, которое хотим записать в эту ячейку. Сначала ячейка памяти читается и сохраняется текущее значение, затем

прочитанное значение сравнивается с тем, которое уже есть в ячейке памяти, и если значение прочитанное ранее совпадает с текущим, происходит запись нового значения. Следует упомянуть, что значение переменной после чтения может быть изменено другим потоком, потому что CAS не является блокирующей операцией.



**Посмотрите демонстрацию в видеолекции:**

- Применение атомарных переменных

## Неизменяемые объекты

И так главная проблема с которой мы сталкиваемся, при работе с несколькими потоками это использование общих ресурсов, которые могут изменить свое состояние (т.е. изменение значений атрибутов объекта) в процессе выполнения. Пожалуй одно из самых эффективных способов решения данной проблемы это использование неизменяемых объектов (механизмы синхронизации просто не нужны). Вы уже встречались с подобными объектами — любая обертка примитивного типа неизменяемый объект. Давайте обсудим принципы, которые позволяют нам проектировать неизменяемые объекты:

- 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`



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация процесса проектирования неизменного класса

## ThreadLocal

Если в рамках реализации задачи, необходимо гарантировать, что каждый поток будет работать со своей уникальной переменной — воспользуйтесь классом `ThreadLocal`.

Класс `ThreadLocal` представляет хранилище тред-локальных переменных. По способу использования он похож на обычную обертку над значением, с методами `get()`, `set()` и `remove()` для доступа к нему, и дополнительным фабричным методом `ThreadLocal.withInitial()`, устанавливающим значение по-умолчанию.

Отличие такой переменной от обычной в том, что `ThreadLocal` хранит отдельную независимую копию значения для каждого ее использующего потока — соответственно работа с такой переменной потокобезопасна.

Таким образом объект класса `ThreadLocal` хранит внутри не одно конкретное значение, а хэш-таблицу (поток и соответствующее значение), и при использовании обращается к значению для текущего потока.

```
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     }
```



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация применения ThreadLocal

## Примитивы синхронизации

И так, коллеги, мы познакомились с некоторыми аспектами работы Java с памятью а также с причинами возникновения типовых проблем (Race Condition, Deadlock, одновременный доступ на запись) с которыми приходится сталкиваться при многопоточном программировании. В этой главе мы рассмотрим многопоточного программирования, призванные решить типовые задачи при работе с потоками.

### Semaphore

Semaphore один из примитивов синхронизации, позволяющий определить N потоков, которым позволено исполнять критическую секцию кода. Когда создается Semaphore, в конструктор передается количество разрешенных потоков (пропуск). Представьте, что вам нужно реализовать приложение для контроля количества свободных мест на парковке - если места на парковке закончились, то остальные машины должны ждать пока место не освободится.

Когда один из потоков попадает в Semaphore - количество допустимых пропусков уменьшается (при освобождении наоборот увеличивается). Вы можете в методе захвата семафора (acquire()) указать, какое количество пропусков возьмет поток, а в методе освобождения (release()) сколько будет возвращено (это количество не может быть больше, чем захваченное количество пропусков).

Если передать в конструктор вторым параметром true, потоки, которые ожидают получение пропуска (хотят войти в критическую секцию), выстраиваются в очередь (вместо blocking set используется очередь по типу FIFO).

```

1 private static Integer count = Integer.valueOf(0);
2
3 public static void main(String[] args) {
4
5     Semaphore sem = new Semaphore(2); // N разрешений
6
7     new Thread(new CountThread(sem)).start();
8     new Thread(new CountThread(sem)).start();
9     new Thread(new CountThread(sem)).start();
10 }
11
12 static class CountThread implements Runnable {
13     Semaphore sem;
14
15     CountThread(Semaphore sem) {
16         this.sem = sem;
17     }
18
19     public void run() {
20         try {
21             System.out.println(Thread.currentThread().getName()
22                 + " ожидает разрешение");
23             sem.acquire(); //получения разрешения у семафора
24             for (int i = 1; i < 5; i++) {
25
26                 System.out.println(Thread.currentThread().getName() + ": " + count);
27                 count++;
28                 Thread.sleep(100);
29             }
30             } catch (InterruptedException e) {
31                 System.out.println(e.getMessage());
32             }
33             System.out.println(Thread.currentThread().getName() + "
34 освобождает разрешение");
35             sem.release(); //Освобождение монитора
36         }
37     }
38 }

```



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация применения Semaphore

## Exchanger

Exchanger — точка синхронизации, позволяющая двум потокам обмениваться значениями. Представьте, что вы реализуете логистическое приложение — в рамках него двум автомобилям нужно пересечься в одной точке для обмена грузами — пока обмен не будет произведен, транспорт не может двинуться дальше по маршруту.

При создании экземпляра Exchanger указывается тип объекта, которым будут обмениваться потоки. Когда поток вызывает метод `exchange()`, а другой поток не готов к обмену значениями, текущий поток переходит в состояние `WAITING`. Отметим, что этот класс стоит использовать для обмена значениями между двумя потоками. Не стоит использовать Exchanger для одного потока (также для ситуации когда потоков больше чем 2) — такой поток будет в состоянии `WAITING` бесконечно долго. Когда поток X подготовил значение, и метод `exchange` выполняет поток Y, то поток Y после выполнения метода не блокируется. Также у класса есть перегруженный метод `exchange`, который принимает время, которое поток будет находится в состоянии `WAITING` — в этом случае если процесс обмена не завершился, то генерируется проверяемое исключение `TimeoutException`.



```

1 public static void main(String[] args) {
2     Exchanger<String> ex = new Exchanger<String>();
3     new Thread(new NewThread(ex, "First message", "First
Thread")).start();
4     new Thread(new NewThread(ex, "Second message", "Second
Thread")).start();
5 }
6
7 static class NewThread implements Runnable{
8     Exchanger<String> exchanger;
9     String message;
10    String name;
11
12    public NewThread(Exchanger<String> exchanger, String message,
String name) {
13        this.exchanger = exchanger;
14        this.message = message;
15        this.name = name;
16    }
17
18    public void run(){
19        try{
20            message=exchanger.exchange(message);
21            System.out.println(name + " has received: " +
message);
22        }
23        catch(InterruptedException ex){
24            System.out.println(ex.getMessage());
25        }
26    }
27 }

```



**Посмотрите демонстрацию в видеолекции:**


- Демонстрация применения Exchanger

## CowntDownLatch

CowntDownLatch — это счетчик значение которого уменьшается каждый раз, когда поток использует счетчик (поток при этом блокируется). Когда значение счетчика будет равно нулю, все заблокированные потоки будут одновременно запущены. Представьте, что вы реализуете механизм старта начала автогонок — вам нужно гарантировать, что все машины встали у старта прежде чем разрешить движение.

- ☐ Для уменьшения числа в счетчике вызывается метод `countDown()`. После вызова этого метода поток продолжает свое выполнение.
- ☐ Метод `await()` используют для блокировки потока.

```
1 public static void main(String args[]) throws InterruptedException {
2     CountdownLatch latch = new CountdownLatch(3);
3
4     new Car(3000, latch, "CAR-1").start();
5     new Car(2000, latch, "CAR-2").start();
6     new Car(1000, latch, "CAR-3").start();
7     // Ждем три потока
8     latch.await();
9
10    System.out.println(Thread.currentThread().getName() + " has
finished");
11 }
12
13 static class Car extends Thread {
14     private int delay;
15     private CountdownLatch latch;
16
17     public Car(int delay, CountdownLatch latch,
18               String name)
19     {
20         super(name);
21         this.delay = delay;
22         this.latch = latch;
23     }
24
25     @Override
26     public void run()
27     {
28         try
29         {
30             Thread.sleep(delay);
31             latch.countDown();
32             System.out.println(Thread.currentThread().getName()
33                               + " finished");
34         }
35         catch (InterruptedException e)
36         {
37             e.printStackTrace();
38         }
39     }
40 }
```

 Посмотрите демонстрацию в видеолекции:

- Демонстрация применения `CountDownLatch`

## CyclicBarrier

`CyclicBarrier` работает аналогично `CountDownLatch`, но есть несколько отличий:

- ☐ вместо методов `countDown()` и `await()` используется один метод `await()`, после вызова которого поток блокируется (если число не равно нулю);
- ☐ класс `CyclicBarrier` можно использовать повторно. Как только значение счетчика становится равным нулю, оно восстанавливается, и объект класса можно использовать заново;
- ☐ как только значение счетчика стало равным нулю, у вас есть возможность выполнить дополнительный метод имплементирующий интерфейс `Runnable`, который может быть передан в конструктор `CyclicBarrier`.

В качестве примера можно привести приложение для паромной переправы — пока паром не будет заполнен машины на 100% он не должен начинать движение.

```

1 public static void main(String args[]) throws BrokenBarrierException,
   InterruptedException {
2     CyclicBarrier cyclicBarrier = new CyclicBarrier (3);
3
4     new Car(3000, cyclicBarrier, "CAR-1").start();
5     new Car(2000, cyclicBarrier, "CAR-2").start();
6     new Car(1000, cyclicBarrier, "CAR-3").start();
7     System.out.println(Thread.currentThread().getName() + " has
   finished");
8 }
9
10 static class Car extends Thread {
11     private int delay;
12     private CyclicBarrier cyclicBarrier;
13
14     public Car(int delay, CyclicBarrier cyclicBarrier,
15               String name)
16     {
17         super(name);
18         this.delay = delay;
19         this.cyclicBarrier = cyclicBarrier;
20     }
21
22     @Override
23     public void run()
24     {
25         try
26         {
27             Thread.sleep(delay);
28             cyclicBarrier.await();
29             System.out.println(Thread.currentThread().getName()
30                               + " finished");
31         }
32         catch (InterruptedException | BrokenBarrierException e)
33         {
34             e.printStackTrace();
35         }
36     }
37 }

```



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация применения CyclicBarrier

## ReentrantLock

Теперь предлагаю рассмотреть высокоуровневые механизмы блокировки, как более удобная альтернатива `synchronized`. Давайте рассмотрим базовый интерфейс `Lock`:

- ☐ `void lock()` — захват блокировки (если доступна). Если блокировка занята другим потоком, текущий поток, который выполняет этот код, переходит в статус `BLOCKED`;
- ☐ `void lockInterruptibly()` — аналог `lock()`, но позволяет прервать заблокированный поток и восстановить выполнение через `InterruptedException`;
- ☐ `boolean tryLock()` — неблокирующий вариант метода `lock()`. Если удалось получить блокировку, то метод возвращает `true`;
- ☐ `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.

Сегодня мы рассмотрим одну из реализаций интерфейса `Lock` — `ReentrantLock`. Он позволяет одному и тому же потоку вызывать метод `lock`, даже если он его вызывал ранее, без освобождения блокировки.

У класса `ReentrantLock`, кроме методов интерфейса `Lock`, есть фабричный метод `newCondition()`. Этот метод возвращает объект `Condition`, который позволяет добавить текущий поток в `wait set` данного объекта `Condition`. Это дает возможность организовывать разные условия ожидания по одной и той же блокировке, чего не позволяют ключевое слово `synchronized` и связки методов `wait()/notify()`. Для того чтоб объект попал в `wait set` для данного `Condition` объекта, нужно вызвать метод `await()`. Чтобы разбудить поток или потоки, которые есть в `wait set`, необходимо вызвать методы `signal()` или `signalAll()`. Эти методы аналогичны методам `wait()`, `notify()` и `notifyAll()` у объекта `Object` (методы `wait()`, `notify()` и `notifyAll()` в объекте `Object` — `final`, методам для объекта `Condition` придумали другие наименования).

```

1 static Integer resource = Integer.valueOf(0);
2
3 public static void main(String[] args) throws InterruptedException
4 {
5     ReentrantLock locker = new ReentrantLock(); // создаем Lock
6     for (int i = 0; i < 5; i++){
7         Thread thread = new Thread(new LockThread(locker));
8         thread.start();
9         thread.join();
10    }
11
12    static class LockThread implements Runnable{
13        ReentrantLock locker;
14        LockThread(ReentrantLock locker){
15            this.locker = locker;
16        }
17        public void run(){
18            locker.lock(); // устанавливаем блокировку
19            try{
20                for (int i = 0; i < 5; i++){
21                    System.out.printf("%s %d \n",
22                        Thread.currentThread().getName(), resource.intValue());
23                    resource++;
24                    Thread.sleep(100);
25                }
26            } catch (InterruptedException e){
27                System.out.println(e.getMessage());
28            } finally{
29                locker.unlock(); // снимаем блокировку
30            }
31        }
32    }

```



Посмотрите демонстрацию в видеолекции:

- Демонстрация применения ReentrantLock

## Коллекции в многопоточной среде

Обычные коллекции в Java не синхронизированы и не могут быть безопасно использованы в многопоточной среде. За исключением случаев, когда обращение к этим коллекциям происходит из синхронизированных блоков кода.



## Синхронизированные коллекции

Первый вариант и самый простой вариант – это превратить обычную коллекцию в синхронизированную, для этого необходимо вызвать соответствующий ее типу метод `Collections.synchronized*()`. В этом случае будет применен паттерн декоратор к обычной коллекции, который оборачивает каждый метод в `synchronized`-блок, в результате при каждом чтении или изменении такой коллекции блокируются все остальные операции. Не самый оптимальный вариант для использования.

## Интерфейс List

`CopyOnWriteArrayList` следует применять, когда в вашем приложении есть множество потоков которые читают элементы из коллекции, и только несколько (относительно небольшое) потоков, которые редко пишут данные в коллекцию — т.е. изменение элементов коллекции происходит не часто (например некий справочник со статусами объектов). В качестве примера использования можно привести задачу с обработкой данных, полученных от некоторого источника (например сигнал GPS датчиков) — каждый поток должен обработать все данные — множество принимающих станций только читают сигнал.

При попытке записи структура `Copy on write` фактически создает новую копию данных. Это позволяет нескольким потокам одновременно читать данные, и одному потоку записывать элементы в коллекцию в каждый конкретный момент времени.

- ☐ `Copy on write` содержит `volatile` массив элементов и при каждом изменении коллекции: добавлении, удалении или замене элементов, создается новая локальная копия такого массива для изменений. После модификации измененная копия становится текущей.
- ☐ `volatile` используется для того, чтобы все потоки сразу увидели изменения в массиве (значением переменной не копируется в кэш CPU). Такой алгоритм гарантирует, что потоки которые будут читать не изменяющиеся во времени данные, а при параллельном внесении изменений не будет выброшено исключение — `ConcurrentModificationException`.
- ☐ Не вызывайте `remove()` при использовании `Iterator` для чтения коллекции — будет сгенерировано `UnsupportedOperationException`, потому что текущая копия коллекции не подлежит изменению.
- ☐ Для операций записи внутри класса `CopyOnWriteArrayList` создается блокировка, чтобы в конкретный момент времени только один поток мог изменять `copy on write` структуру данных.

## Интерфейс Set

`CopyOnWriteArraySet` (с точки зрения реализация это `Copy on write` коллекция рассмотренная выше) — гарантирует уникальность хранимых значений (мы используем интерфейс `Set`). `CopyOnWriteArraySet` реализован с использованием `CopyOnWriteArrayList`, таким образом их поведение аналогичны - по сути добавляет проверку на уникальность для элементов коллекции

`CopyOnWriteArraySet` это не упорядоченное множество, и если вам необходимо обеспечить сортировку элементов, то воспользуйтесь коллекцией `ConcurrentSkipListSet`, которая так же implements интерфейсы `SortedSet` и `NavigableSet`.

Реализация `ConcurrentSkipListSet` основана на соответствующей коллекции `ConcurrentSkipListMap`. Каждый элемент структуры (кроме самого значения), содержит ссылку на соседние элементы - аналог `LinkedHashMap`. В такой коллекции есть ссылки вышних порядков, которые указывают на элементы, находящиеся впереди текущего, на произвольное число в определенном диапазоне, заданном для этого уровня ссылок. Для следующего уровня ссылок это число больше, чем для предыдущего.

Скорость поиска элемента обеспечивается за счет использования ссылок вышних порядков - эффективность поиска сравнима с поиском элементов по бинарному дереву. Для изменения, вставки не требуется полной блокировки `ConcurrentSkipListSet`, достаточно найти элемент, который будет изменен, и заблокировать только два соседних элемента (для изменения ссылок), указывающих на элемент, подлежащий изменению.

## Интерфейс Map

Есть несколько реализаций интерфейса `Map` для использования в многопоточных коллекциях: `HashMap`, `ConcurrentHashMap` и `ConcurrentSkipListMap`.

Такие коллекции, реализующие интерфейс `Map` позволяют нам решать прикладные задачи, связанные с отображением список используемых сразу в нескольких потоков. Представьте, что мы реализуем многопользовательское приложение - клиенту проходят процедуру идентификации, аутентификации и авторизуются - наша задача обеспечить хранение `Map` отображения идентификатора клиента на его сессию (любая сессия может обрабатываться любым потоком).

В первую очередь стоит упомянуть конечно же `HashMap` - это классическая реализация интерфейса `Map`, все методы которой синхронизированы - по этой причине она является `deprecated` и не рекомендована к применению.



Сегодня мы рассмотрим наиболее популярную реализацию - `ConcurrentHashMap`. `ConcurrentHashMap` это массив сегментов, каждый из которых представляет собой `HashMap` с массивом корзин. Мы помним, что каждая корзина или бакет `HashMap` это связанный список или бинарное дерево.

И так давайте посмотрим, что происходит при работе с `ConcurrentHashMap`:

- ☐ При добавлении значения (пары ключ и значение) блокируется только один из сегментов, а после добавления пары все происходит, как в обычной `Map`.
- ☐ При чтении, сегменты не блокируются - несколько потоков могут читать данные одновременно.
- ☐ Параллельное изменение возможно, только при обращении к двум разным сегментам.
- ☐ При внесении изменений в один из сегментов, другой поток может читать из него параллельно. При одновременном чтении и модификации данных в результате чтения вернется последнее измененное значение.
- ☐ Рехеширование в `ConcurrentHashMap` происходит по отдельности в каждом сегменте, поэтому оно может выполняться одновременно с записью в другой сегмент.

## Интерфейс `Queue`

`BlockingQueue` — мы используем интерфейс потокобезопасных очередей, если несколько потоков должны читать и писать из общей `Queue`. `BlockingQueue` способна блокировать поток, который добавляет или читает элементы из очереди — если очередь пуста, а поток пытается получить элемент то он блокируется, до того момента пока в очереди не будет добавлен новый. Также и в обратную сторону, если поток пытается добавить элемент в заполненную очередь, он также блокируется.

В качестве примера можно привести реализацию обычной электронной очереди в любом государственном учреждении — каждой операционист по сути это поток, который приступает к обслуживанию очередного клиента из очереди при освобождении.

`BlockingQueue` используются, когда одни потоки только добавляют элементы в очередь, а другие только читают — классический паттерн проектирования, применяемый при разработке микро сервисной архитектуры при использовании брокеров сообщений. Если поток добавляет объекты в очередь то его называют `producer` (Производитель) — добавление нового элемента в очередь возможно до

тех пор, пока она не заполнится. Если поток который читает значение то он consumer (Потребитель) — чтение из очереди возможно только если в ней есть элементы.

BlockingQueue реализует принцип FIFO. Элементы отсортированы по времени нахождения в очереди, а голова находится там дольше всех остальных.

BlockingQueue не поддерживает значение null — при попытке добавление, будет сгенерирован NullPointerException.



**Посмотрите демонстрацию в видеолекции:**

- Демонстрация применения потокобезопасных коллекций

## Подведем итоги

И так коллеги, сегодня мы узнали как создавать потокобезопасные приложения. Мы рассмотрели основные принципы применяемые при использовании многопоточности, причины возникновения race condition, а также способы их устранения. Также мы познакомились с Java Memory Model для более глубокого понимания устройства JVM.

На следующем уроке мы познакомимся с утилитарными механизмами автоматизации сборки проектов Maven и Gradle, которые существенно упрощают процесс управления проектом и работы с библиотеками.

## Подготовка к семинару

### Что можно почитать еще?

1. <https://habr.com/ru/sandbox/167189/>
2. <https://habr.com/ru/articles/510454/>

# Используемая литература

1. Java Concurrency in Practice. Brian Goetz
2. Философия Java. Брюс Эккель
3. Java. Полное руководство. Шилдт Герберт