СБЕРБАНК ТЕХНОЛОГИИ

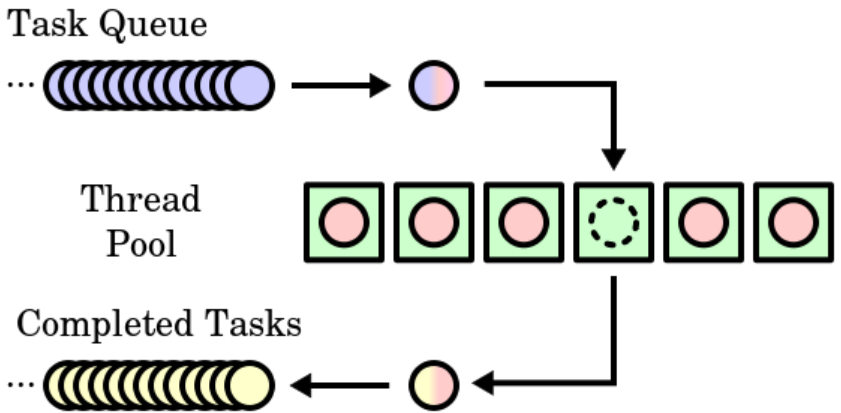
java.util.concurrent package

- Thread pools
- Futures
- Locks
- Atomics
- Concurrent data structures

THREAD POOLS:ЧТО ТАКОЕ ПУЛ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

Thread pool - это шаблон проектирования, позволяющий эффективно организовать параллельное выполнение задач.



THREAD POOLS: ПРИЧИНЫ ПРИМЕНЕНИЯ

СБЕРБАНК ТЕХНОЛОГИИ

1. Уменьшение нагрузки на JVM, возникающей вследствие создания новых потоков.

THREAD POOLS: ПРИЧИНЫ ПРИМЕНЕНИЯ

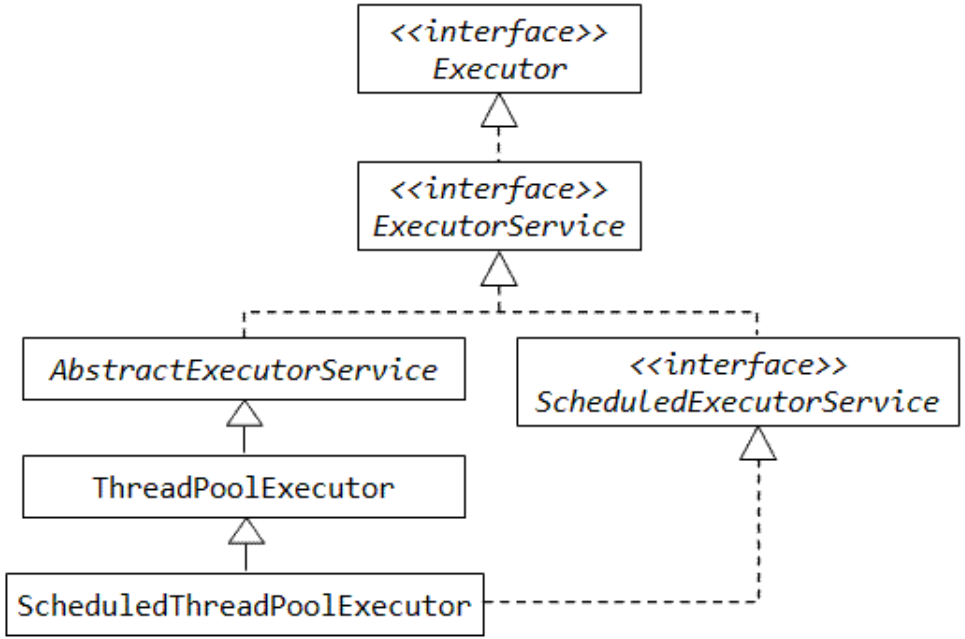
СБЕРБАНК ТЕХНОЛОГИИ

1. Уменьшение нагрузки на JVM, возникающей вследствие создания новых потоков.
2. Контроль большого количества потоков, в которых происходит параллельное выполнение задач.

THREAD POOLS: ИЕРАРХИЯ КЛАССОВ И ИНТЕРФЕЙСОВ

СБЕРБАНК ТЕХНОЛОГИИ

Все необходимые классы и интерфейсы позволяющие организовать работу с Thread pool находятся в пакете java.util.concurrent



THREAD POOLS: ИНТЕРФЕЙС EXECUTOR

СБЕРБАНК ТЕХНОЛОГИИ

public interface Executor

Позволяет построить классы, которые знают как именно необходимо выполнить Runnable задачу, разделить представление задачи от ее выполнения.

- void execute(Runnable command)

THREAD POOLS: ИНТЕРФЕЙС CALLABLE<V>

СБЕРБАНК ТЕХНОЛОГИИ

public interface Callable<V>

По сравнению с интерфейсом Runnable обладаем методом call, который позволяет вернуть результат выполненной задачи

- V call() throws Exception

THREAD POOLS: ИНТЕРФЕЙС EXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

public interface ExecutorService extends Executor

Расширяет интерфейс Executor и определяет методы для управления процессом и завершения работы задач, переданных в сервис.

THREAD POOLS: МЕТОДЫ EXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

Постановка задачи на исполнение:

```
Future<?> submit(Runnable task);

<T> Future<T> submit(Callable<T> task);
```

THREAD POOLS: МЕТОДЫ EXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

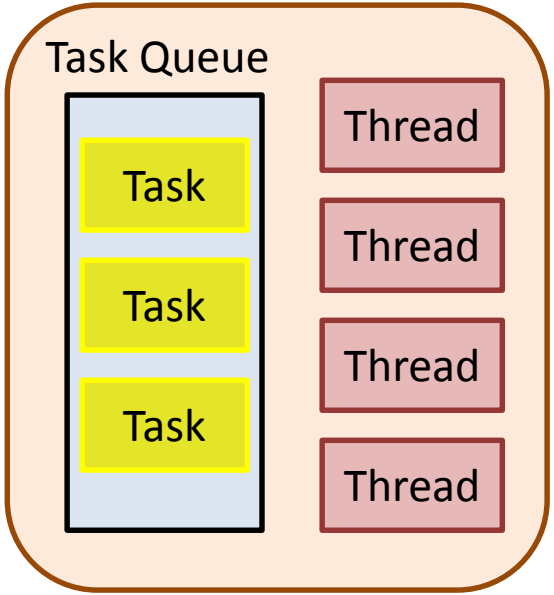
Завершение работы пула потоков:

```
void shutdown();

List<Runnable> shutdownNow();

boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
```

СБЕРБАНК ТЕХНОЛОГИИ

public class ThreadPoolExecutor extends AbstractExecutorService

Реализация интерфейса ExecutorService, выполняет задачу в одном из свободных на момент запуска задачи потоке.



THREAD POOLS: КЛАСС THREADPOOLEXECUTOR

СБЕРБАНК ТЕХНОЛОГИИ

Объект класса можно создать через один из четырех конструкторов.

Каждый конструктор можно специфицировать, передав соответствующие параметры.

THREAD POOLS: КЛАСС THREADPOOLEXECUTOR

СБЕРБАНК ТЕХНОЛОГИИ

Основные из них:

1) Количество постоянно живущих потоков
2) Максимальное количество потоков
3) Время жизни потока
4) Очередь для хранения задач
5) Фабрика для создания новых потоков
6) Обработчик, используется когда очередь заполнена и все потоки заняты

THREAD POOLS: КЛАСС EXECUTORS

СБЕРБАНК ТЕХНОЛОГИИ

Для удобной работы с пулами потоков лучше использовать класс Executors.

Класс, содержащий множество статичных методов, которые позволяют получить пулы потоков с различными конфигурациями

THREAD POOLS: МЕТОДЫ EXECUTORS

СБЕРБАНК ТЕХНОЛОГИИ

Основные методы:

```
ExecutorService newSingleThreadExecutor() {...}

ExecutorService newFixedThreadPool(int nThreads) {...}

ExecutorService newCachedThreadPool() {...}

ScheduledExecutorService newSingleThreadScheduledExecutor()
{...}

ScheduledExecutorService
            newScheduledThreadPool(int corePoolSize) {...}
```

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Определим класс задачи, которую необходимо выполнить

```
public class Task implements Runnable {
    @Override
    public void run() {
        // вывести текст на экран
    }
}
```

THREAD POOLS: РАБОТА С ПУЛОМ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

Запустим задачу в отдельном потоке:

```
public class TaskTest {
    public static void main(String []args)
            throws Exception {
        Runnable task = new Task();
        ExecutorService executorService =Executors
                .newSingleThreadExecutor();
        executorService.submit(task);
        executorService.shutdown();
    }
}
```

THREAD POOLS: ИНТЕРФЕЙС SCHEDULEDEXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

ScheduledExecutorService расширяет интерфейс ExecutorService и определяет методы, позволяющие запустить задачу через некоторое время или с определенной периодичностью

THREAD POOLS: МЕТОДЫ SCHEDULEDEXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

Основные методы:

```
ScheduledFuture<?> schedule(Runnable command,
                            long delay, TimeUnit unit);

<V> ScheduledFuture<V> schedule(Callable<V> callable,
                                long delay, TimeUnit unit);
```

THREAD POOLS: МЕТОДЫ SCHEDULEDEXECUTORSERVICE

СБЕРБАНК ТЕХНОЛОГИИ

Основные методы:

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                       long initialDelay,
                                       long period,
                                       TimeUnit unit);

ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                          long initialDelay,
                                          long delay,
                                          TimeUnit unit);
```

THREAD POOLS:КЛАСС SCHEDULEDTHREADPOOLEXECUTOR

СБЕРБАНК ТЕХНОЛОГИИ

ScheduledThreadPoolExecutor - реализация интерфейса ScheduledExecutorService.

Объект класса ScheduledThreadPoolExecutor можно создать используя один из 4-х конструкторов, передав в качестве аргумента в конструктор следующие параметры:

1) Количество постоянно живущих потоков
2) Фабрику для создания потоков
3) Обработчик, используется когда очередь заполнена и все потоки заняты

СБЕРБАНК ТЕХНОЛОГИИ

Создадим задачу

```
public class Reminder implements Runnable {
    public void run() {
        // вывод текста на экран
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

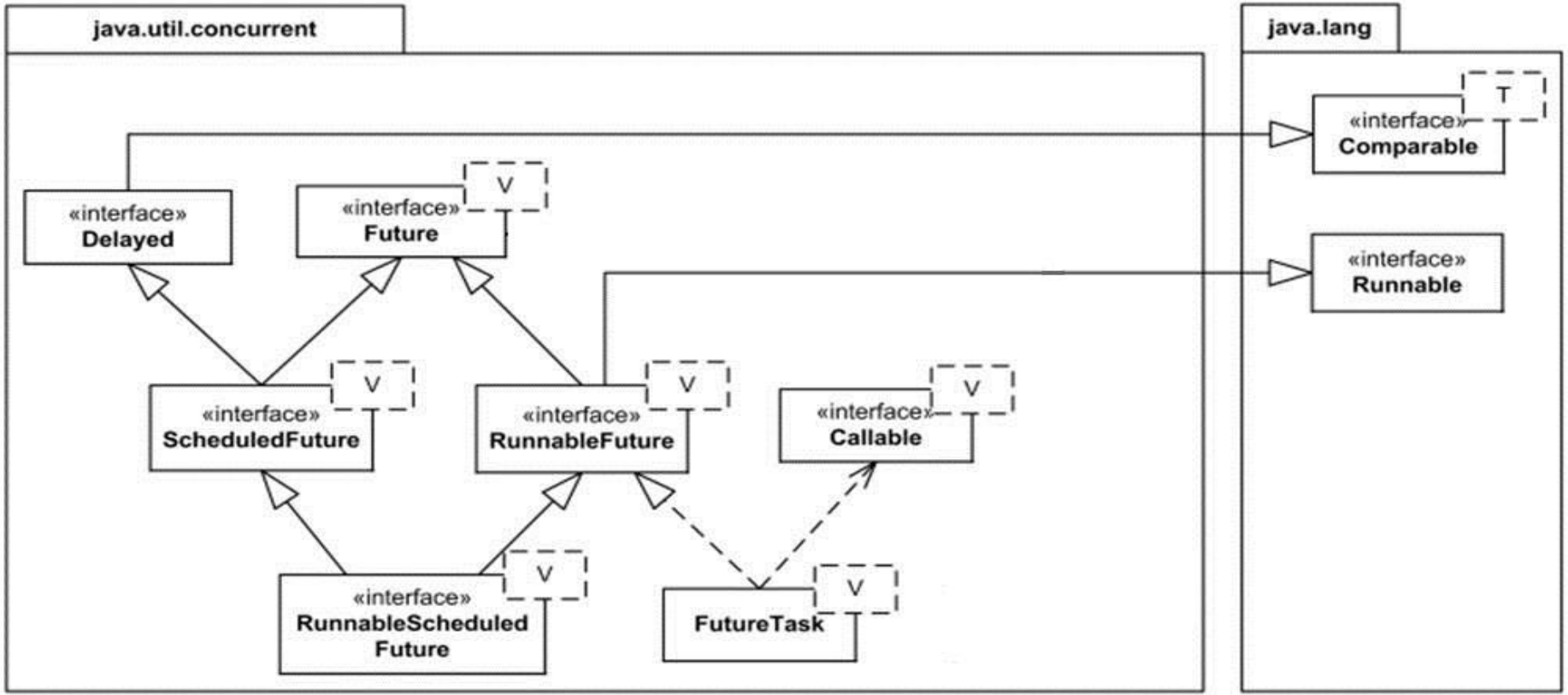
Передадим задачу в пул потоков на выполнение

```
public class ReminderMessenger {
    private static ScheduledExecutorService service =
            Executors.newScheduledThreadPool(1);

    public static void main(String args[]) {
        Reminder reminder = new Reminder();
        service.scheduleAtFixedRate(reminder, 0, 24,
                                    TimeUnit.HOURS);
    }
}
```

Концепция Future предусматривает обращение за результатом, вычисление которого может быть не завершено на данный момент

FUTURES

СБЕРБАНК ТЕХНОЛОГИИ



FUTURES:ИНТЕРФЕЙС FUTURE<V>

СБЕРБАНК ТЕХНОЛОГИИ

Future представляет результат асинхронного выполнения.

Методы позволяют определить завершено ли выполнение задачи и получить результат выполнения

```
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
boolean isDone();
V get() throws InterruptedException, ExecutionException;
V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
               TimeoutException;
```

FUTURES:КЛАСС FUTURETASK<V>

СБЕРБАНК ТЕХНОЛОГИИ

RunnableFuture - Future которая также является Runnable.

FutureTask - реализация интерфейса RunnableFuture.

```
public FutureTask(Callable<V> callable) {...}

public FutureTask(Runnable runnable, V result) {...}
```

Определим задачу:

```
public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        return Thread.currentThread().getName();
    }
}
```

FUTURES:РАБОТА С FUTURETASK<V>

СБЕРБАНК ТЕХНОЛОГИИ

Запустим задачу на выполнение:

```
MyCallable callable = new MyCallable();
FutureTask<String> futureTask =
                        new FutureTask<String>(callable);
ExecutorService executor =
                        Executors.newSingleThreadPool();
executor.execute(futureTask);
```

FUTURES:РАБОТА С FUTURETASK<V>

СБЕРБАНК ТЕХНОЛОГИИ

Получение результата:

```
if (!futureTask.isDone()) {
    //wait indefinitely for future task to complete
    System.out.println("FutureTask output=" +
                                futureTask.get());
}
```

В java 6 появился интерфейс для реализаций внешней блокировки.

```
public interface Lock {
    void lock();
    void lockInterruptibly()
            throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Реализация в JDK через ReentrantLock

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // update object state
    // catch exceptions and restore
                    invariants if necessary
} finally {
    lock.unlock();
}
```

Гарантии:

- Все реализации должны гарантировать mutual exclusion как и внутренний лок
- Все реализации должны предоставлять memory visibility симантику как и внутренний лок

Возможности:

- Захват/освобождение – явные
- Можно пробовать захватить с помощью try
- Можно пробовать заданное время
- Прерываемые методы захвата

Возможности:

- Захват/освобождение – явные
- Можно пробовать захватить с помощью try
- Можно пробовать заданное время
- Прерываемые методы захвата
- С помощью try lock может избежать deadlock
- Прерываемость позволяет участвовать в механизме прерывания потока
- Производительности сравнима с производительностью внутреннего лока
- Есть возможность «справедливой» блокировки

Опасности:

- Явный unlock (забыли – лок на вечно)

Итог:

Использовать ReentrantLock нужно только в тех случаях где нужны его преимущества, в остальных случаях нужно использовать synchronized.

READ WRITE LOCKS

СБЕРБАНК ТЕХНОЛОГИИ

В некоторых случаях операции чтения превалируют над записью по частоте.

Можно разрешит параллельное чтение, и блокировать только при записи.

READ WRITE LOCKS

СБЕРБАНК ТЕХНОЛОГИИ

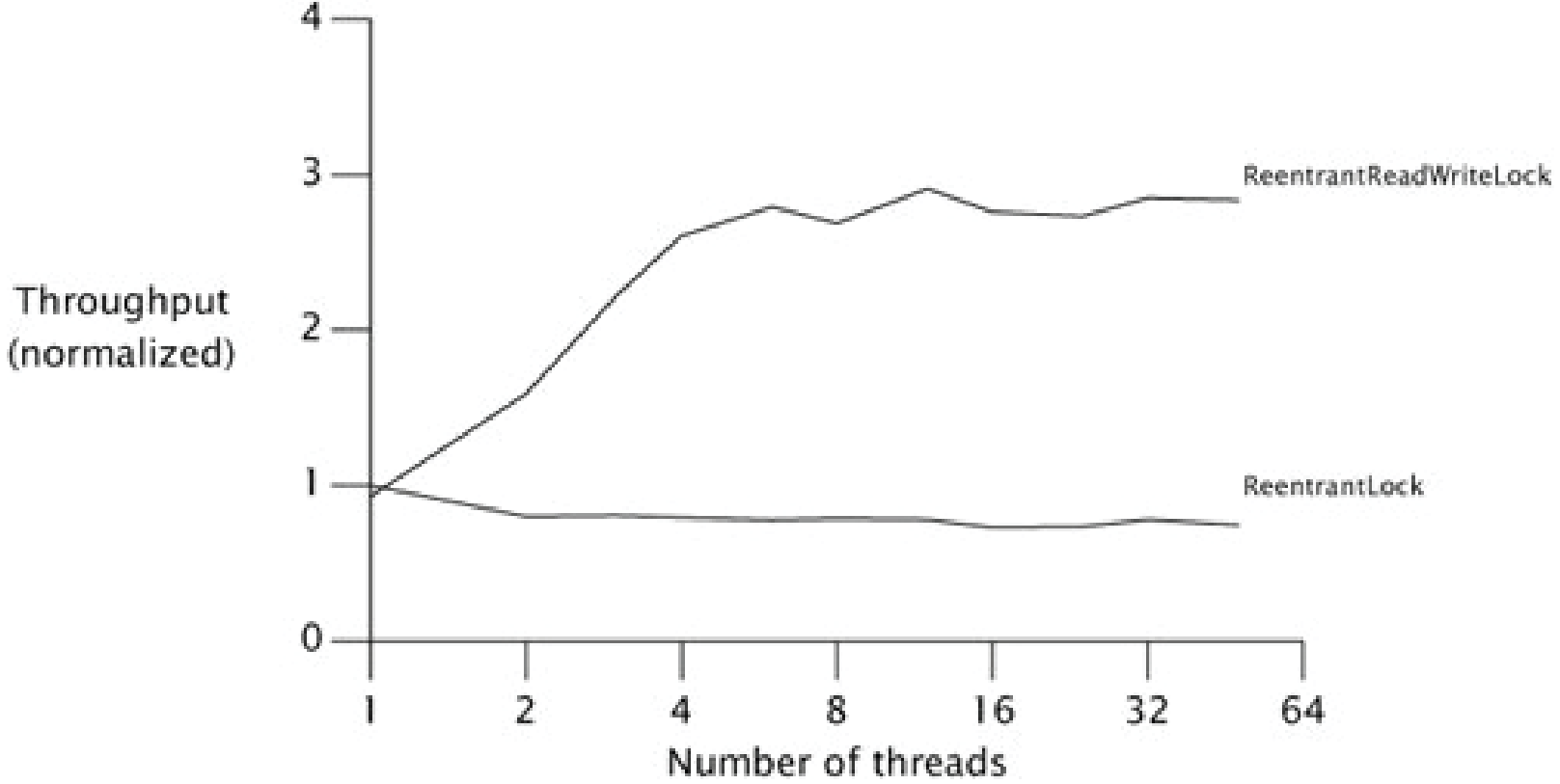
В concurrent пакете есть решение.

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

И реализация - ReentrantReadWriteLock

СБЕРБАНК ТЕХНОЛОГИИ

Сравнение по производительности ReentrantLock для read/write - 50/50



СБЕРБАНК ТЕХНОЛОГИИ

Пример:

Создадим простой потокобезопасный враппер для map.

```
public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock =
                            new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }
```

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Положить в коллекцию:

```
public V put(K key, V value) {
     w.lock();
     try {
          return map.put(key, value);
     } finally {
          w.unlock();
     }
}
```

ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Взять из коллекции:

```
public V get(Object key) {
     r.lock();
     try {
          return map.get(key);
     } finally {
          r.unlock();
     }
}
```

ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПОТОКАМИ

СБЕРБАНК ТЕХНОЛОГИИ

Синхронизаторы (synchronizer) – объекты, позволяющие организовать взаимодействие между потоками опираясь на их состояния.

Виды в java:

- latches
- Semaphores
- Barriers
- Blocking queues

ЗАЩЁЛКИ - LATCHES

СБЕРБАНК ТЕХНОЛОГИИ

Latches – позволяют заблокироваться в потоке до наступления терминального состояния.

ЗАЩЁЛКИ - LATCHES

СБЕРБАНК ТЕХНОЛОГИИ

Latches позволяют решать следующие задачи:

- Дождаться завершения инициализации ресурса, необходимого для дальнейшей работы
- Дождаться завершения старта сервиса, от которого зависим
- Дождаться пока все участники не подключатся

LATCHES - COUNTDOWNLATCH

СБЕРБАНК ТЕХНОЛОГИИ

Одна из реализаций защёлки – CountDownLatch.

CountDownLatch позволяет одному или нескольким потокам дождаться сета событий.

LATCHES - COUNTDOWNLATCH

СБЕРБАНК ТЕХНОЛОГИИ

```
public class CountDownLatch {

    public CountDownLatch(int count) {...}

    public void await() throws InterruptedException
{...}
    public boolean await(long timeout, TimeUnit unit)
            throws InterruptedException {...}
    public void countDown() {...}
    public long getCount() {...}
}
```

COUNTDOWNLATCH ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Реализуем одновременный запуск n потоков и ожидание их завершения.

Для этого создадим два управляющих CountDownLatch:

```
final CountDownLatch startGate = new CountDownLatch(1);
final CountDownLatch endGate = new CountDownLatch(nThreads);
```

COUNTDOWNLATCH ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Запустим n потоков со следующим телом:

```
public void run() {
    try {
        startGate.await();
        try {
            task.run();
        } finally {
            endGate.countDown();
        }
    } catch (InterruptedException ignored) { }
}
```

COUNTDOWNLATCH ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Управление из другого потока:

```
startGate.countDown(); // signal for start
endGate.await(); // wait until ending all threads
```

FutureTask так же ведёт себя как latch.

Возможны 3 состояния:

1. Waiting to run
2. Running
3. Completed

FUTURETASK - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Загрузим данные которые нужны будут позже:

Создадим FutureTask с бизнес логикой:

```
private final FutureTask<ProductInfo> future =
        new FutureTask<ProductInfo>(
             new Callable<ProductInfo>() {
                  public ProductInfo call()
                                  throws DataLoadException {
                return loadProductInfo();
             }
        });
```

FUTURETASK - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Запустим задачу в отдельном потоке и дождёмся завершения:

```
private final Thread thread = new Thread(future);
public void start() { thread.start(); }
public ProductInfo get()
        throws DataLoadException, InterruptedException {
    return future.get();
}
```

LATCHES – SEMAPHORE

СБЕРБАНК ТЕХНОЛОГИИ

Semaphore – используется для контроля кол-ва конкурентного доступа к некоторому ресурсу или выполнению некоторого действия в одно и тоже время.

Случаи использования:

- Пул ресурсов, например соединения к БД
- Bounded коллекции

СБЕРБАНК ТЕХНОЛОГИИ

Основные методы Semaphore:

```
public class Semaphore implements java.io.Serializable {

    public Semaphore(int permits) {...}
    public void acquire() throws InterruptedException {...}
    public void acquireUninterruptibly() {...}
    public boolean tryAcquire() {...}
    public boolean tryAcquire(long timeout, TimeUnit unit)
            throws InterruptedException {...}
    public void release() {...}
}
```

SEMAPHORE -ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Реализуем bounded hash set:

```
public class BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet
                                    (new HashSet<T>());
        sem = new Semaphore(bound);
    }
```

SEMAPHORE -ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Добавление элемента:

```
public boolean add(T o) throws InterruptedException {
     sem.acquire();
     boolean wasAdded = false;
     try {
          wasAdded = set.add(o);
          return wasAdded;
     }
     finally {
          if (!wasAdded)
               sem.release();
     }
}
```

SEMAPHORE -ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Удаление элемента:

```
public boolean remove(Object o) {
    boolean wasRemoved = set.remove(o);
    if (wasRemoved)
        sem.release();
    return wasRemoved;
}
```

LATCHES – CYCLICBARRIER

СБЕРБАНК ТЕХНОЛОГИИ

CyclicBarrier – блокируют фиксированную группу участников в общей точке, до тех пор пока вся группа не дойдёт до этой точки.

Может использоваться повторно!

Случаи использования:

Обычно для задач, где группа подзадач может выполняться параллельно в фазе 1, но перейти в фазу 2 могут только когда вся группа завершит фазу 1.

Основные методы:

```
public class CyclicBarrier {

    public CyclicBarrier(int parties) {...}

    public int await() throws InterruptedException,
                                  BrokenBarrierException {...}

    public int await(long timeout, TimeUnit unit)
            throws InterruptedException,
            BrokenBarrierException,
            TimeoutException {...}
}
```

CYCLICBARRIER - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Запустим все задачи одновременно:

```
public class StartAllThreadSameTime implements Runnable {
    private final CyclicBarrier barrier =
                                        new CyclicBarrier(10);
    @Override
    public void run() {
        // do work in phase 1
        barrier.await();
        // do work in phase 2
        barrier.await();
        // do work in phase 3
        ...
    }
}
```

ATOMICS:ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Пакет java.util.concurrent.atomic содержит множество классов, методы которых позволяют осуществлять атомарные операции.

ATOMICS:ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Пакет java.util.concurrent.atomic содержит множество классов, методы которых позволяют осуществлять атомарные операции.

Операция в общей области памяти называется **атомарной**, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти.

ATOMICS:ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Пакет java.util.concurrent.atomic содержит множество классов, методы которых позволяют осуществлять атомарные операции.

Операция в общей области памяти называется **атомарной**, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти.

Во время выполнения такой операции над переменной, ни один поток не может наблюдать изменение наполовину завершенным.

ATOMICS:ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Пакет java.util.concurrent.atomic содержит множество классов, методы которых позволяют осуществлять атомарные операции.

Операция в общей области памяти называется атомарной, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти.

Во время выполнения такой операции над переменной, ни один поток не может наблюдать изменение наполовину завершенным.

Атомарная загрузка гарантирует, что переменная будет загружена целиком в один момент времени. Неатомарные операции не дают такой гарантии.

ATOMICS:КЛАССЫ

СБЕРБАНК ТЕХНОЛОГИИ

Основные классы в JDK:

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicIntegerArray
- AtomicLongArray
- AtomicReference<V>
- AtomicReferenceArray<E>

ATOMICS:НЕ АТОМАРНЫЕ ДЕЙСТВИЯ

СБЕРБАНК ТЕХНОЛОГИИ

```
class Book {
    int copiesSold = 0;

    public void newSale() {
        ++copiesSold;
    }

    public void returnBook() {
        --copiesSold;
    }
}
```

Какие проблемы у этого кода?

ATOMICS:НЕ АТОМАРНЫЕ ДЕЙСТВИЯ

СБЕРБАНК ТЕХНОЛОГИИ

```
class Book {
    int copiesSold = 0;

    public void newSale() {
        ++copiesSold;
    }

    public void returnBook() {
        --copiesSold;
    }
}
```

Какие проблемы у этого кода?

Неатомарные выражения, включающие загрузку значения из памяти, манипуляции со значениями и загрузку обратно в память

ATOMICS:ПРИМЕНЕНИЕ ATOMIC-КЛАССА

СБЕРБАНК ТЕХНОЛОГИИ

```
class Book {
    AtomicInteger copiesSold = new AtomicInteger(0);

    public void newSale() {
        copiesSold.incrementAndGet();
    }

    public void returnBook() {
        copiesSold.decrementAndGet();
    }
}
```

Атомарные операции тесно взаимосвязаны с операцией CAS(Compare and Swap)

Операция CAS включает 3 объекта-операнда:

1. адрес ячейки памяти
2. ожидаемое старое значение
3. новое значение

Процессор атомарно обновляет адрес ячейки, если новое значение совпадает со старым, иначе изменение не зафиксируется.

ATOMICS:ТИПОВЫЕ МЕДОТЫ НА ОСНОВЕ CAS

СБЕРБАНК ТЕХНОЛОГИИ

```
public final boolean compareAndSet(int expect, int update)

public final int addAndGet(int delta)

public final int getAndDecrement()

public final int getAndSet(int newValue)

public final int incrementAndGet()
```

СИНХРОННЫЕ КОЛЛЕКЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

Проблемы синхронных коллекций:

- Производительность
- ConcurrentModificationException при итерировании
- Дополнительная синхронизация для составных действий
  - ✓ Итерирование
  - ✓ Навигация
  - ✓ Условные (put if absent)

СИНХРОННЫЕ КОЛЛЕКЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

Проблемы синхронных коллекций:

- Производительность
- ConcurrentModificationException при итерировании
- Дополнительная синхронизация для составных действий
  - ✓ Итерирование
  - ✓ Навигация
  - ✓ Условные (put if absent)
- При итерировании слишком на долго лочим коллекцию
- Риск dead lock-а при использовании дополнительной синхронизации
- Неявность в коде где нужна дополнительная синхронизация

СКРЫТОЕ ИТЕРИРОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Рассмотрим пример неявности необходимости в дополнительной блокировки:

```
public class HiddenIterator {
    private final Set<Integer> set = new HashSet<Integer>();
    public synchronized void add(Integer i) {set.add(i);}
    public synchronized void remove(Integer i) {set.remove(i);}
    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to «
                                                    + set);
    }
}
```

CONCURRENT КОЛЛЕКЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

Для решения проблем обычных синхронных коллекций в java 5 и 6 были введены конкурентные коллекции.

Синхронная коллекция	Конкурентный аналог
Synchronized hash map	ConcurrentHashMap
Synchronized List	CopyOnWriteArrayList
Synchronized SortedMap map	ConcurrentSkipListMap
Synchronized SortedSet	ConcurrentSkipListSet

COPYONWRITEARRAYLIST

СБЕРБАНК ТЕХНОЛОГИИ

Важно:

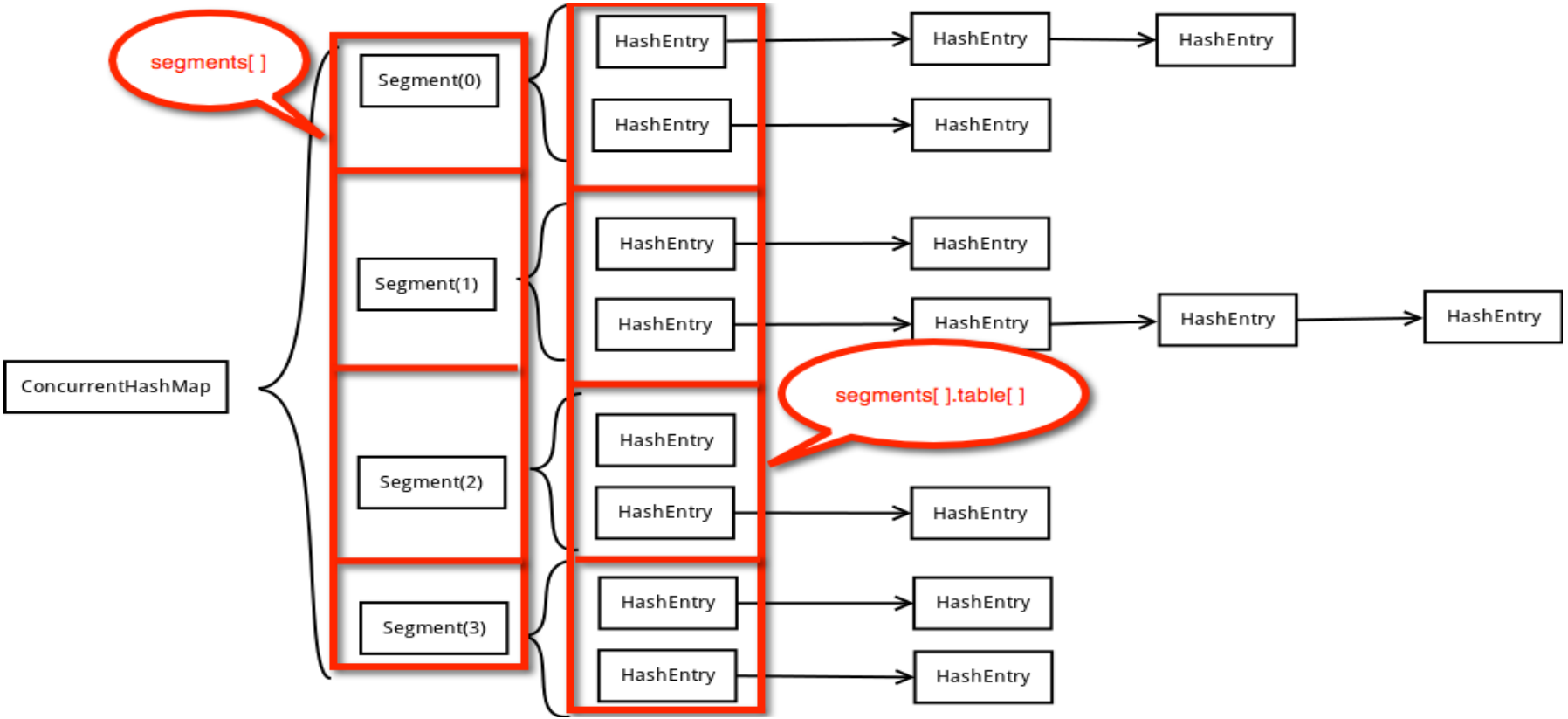
- Во время модификации создаётся копия
- Итератор так же ссылается на копию массива, полученную во время его создания
- Итератор не выкидывает ConcurrentModificationException
- Эффективен в случае преобладания read операций над write

ConcurrentHashMap – конкурентный аналог HashMap

CONCURRENTHASHMAP

СБЕРБАНК ТЕХНОЛОГИИ

Как устроен:



CONCURRENTHASHMAP

СБЕРБАНК ТЕХНОЛОГИИ

Что даёт:

- Операции чтения НЕ блокируются
- Операции чтения возвращают наиболее актуальный результат
- Итератор отражает состояние коллекции на момент создания
- Итератор не кидает ConcurrentModificationException
- Поддерживает составные операции (put-if-absent, remove-if-equal and replace-if-equal)

ПОТОКОБЕЗОПАСНЫЕ ОЧЕРЕД

СБЕРБАНК ТЕХНОЛОГИИ

Так же в java 5 ввели: Queue и BlockingQueue.

Очередь предназначена для временного хранения элементов во время их процессинга.

Queue – все операции НЕ блокирующие.

BlockingQueue – операции вставки и получения результата БЛОКИРУЮЩИЕ

QUEUE – ОСНОВНЫЕ ОПЕРАЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

ОСНОВНЫЕ РЕАЛИЗАЦИИ QUEUE

СБЕРБАНК ТЕХНОЛОГИИ

ConcurrentLinkedQueue – неограниченная потокобезопасная очередь FIFO, построенная на "wait-free" алгоритмах.

ОСНОВНЫЕ РЕАЛИЗАЦИИ QUEUE

СБЕРБАНК ТЕХНОЛОГИИ

ConcurrentLinkedQueue – неограниченная потокобезопасная очередь FIFO, построенная на "wait-free" алгоритмах.

- Гарантирует insert элемента happen-before delete
- Итератор weakly consistent – не выкидывает ConcurrentModificationException, отражает состояние на момент создания
- Size НЕ эффективная операция
- Не гарантирована атомарность групповых операций (addAll, removeAll,…)

BLOCKINGQUEUE – ОСНОВНЫЕ ОПЕРАЦИИ

СБЕРБАНК ТЕХНОЛОГИИ

	Throws exception	Returns special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	-	-

ОСНОВНЫЕ РЕАЛИЗАЦИИ BLOCKINGQUEUE

СБЕРБАНК ТЕХНОЛОГИИ

- LinkedBlockingQueue – неограниченная потокобезопасная очередь

ОСНОВНЫЕ РЕАЛИЗАЦИИ BLOCKINGQUEUE

СБЕРБАНК ТЕХНОЛОГИИ

- LinkedBlockingQueue – неограниченная потокобезопасная очередь
- ArrayBlockingQueue – потокобезопасная очередь фиксированного размера

ОСНОВНЫЕ РЕАЛИЗАЦИИ BLOCKINGQUEUE

СБЕРБАНК ТЕХНОЛОГИИ

- LinkedBlockingQueue – неограниченная потокобезопасная очередь
- ArrayBlockingQueue – потокобезопасная очередь фиксированного размера
- PriorityBlockingQueue – неограниченная потокобезопасная очередь с приоритетами

ОСНОВНЫЕ РЕАЛИЗАЦИИ BLOCKINGQUEUE

СБЕРБАНК ТЕХНОЛОГИИ

- LinkedBlockingQueue – неограниченная потокобезопасная очередь
- ArrayBlockingQueue – потокобезопасная очередь фиксированного размера
- PriorityBlockingQueue – неограниченная потокобезопасная очередь с приоритетами
- SynchronousQueue – потокобезопасная очередь фиксированного размера, где каждый вызов insert блокируется до соответствующего remove из другого потока и наоборот

BLOCKINGQUEUE И PRODUCER CONSUMER PATTERN

СБЕРБАНК ТЕХНОЛОГИИ

Классический пример использования blocked queue – это реализация шаблона поставщик – потребитель.

```
class Producer {
    private final BlockingQueue queue;
    Producer(BlockingQueue q) {
        queue = q;
    }
    public void doProduce() {
        try {
            while (true) {
                queue.put(produce());
            }
        } catch (InterruptedException ex) { ...handle ...}
    }
}
```

BLOCKINGQUEUE И PRODUCER CONSUMER PATTERN

СБЕРБАНК ТЕХНОЛОГИИ

```
class Consumer {
    private final BlockingQueue queue;
    Consumer(BlockingQueue q) {
        queue = q;
    }
    public void doConsume() {
        try {
            while (true) {
                consume(queue.take());
            }
        } catch (InterruptedException ex) { ...handle ...}
    }
}
```

ЗАДАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Модифицировать кэш, разработанный на лекции 6 и 9, сделать его конкурентным. Должна быть возможность не лочить весь кэш, а только тот элемент который в данный момент добавляется или удаляется.