Java Pro 6.3

TEL-RAN
by Starta Institute

Чистый код в многопоточности. Concurrent

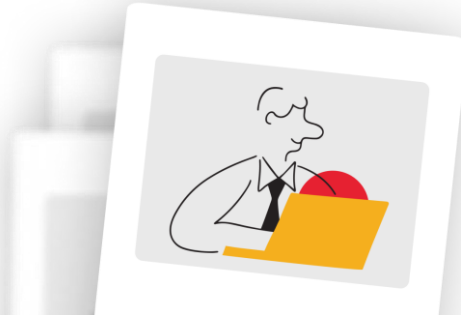
ПРЕПОДАВАТЕЛЬ

TEL-RAN
by Starta Institute

Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

TEL-RAN
by Starta Institute

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

TEL-RAN
by Starta Institute

1 ПОВТОРЕНИЕ ИЗУЧЕННОГО

Повторение

TEL-RAN
by Starta Institute

Какие модели многопоточности Вам известны?

В чём их сходства и отличия?

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде.

```
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Task1());
        t1.start();
        t1.notify();
    }

    public static class Task1 implements Runnable {
        private static final Object LOCK = new Object();

        @Override
        public void run() {
            synchronized (LOCK) {
                System.out.println("Task is running");
            }
        }
    }
}
```

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде.

В данном коде вызывается notify() на объекте, который не находится в состоянии монитора (lock), что приводит к IllegalMonitorStateException.

Решение: Нельзя вызывать notify() или notifyAll() без предварительного захвата монитора с помощью synchronized. Используйте lock.notify() внутри синхронизированного блока.

```
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Task1());
        t1.start();
        t1.notify();
    }

    public static class Task1 implements Runnable {
        private static final Object LOCK = new Object();

        @Override
        public void run() {
            synchronized (LOCK) {
                System.out.println("Task is running");
            }
        }
    }
}
```

Повторение

Исправьте ошибку в коде.

```
public class Main1 {
    public static void main(String[] args) {
        Task2 task = new Task2();
        Thread t1 = new Thread(task);
        t1.start();
        t1.interrupt();
    }
    public static class Task2 implements Runnable {
        private final Object monitor = new Object();

        @Override
        public void run() {
            synchronized (monitor) {
                System.out.println("Task is running");
                monitor.wait();
            }
        }
    }
}
```

Повторение

Исправьте ошибку в коде.

InterruptedException необходимо обработать.

```
public class Main1 {
    public static void main(String[] args) {
        Task2 task = new Task2();
        Thread t1 = new Thread(task);
        t1.start();
        t1.interrupt();
    }
    public static class Task2 implements Runnable {
        private final Object monitor = new Object();

        @Override
        public void run() {
            try {
                synchronized (monitor) {
                    System.out.println("Task is running");
                    monitor.wait();
                }
            } catch (InterruptedException e) {
                System.out.println("Task interrupted");
            }
        }
    }
}
```

Повторение

Исправьте ошибку в коде.

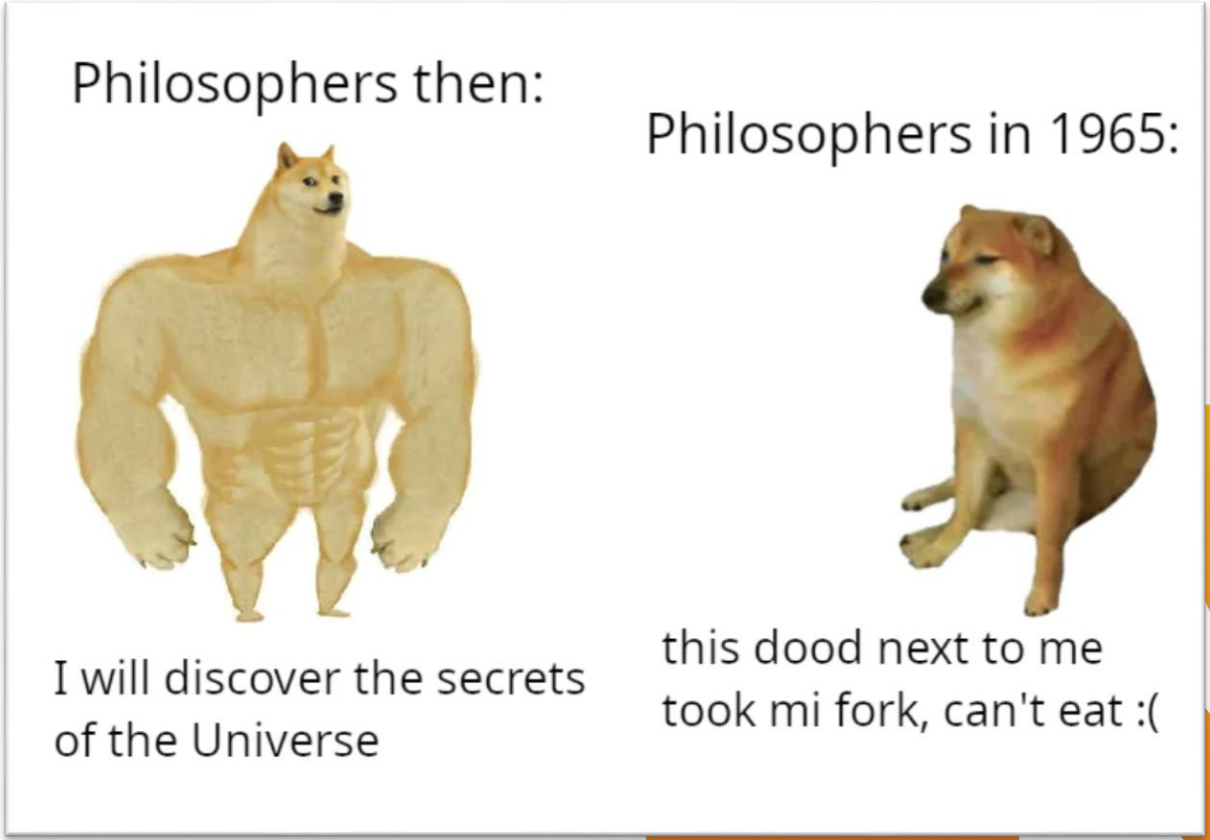
InterruptedException не обрабатывается корректно.

Решение: В коде нужно добавить обработку InterruptedException или использовать Thread.currentThread().interrupt() для сохранения статуса прерывания.

```
public class Main1 {
    public static void main(String[] args) {
        Task2 task = new Task2();
        Thread t1 = new Thread(task);
        t1.start();
        t1.interrupt();
    }
    public static class Task2 implements Runnable {
        private final Object monitor = new Object();

        @Override
        public void run() {
            try {
                synchronized (monitor) {
                    System.out.println("Task is running");
                    monitor.wait();
                }
            } catch (InterruptedException e) {
                System.out.println("Task interrupted");
            }
        }
    }
}
```

Повторение

В чём прикол мема?



TEL-RAN
by Starta Institute

2 ОСНОВНОЙ БЛОК

Введение

TEL-RAN
by Starta Institute

- Чистопоточно
- Конкуренция
- Делим коллекции

Проблема

TEL-RAN
by Starta Institute

Кроме изученных ранее моделей в многопоточности есть и другие сложившиеся практики, которые позволяют писать код правильнее.

Какие практики ещё могут быть полезны?

Проблема

TEL-RAN
by Starta Institute

Многие из них изложены в книге «Чистый код» Роберта Мартина в главе «Многопоточность»

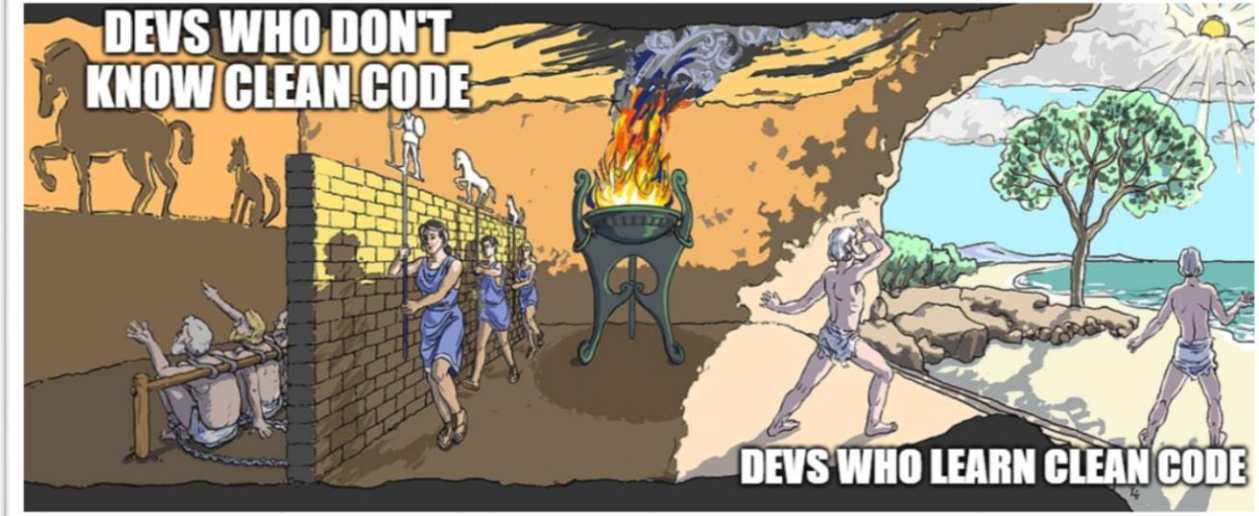
Чистопоточно

Мифы о многопоточности

TEL-RAN
by Starta Institute

- Многопоточность всегда повышает быстродействие.

Действительно, многопоточность иногда повышает быстродействие, но только при относительно большом времени ожидания, которое могло бы эффективно использоваться другими потоками или процессорами.



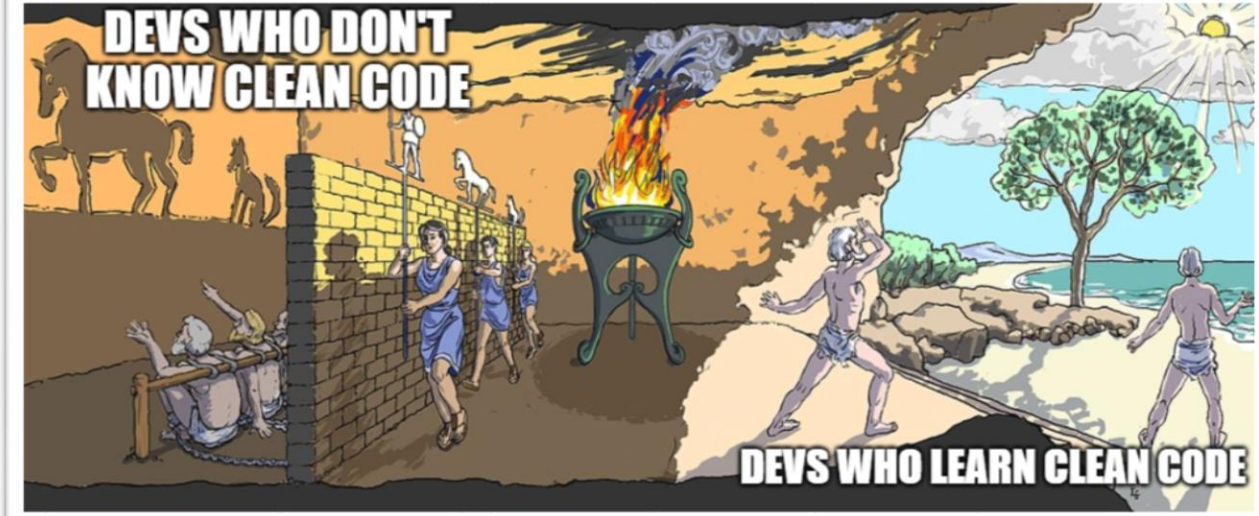
Чистопоточно

Мифы о многопоточности

TEL-RAN
by Starta Institute

- Написание многопоточного кода не изменяет архитектуру программы.

На самом деле архитектура многопоточного алгоритма может заметно отличаться от архитектуры однопоточной системы. Отделение «что» от «когда» обычно оказывает огромное влияние на структуру системы.



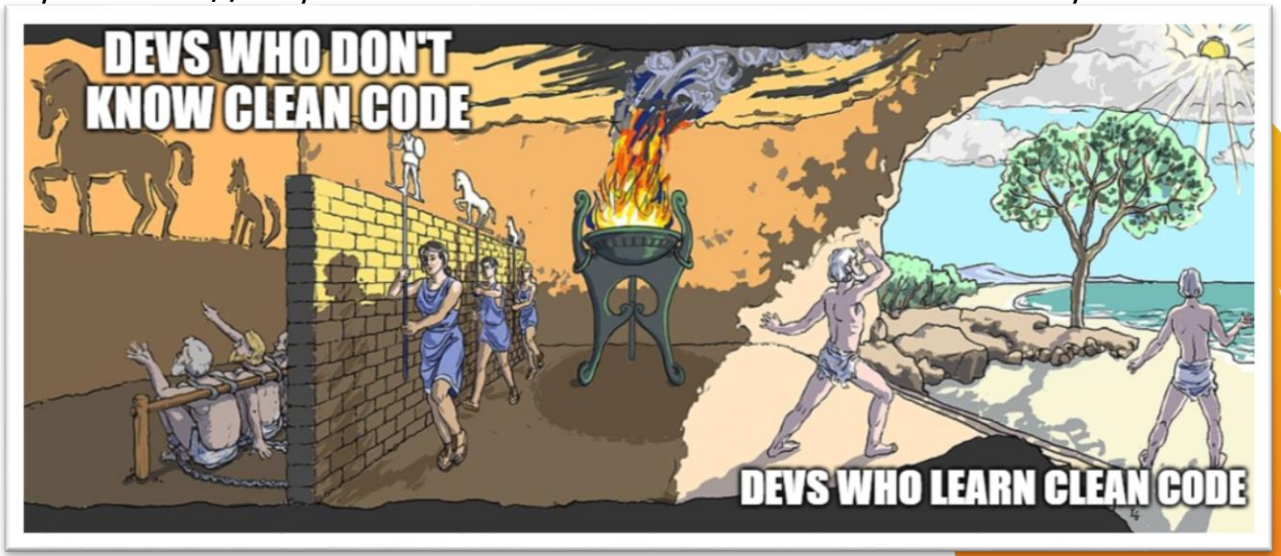
Чистопоточно

Мифы о многопоточности

TEL-RAN
by Starta Institute

- При написании web-приложений разбираться в проблемах многопоточного программирования не обязательно.

Отчасти это правда, но всегда желательно знать, как работает технология, чтобы защититься от проблем одновременного обновления и взаимных блокировок.

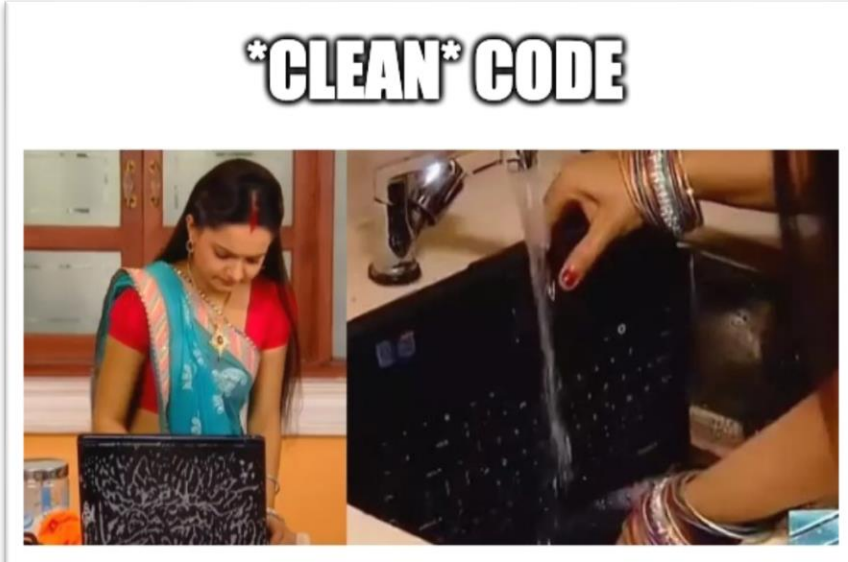


Чистопоточно

Защита от ошибок многопоточности

Принцип единой ответственности (SRP)] гласит, что метод/класс/компонент должен иметь только одну причину для изменения. Многопоточные архитектуры достаточно сложны, чтобы их можно было рассматривать как причину изменения сами по себе, а следовательно, они должны отделяться от основного кода.

Рекомендация: отделяйте код, относящийся к реализации многопоточности, от остального кода.



Чистопоточно

Защита от ошибок многопоточности

Ограничивайте область видимости данных.

Два программных потока, изменяющих одно поле общего объекта, могут мешать друг другу, что приводит к непредвиденному поведению.

Одно из возможных решений – защита критической секции кода, в которой происходят обращения к общему объекту, ключевым словом *synchronized*. Количество критических секций в коде должно быть сведено **к минимуму**.

Серьезно относитесь к инкапсуляции данных; жестко ограничьте доступ ко всем общим данным.

Чистопоточно

Защита от ошибок многопоточности

Используйте копии данных.

Чтобы потоки не боролись за один ресурс, можно дать каждому потоку копию для работы:

- в одних ситуациях можно скопировать общий объект и ограничить доступ к копии (доступ только для чтения);
- в других ситуациях объекты копируются, результаты работы нескольких программных потоков накапливаются в копиях, а затем объединяются в одном потоке.

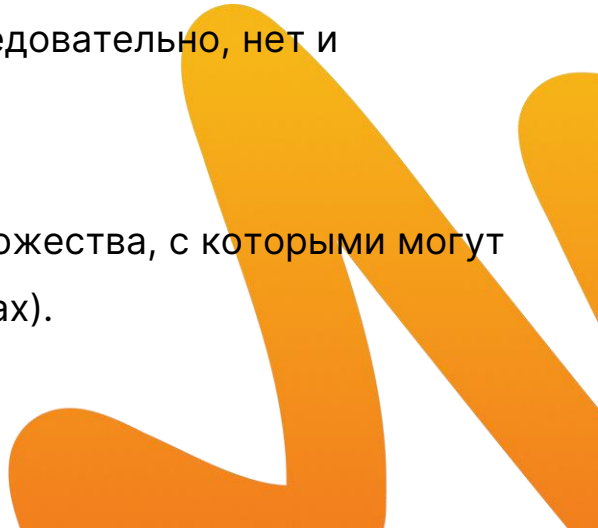
Чистопоточно

Защита от ошибок многопоточности

Потоки должны быть как можно более независимы.

Постарайтесь писать многопоточный код так, чтобы каждый поток существовал в собственном замкнутом пространстве и не использовал данные совместно с другими процессами. Каждый поток обрабатывает один клиентский запрос, все его данные берутся из отдельного источника и хранятся в локальных переменных. В этом случае каждый поток работает так, словно других потоков не существует, а следовательно, нет и требований к синхронизации.

Для этого постарайтесь разбить данные не независимые подмножества, с которыми могут работать независимые потоки (возможно, на разных процессорах).

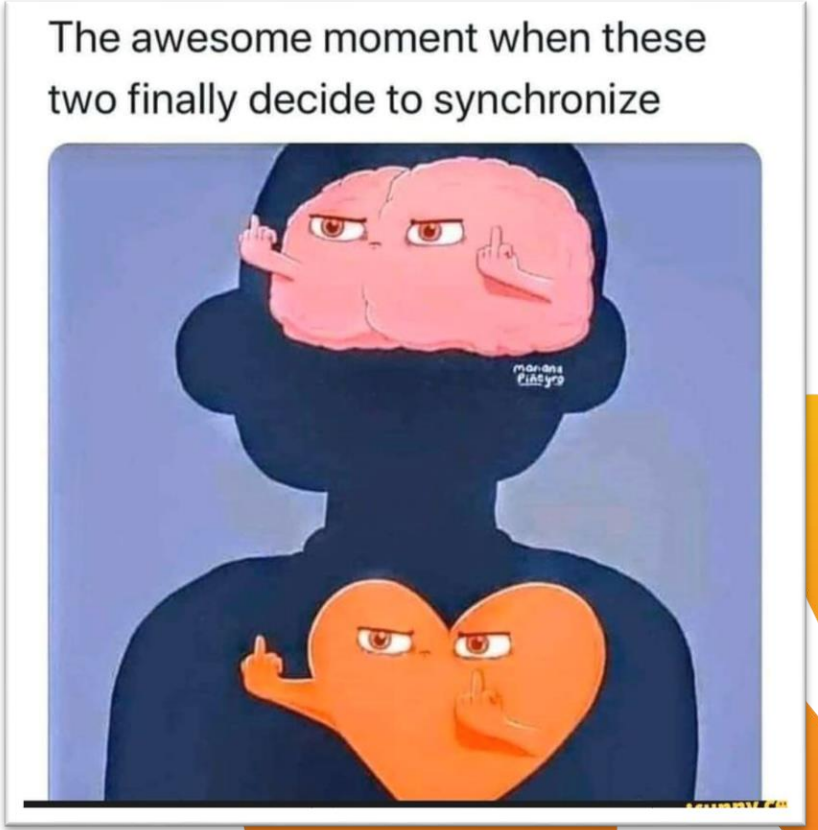


Чистопоточно

Защита от ошибок многопоточности

Остерегайтесь зависимостей между синхронизированными методами.

Зависимости между синхронизированными методами приводят к появлению коварных ошибок в многопоточном коде. Если класс содержит более одного синхронизированного метода, стоит рассмотреть вариант разбиения такого класса. Лучше избегать использования нескольких методов одного совместно используемого объекта.



Чистопоточно

Защита от ошибок многопоточности

Синхронизированные секции должны иметь минимальный размер.

Ключевое слово *synchronized* устанавливает блокировку. Все секции кода, защищенные одной блокировкой, в любой момент времени гарантированно выполняются только в одном программном потоке. Блокировки обходятся дорого, так как они создают задержки и увеличивают затраты ресурсов.

Синхронизация за пределами минимальных критических секций увеличивает конкуренцию между потоками и снижает производительность.

Чистопоточно

Защита от ошибок многопоточности

Предусмотрите корректное завершение потоков на ранней стадии разработки.

Написание системы, которая должна работать бесконечно, заметно отличается от написания системы, которая работает в течение некоторого времени, а затем корректно завершается. Реализовать корректное завершение порой бывает весьма непросто.

Одна из типичных проблем – взаимная блокировка программных потоков, бесконечно долго ожидающих сигнала на продолжение работы.

Представьте систему с родительским потоком, который порождает несколько дочерних потоков, а затем дожидается их завершения, чтобы освободить свои ресурсы и завершиться. Если один из дочерних потоков попадет во взаимную блокировку?

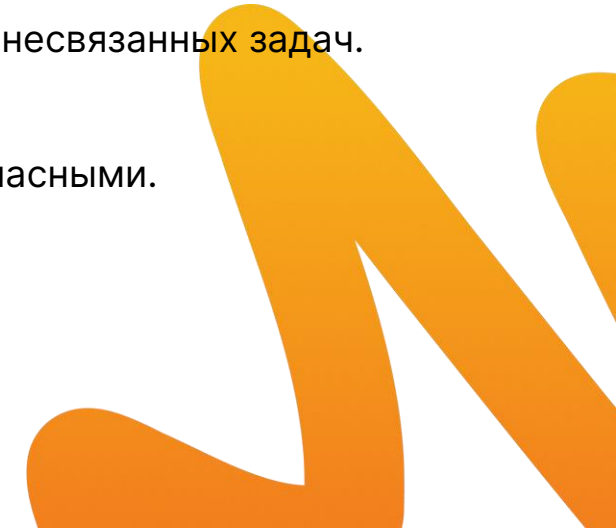
Чистопоточно

Защита от ошибок многопоточности

Знайте свою библиотеку.

В Java 5 возможности многопоточной разработки были значительно расширены по сравнению с предыдущими версиями благодаря пакету java.util.concurrent. При написании многопоточного кода в Java 5+ следует руководствоваться следующими правилами:

- Используйте потоко-безопасные коллекции.
- Используйте механизм Executor Framework для выполнения несвязанных задач.
- По возможности используйте неблокирующие решения.
- Некоторые библиотечные классы не являются потоко-безопасными.



Проблема

Мы знаем, что для удобной работы с массивами/коллекциями/потоками ввода-вывода и т.д. в стандартной библиотеке Java есть утилитарные классы (Arrays, Collections, Files), удобные конструкции (try-with-resources, лямда-выражения и т.д.).

Для многопоточности тоже должны быть свои классы для автоматизации решения типовых задач.

Конкуренция

Пакет java.util.concurrent

TEL-RAN
by Starta Institute

Классы и интерфейсы java.util.concurrent берут на себя решение типовых задач, связанных с многопоточностью. «Под капотом» этих классов используются базовые потоки, синхронизации и распараллеливание доступа к ресурсам. Но этому пакету анписание многопоточных приложений становится значительно проще.

Конкуренция

Структура java.util.concurrent

TEL-RAN
by Starta Institute

Наименование	Примечание
collections	Набор более эффективно работающих в многопоточной среде коллекций нежели стандартные универсальные коллекции из *java.util* пакета
synchronizers	Объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков.
atomic	Набор атомарных классов, позволяющих использовать принцип действия механизма *оптимистической блокировки* для выполнения атомарных операций.
Queues	Объекты создания блокирующих и неблокирующих очередей с поддержкой многопоточности.
Locks	Механизмы синхронизации потоков, альтернативы базовым *synchronized, wait, notify, notifyAll*
Executors	Механизмы создания пулов потоков и планирования работы асинхронных задач

Проблема

Обычные наборы данных, реализующих интерфейсы *List*, *Set* и *Map*, нельзя использовать в многопоточных приложениях, если требуется синхронизация, т.е. такие коллекции недопустимы для одновременного чтения и изменения данных разными потоками.

Методы *обрамления Collections framework* (*synchronizedList*, *synchronizedSet*, *synchronizedMap*), появившиеся в JDK 1.2, имеют существенный недостаток, связанный с препятствованием масштабируемости, поскольку с коллекцией одновременно может работать только один поток.

Как сделать коллекцию действительно многопоточной?

Делим коллекции

Concurrent Collections

TEL-RAN
by Starta Institute

Пакет *java.util.concurrent* предлагает свой набор потокобезопасных классов, допускающих разными потоками одновременное чтение и внесение изменений. Итераторы классов данного пакета представляют данные на определенный момент времени и не вызывают исключение *ConcurrentModificationException*. Все операции по изменению коллекции (*add*, *set*, *remove*) приводят к созданию новой копии внутреннего массива. Этим гарантируется, что при проходе итератором по коллекции не будет *ConcurrentModificationException*. Следует помнить, что при копировании массива копируются только ссылки на объекты.

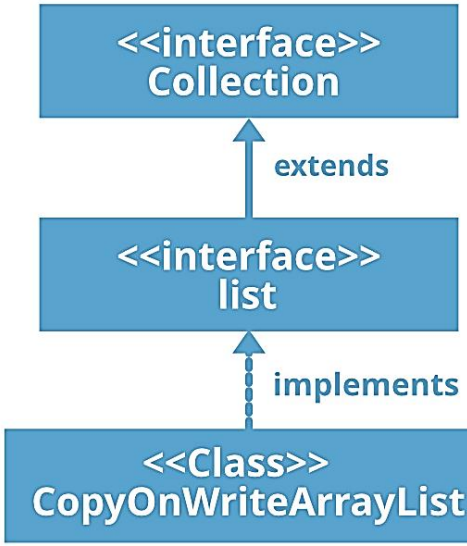
Делим коллекции
CopyOnWriteArrayList

TEL-RAN
by Starta Institute

CopyOnWriteArrayList реализует алгоритм *CopyOnWrite* и является потокобезопасным аналогом *ArrayList*.

Класс *CopyOnWriteArrayList* содержит изменяемую ссылку на неизменяемый массив, обеспечивая преимущества потокобезопасности без необходимости использования блокировок. Т.е. при выполнении модифицирующей операции *CopyOnWriteArrayList* создаёт новую копию списка и гарантирует, что её итераторы вернут состояние списка на момент создания итератора и не вызовут *ConcurrentModificationException*.

CopyOnWriteArrayList следует использовать вместо *ArrayList* в потоконагруженных приложениях, где могут иметь место нечастые операции вставки и удаления в одних потоках и одновременный перебор в других.



Делим коллекции

Конструкторы CopyOnWriteArrayList

```
// создание пустой потокобезопасной коллекции
CopyOnWriteArrayList()

// создание потокобезопасной коллекции с данными list
CopyOnWriteArrayList(Collection<? extends E> list);

// создание потокобезопасной коллекции с копированием данных
CopyOnWriteArrayList(E[] toCopyIn)
```

Делим коллекции

Пример CopyOnWriteArrayList

CopyOnWriteArrayListExample.zip

Задание

TEL-RAN
by Starta Institute

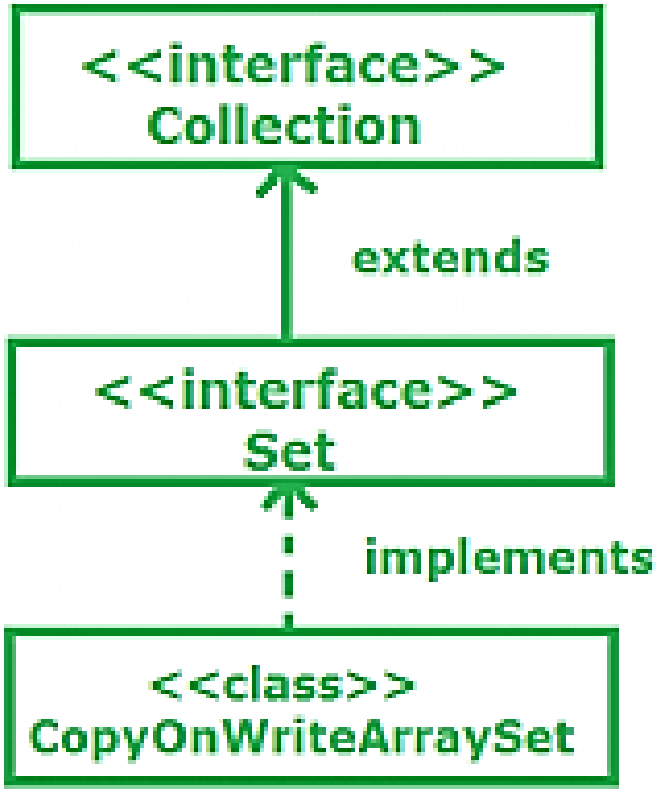
Создайте 5 потоков, каждый из которых генерирует случайные числа. Создайте класс ParallelRandom и в нём метод генерации списка из заданного количества элементов. Заполните список из 10_000 элементов с помощью ParallelRandom.

Делим коллекции
CopyOnWriteArraySet

TEL-RAN
by Starta Institute

CopyOnWriteArraySet выполнен на основе *CopyOnWriteArrayList* с реализацией интерфейса Set, т.е. использует все его возможности.

Лучше всего CopyOnWriteArraySet использовать для read-only коллекций небольших размеров. Если в данных коллекции произойдут изменения, накладные расходы, связанные с копированием, не должны быть ресурсоёмкими.

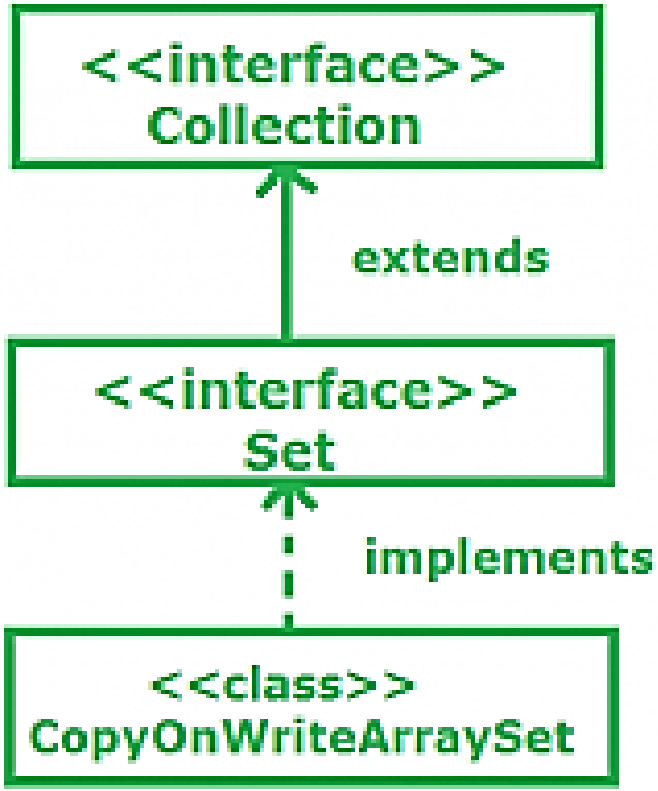


Делим коллекции
CopyOnWriteArraySet

TEL-RAN
by Starta Institute

Итераторы класса *CopyOnWriteArraySet* не поддерживают операцию *remove()*. Попытка удалить элемент во время итерирации приведет к вызову исключения *UnsupportedOperationException*. В своей работе итераторы используют «моментальный снимок» массива, который был сделан на момент создания итератора.

Таким образом, если набор данных небольшой и не подвержен изменениям, то лучше использовать *CopyOnWriteArraySet*.



Делим коллекции

Конструкторы CopyOnWriteArraySet

```
// Создание пустого набора данных
CopyOnWriteArraySet()

// Создание набора с элементами коллекции coll
CopyOnWriteArraySet(Collection<? extends E> coll)
```

Делим коллекции

Пример CopyOnWriteArraySet

ArraySetExample.zip

Задание

TEL-RAN
by Starta Institute

Дополните ParallelRandom методом генерации множества случайных чисел.

Ради эксперимента попробуйте у сформированного CopyOnWriteArraySet вызвать операцию удаления элемента.

Делим коллекции
ConcurrentMap

TEL-RAN
by Starta Institute

Интерфейс **java.util.concurrent.ConcurrentMap** определяет методы потокобезопасной мапы.

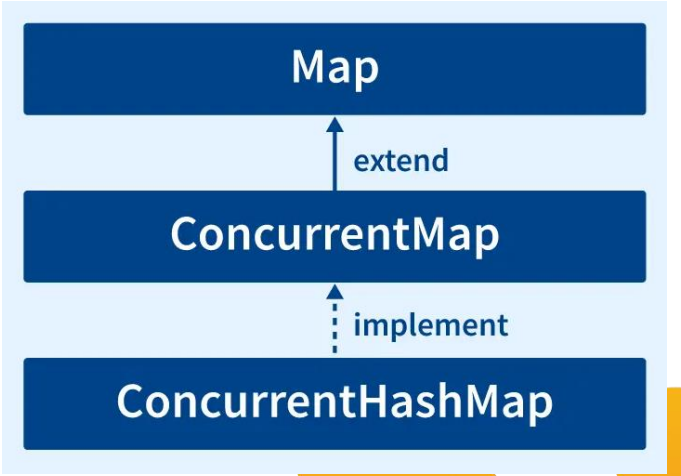
```
public interface ConcurrentMap<K,V> extends Map<K,V>
{
    // добавить, если нет объекта value по ключу K
    V putIfAbsent(K key, V value);

    // удалить, если имеется объект value с ключом K
    boolean remove(K key, V value);

    // заменить oldValue новым newValue объекта с ключом K
    boolean replace(K key, V oldValue, V newValue);

    // заменить новым значением newValue объект с ключом K
    V replace(K key, V newValue);
}
```

Делим коллекции
ConcurrentHashMap

TEL-RAN
by Starta Institute

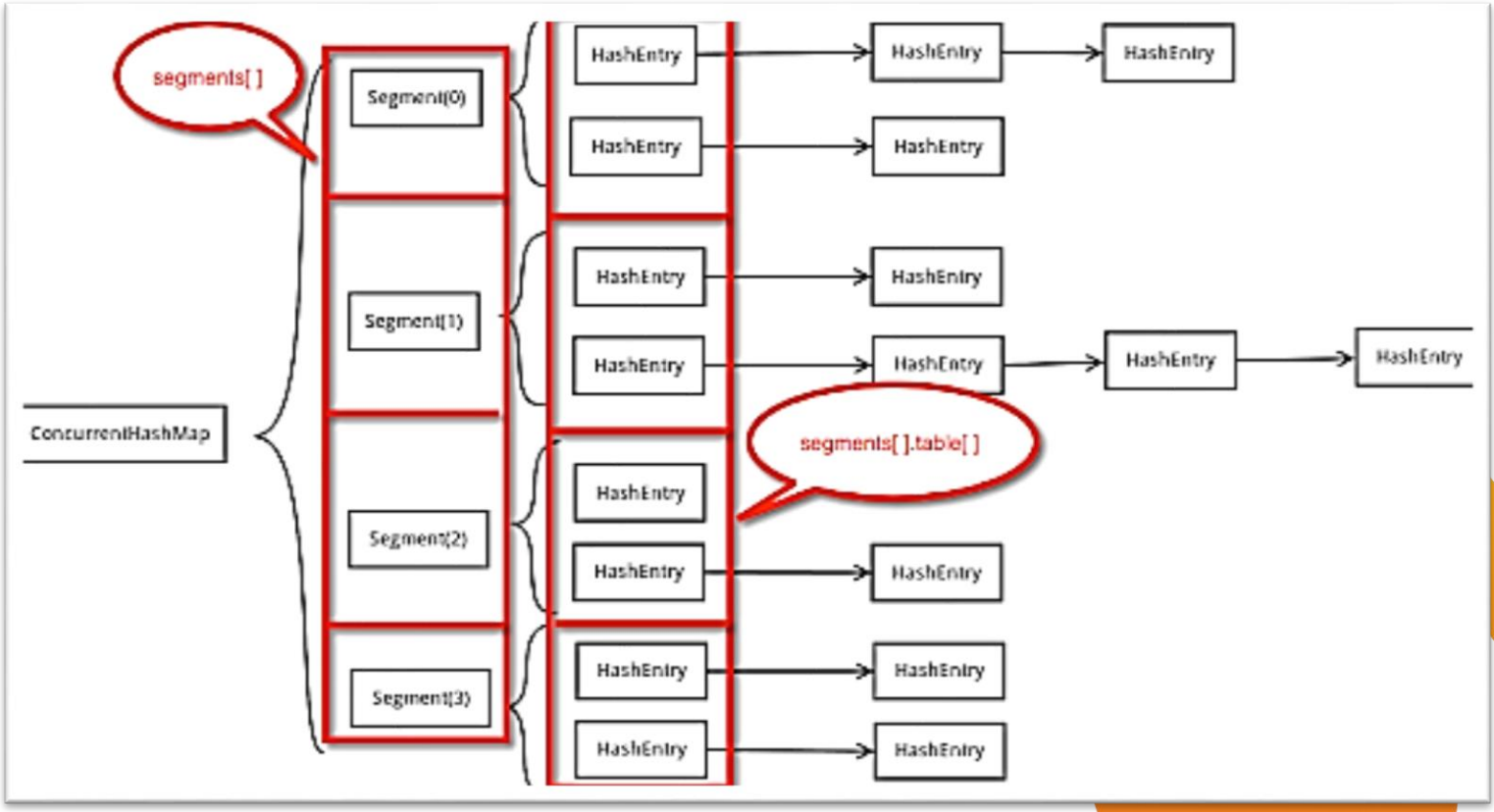
ConcurrentHashMap<K, V> реализует интерфейс *java.util.concurrent.ConcurrentMap* и отличается от *HashMap* и *Hashtable* внутренней структурой хранения пар key-value.

ConcurrentHashMap использует несколько сегментов, и данный класс можно рассматривать как группу *HashMap'ов*. По умолчанию количество сегментов равно 16. Доступ к данным определяется по сегментам, а не по объекту. Итераторы данного класса фиксируют структуру данных на момент начала его использования.



Делим коллекции
ConcurrentHashMap

TEL-RAN
by Starta Institute



Делим коллекции

Конструкторы ConcurrentHashMap

```
/* Создание пустой коллекции с параметрами по умолчанию:  * capacity (16), load factor (0.75),
concurrencyLevel (16) */
ConcurrentHashMap();
/* Создание пустой коллекции с заданным initialCapacity, остальные параметры по умолчанию */
ConcurrentHashMap(int initialCapacity)
/* Создание пустой коллекции с заданными параметрами initialCapacity, loadFactor; параметр
concurrencyLevel определен по умолчанию */
ConcurrentHashMap(int initialCapacity, float loadFactor);
/* Создание пустой коллекции с заданными параметрами initialCapacity, loadFactor,
oncurrencyLevel */
ConcurrentHashMap (int initialCapacity,
                   float loadFactor,
                   int concurrencyLevel);
/*Создание потокобезопасной коллекции с заданными значения map initialCapacity, loadFactor,
concurrencyLevel */
ConcurrentHashMap(Map<? extends K,? extends V> map)
```

Делим коллекции

Пример ConcurrentHashMap

HashMapExample.zip

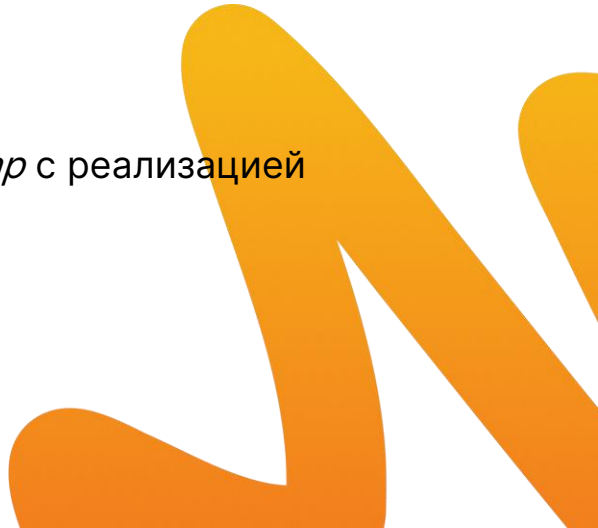
Делим коллекции

Другие реализации ConcurrentMap

ConcurrentNavigableMap расширяет возможности интерфейса *NavigableMap* для использования в многопоточных приложениях; итераторы класса декларируются как потокобезопасные и не вызывают *ConcurrentModificationException*.

ConcurrentSkipListMap является аналогом коллекции *TreeMap* с сортировкой данных по ключу и с поддержкой многопоточности.

ConcurrentSkipListSet выполнен на основе *ConcurrentSkipListMap* с реализацией интерфейса Set.



Задание

TEL-RAN
by Starta Institute

Реализуйте программу, использующую *ConcurrentMap*, чтобы хранить и обновлять информацию о балансе нескольких пользовательских счетов в многопоточной среде. Разработайте метод для атомарного увеличения баланса на счету.

TEL-RAN
by Starta Institute

3 Домашнее задание

Домашнее задание

TEL-RAN
by Starta Institute

1 Прочитайте главу «Многопоточность» в книге «Чистый код» Роберта Мартина. Обратите внимание на раздел «Тестирование многопоточного кода»

2 Дан список чисел из большого количества случайных чисел. Используя синхронизированные коллекции, уберите из списка отрицательные значения. Замерьте время выполнения метода в однопоточном режиме и в многопоточном.

3 Разработайте программу, использующую *ConcurrentMap*, для обеспечения безопасного обновления значения по условию. Например, уменьшайте значение ключа "stock" на 1 только если текущее значение больше 0.

ЗАКЛЮЧЕНИЕ

TEL-RAN
by Starta Institute