Лабораторна робота № 6

Розробка програм з використанням засобів Java Concurrency API

Мета роботи: Вивчити засоби Java Concurrency API для побудови багатопотокових програм.

1. Теоретичні відомості

Concurrency API був введений в Java 5 з метою надання прикладним програмістам зручних інструментів побудови багатопотокових програм. Такі інструменти розміщені у пакеті java.util.concurrent і можуть бути поділені за функціональними ознаками на такі (Рис. 1):

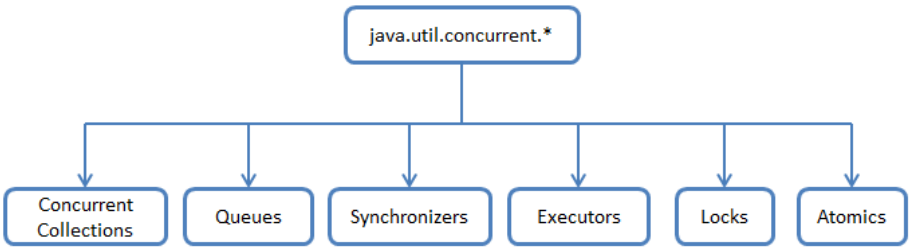

Рис. 1. Інструменти Java Concurrency API

Concurrent Collections - набір колекцій, що більш ефективно працюють в багатопотоковому середовищі ніж стандартні універсальні колекції з java.util пакету. На відміну від засобів багатопотокового доступу стандартних колекцій, коли блокується доступ доступу до всієї колекції, використовуються блокування сегментів даних або ж оптимізується робота для паралельного читання даних.

Queues - блокуючі та неблокуючі черги з підтримкою багатопоточності. Блокуючі черги використовуються, коли потрібно «пригальмувати» потоки постачальника або споживача, якщо не виконані певні умови, наприклад, черга порожня або переповнена, або ж немає вільного споживача. Неблокуючі черги забезпечують швидкість роботи потоків.

Synchronizers - допоміжні утиліти для синхронізації потоків, що дозволяють розробнику управляти та/або обмежувати роботу декількох потоків.

Executors - засоби створення пулів потоків і планування роботи асинхронних завдань з отриманням результатів.

Locks - являє собою альтернативні і більш гнучкі механізми синхронізації потоків в порівнянні з базовими `synchronized`, `wait`, `notify`, `notifyAll`.

Atomics - класи з підтримкою атомарних операцій над примітивами і посиланнями.

Створення потоку для кожного завдання вимагає значних комп'ютерних ресурсів. Певним виходом з такої ситуації є використання *пулів потоків* (*Thread pool*). Переваги використання пулів потоків:

- підвищення ефективності програми за рахунок повторного використання потоків (зниження витрат процесорного часу і пам'яті);
- кращий дизайн програми, що дозволяє зосередитися на логіці роботи програми, а не на технічних засобах організації її паралельного виконання.

Основна ідея використання пулу потоків полягає в наступному (Рис. 2):
- коли програмі необхідно виконати деяку задачу, замість створення потоку для виконання цього завдання і передачі йому завдання, вона просто розміщує завдання у *черзі* (*Job queue*);
- а один з працюючих в нескінченному циклі потоків з *пулу потоків* (*Thread pool*) вибирає завдання з черги і виконує його.

Об'єкти, які реалізують такі функції, відомі як *виконавці* (*Excecutors*).

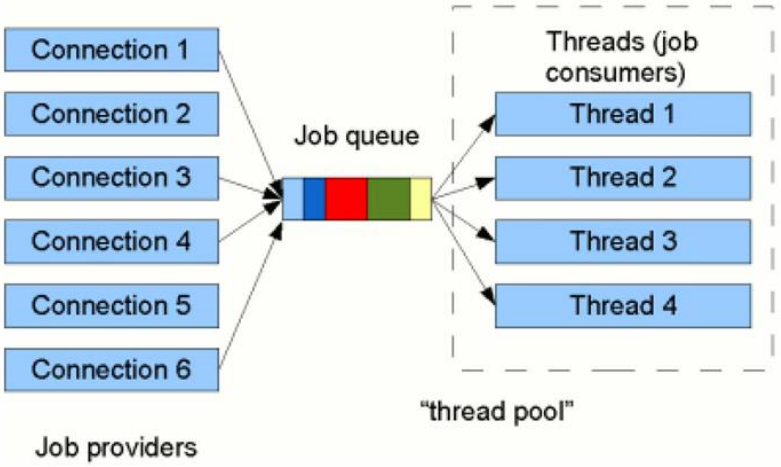

Рис. 2. Використання пулу потоків

Пакет java.util.concurrent визначає три інтерфейси з функціональністю пулів потоків (Рис. 3):

Executor - простий інтерфейс, який підтримує запуск нових завдань.

ExecutorService - успадковує Executor, додаючи функції, що забезпечують управління життєвим циклом, як індивідуальних завдань, так і самого виконавця.

ScheduledExecutorService - успадковує ExecutorService, даний інтерфейс додає можливість запускати відкладені завдання.

Як правило, об'єкти реалізацій виконавців оголошуються як об'єкти одного з цих трьох типів інтерфейсів, а не як об'єкти класів-реалізацій.

Інтерфейс Executor містить єдиний метод `void` **`execute`**`(Runnable r)`. Об'єкт Executor буде використовувати існуючий робочий потік з пулу потоків для запуску завдання `r`, або помістить `r` в чергу у разі, якщо жоден робочий потік недоступний.

Інтерфейс ExecutorService містить методи, які беруть в якості параметра як Runnable об'єкти, так і Callable об'єкти (останні дозволяють повернути з завдання результат його виконання). Найбільш часто використовуваними є методи `Future<?>`**`submit`**`(Runnable task)`, `<T> Future<T>` **`submit`**`(Runnable task, T result)`, `<T> Future<T>` **`submit`**`(Callable<T> task)`, які

повертають об'єкт Future, що використовується для отримання результату виконаного у потоці завдання або для отримання статусу завдання.

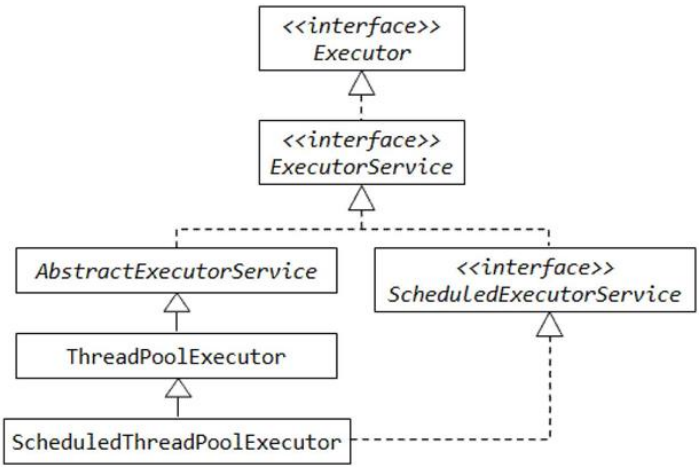

Рис. 3. Ієрархія виконавців пулів потоків

Callable<V> - аналог інтерфейсу Runnable для завдань, які повертають значення, отримане при виконанні завдання у потоці (нагадаємо, що метод `void run()` інтерфейсу Runnable нічого не повертає). Єдиним методом Callable<V> є метод `V call()`, що дозволяє повертати типізоване значення. Callable-завдання, як і Runnable-завдання можуть бути передані об'єктам ExecutorService. як аргументи методу `submit`. Але як тоді отримати результат, який вони повертають? Оскільки метод `submit` не чекає завершення завдання, об'єкт ExecutorService не може повернути результат завдання безпосередньо. Замість цього він повертає спеціальний об'єкт Future, у якого можливо запросити результат виконання завдання (таке виконання називають *асинхронним*).

Future<V> - інтерфейс для отримання результатів виконання завдання у потоках пулу. Основними методами є `V get()` та `V get(long, TimeUnit)`, які очікують завершення завдання та повертають значення результату виконаного завдання (другий метод чекає максимум впродовж часу, вказаного як аргумент). Також існують методи для скасування виконання завдання `boolean cancel(boolean mayInterruptIfRunning)` і перевірки чи виконане завдання `boolean isDone()` та чи скасоване виконання завдання до його завершення `boolean isCancelled()`. Як імплементацію інтерфейсу Future<V> часто використовують клас FutureTask<V>.

Методи інтерфейсу ExecutorService `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)` та `<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` працюють зі списками завдань з блокуванням потоку до завершення всіх завдань в переданому списку або до закінчення заданого часу очікування, вони повертають список об'єктів Future з результатами та статусами відповідних завдань. Методи `<T> T invokeAny(Collection<? extends Callable<T>> tasks)`, `<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`

блокують потік до завершення будь-якого з переданих завдань або до закінчення заданого часу очікування, вони повертають результат виконаного завдання.

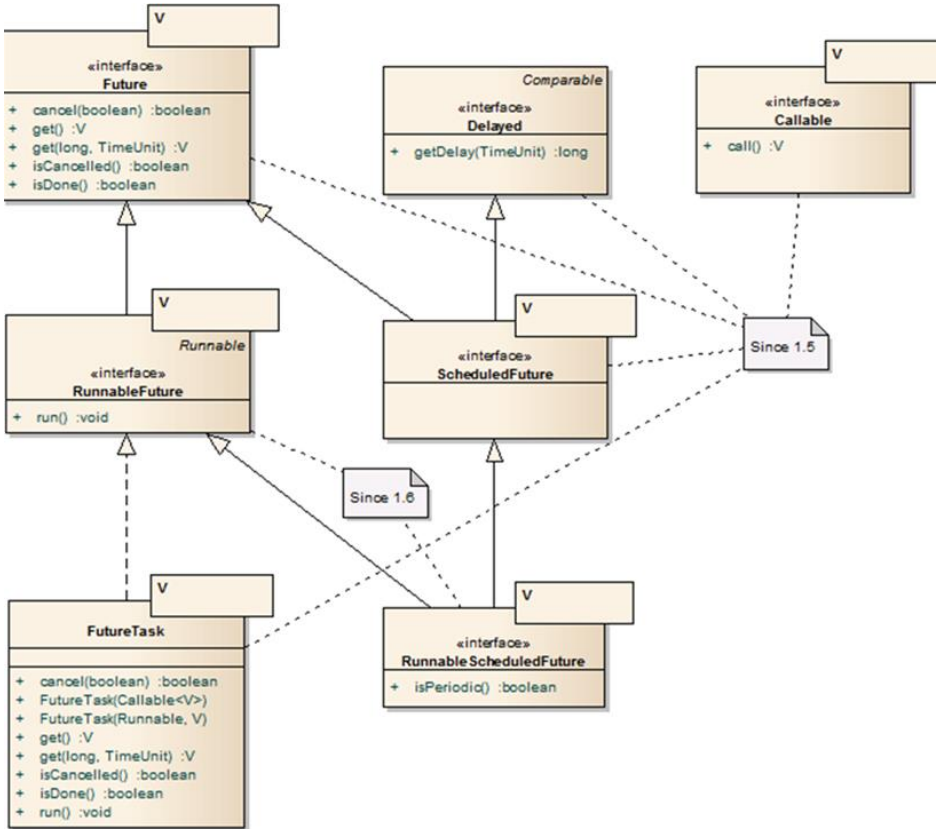

Рис. 4. Інтерфейси Callable<V> та Future<V>

Інтерфейс також містить метод `void shutdown()`, який завершить роботу ExecutorService після завершення усіх завдань, які виконувались у момент виклику `shutdown()`, та метод `List<Runnable> shutdownNow()`, який намагається зупинити усі завдання, що виконуються та очікують виконання та повертає список завдань, які буди у стані очікування. Після виклику методів `shutdown` об'єкт ExecutorService більше не братиме завдання, кидаючи RejectedExecutionException при спробі додати завдання. Також інтерфейс містить методи `boolean isShutdown()`, що повертає `true`, якщо поточний ExecutorService вимкнений, та `boolean isTerminated()`, що повертає `true`, якщо всі завдання були завершені після вимкнення ExecutorService. Метод `boolean awaitTermination(long timeout, TimeUnit unit)` блокує завершення ExecutorService після отримання команди `shutdown()`, поки всі завдання не будуть завершені або не сплине заданий як параметр час очікування або поточний потік не буде перерваний методом `interrupt()`, залежно від того, що трапиться раніше. Як імплементацію інтерфейсу ExecutorService часто використовують клас ThreadPoolExecutor.

ScheduledExecutorService успадковує ExecutorService та додає можливість запускати завдання через заданий як аргумент проміжок часу методами `<V> ScheduledFuture <V> schedule (Callable<V> callable, long delay, TimeUnit unit)` та `(ScheduledFuture <?> schedule (Runnable`

command, long delay, TimeUnit unit)). Також є методи ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) та ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit), які виконують періодичний запуск завдань - перший після первинної затримки з фіксованим періодом, а другий - після первинної затримки з фіксованим проміжком часу між завершенням попереднього завдання та початком наступного.

Для організації виконання завдання у потоці, що управляється об'єктом ExecutorService, необхідно:

1) реалізувати задачу в методі void run() об'єкта Runnable;

2) створити об'єкт Executor, керуючий пулом потоків, безпосередньо (в цьому випадку буде доступно більше опцій при створенні об'єкта) або з використанням одного з фабричних методів класу Executors. Пул потоків виконавця може мати один потік, фіксовану кількість потоків або необмежене число потоків (буде продемонстровано далі);

3) створити (отримати) робочий потік, виконуючий завдання шляхом виклику методу void execute(Runnable r), що додає завдання в чергу пулу потоків. Завдання буде заплановане і виконається доступним робочим потоком пулу.

Нехай є завдання вивести на екран деяку інформацію про це завдання:

```
public class MyTask implements Runnable {

    String taskInfo;

    public MyTask(String taskInfo) {
        this.taskInfo = taskInfo;
    }

    @Override
    public void run() {
        System.out.println(taskInfo);
    }
}
```

Тоді у деякому класі:

```
public static void main(String[] args) {
    ThreadPoolExecutor tpe = new ThreadPoolExecutor(5, 10,
 30L, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
    MyTask[] tasks = new MyTask[25];
    for (int i = 0; i < tasks.length; i++) {
        tasks[i] = new MyTask("Task-" + i);
        tpe.execute(tasks[i]);
    }
    tpe.shutdown();
}
```

Параметри конструктора ThreadPoolExecutor: розмір ядра пула потоків `corePoolSize = 5`, максимальний розмір пула `maximumPoolSize = 10`. Коли нове завдання передається в метод `execute`, і в цей час кількість потоків в пулі менше `corePoolSize`, створюється новий потік для цього завдання, навіть якщо інші потоки простоюють. Якщо кількість потоків в пулі більше `corePoolSize`, але менше `maximumPoolSize`, новий потік буде створений, лише якщо черга заповнена повністю. Період життя надлишкових потоків `keepAliveTime = 30 seconds` - якщо в пулі є більш `corePoolSize` потоків, надлишкові потоки будуть знищені, якщо вони не використовується протягом проміжку часу, більш ніж `KeepAliveTime`. Останній параметр конструктора - посилання на чергу для завдань `workQueue` (використовується колекція-черга, до якої можуть мати доступ декілька потоків LinkedBlockingQueue). Запуск програми виведе 25 рядків з інформацією про завдання, як правило, не в порядку їх запуску, оскільки планування потоків пов'язано з випадковістю:

Task-0
Task-2
Task-1
Task-7
Task-8
…

Якщо задати невелику початкову місткість для черги (останнього параметру конструктора), наприклад,

```
new LinkedBlockingQueue<Runnable>(5),
```

то можна спостерігати генерування виключення (якщо виключення немає, спробуйте підвищити кількість завдань):

```
java.util.concurrent.RejectedExecutionException: Task executors.threadpoolexecutor.MyTask@67f89fa3 rejected from java.util.concurrent.ThreadPoolExecutor@4ac68d3e[Running, pool size = 10, active threads = 9, queued tasks = 5, completed tasks = 6]
```

Ми бачимо, що розмір пулу складає 10 потоків після 6 виконаних завдань і заповненій повністю черзі - 5 завдань, і при доданні ще одного нового завдання виникає виключення.

API передбачає менш складний спосіб створення пулу потоків, аніж використання класу ThreadPoolExecutor - це використання класу-фабрики Executors, яка дозволяє створювати об'єкти ThreadPoolExecutor та ScheduledThreadPoolExecutor шляхом виклику одного зі статичних методів:

`newSingleThreadExecutor()` - створює об'єкт-реалізацію ExecutorService (ThreadPoolExecutor) з єдиним робочим потоком для завдання, автоматично замінюючи його, якщо потік несподівано завершується;

`newFixedThreadPool(int numThreads)` - створює об'єкт-реалізацію ExecutorService (ThreadPoolExecutor) зі зростаючою (при додаванні виконавцю завдань) кількістю робочих потоків, аж до заданої як параметр максимальної

кількості потоків. При досягненні максимального значення виконавець намагається зберегти розмір пулу постійним (додає нові потоки тільки у разі, якщо робочий потік завершується через несподіване виключення);

`newCachedThreadPool()` - створює об'єкт-реалізацію ExecutorService (ThreadPoolExecutor) з пулом потоків, який створює нові потоки (кількістю до Integer.MAX_VALUE), у міру необхідності, але буде повторно використовувати раніше створені потоки в разі, якщо вони доступні.

`newScheduledThreadPool(int corePoolSize)` - створює об'єкт-реалізацію ScheduledExecutorService (ScheduledThreadPoolExecutor) с пулом потоків, здатним запускати завдання з затримкою та/або періодично (`corePoolSize` - кількість потоків, які підтримуються в пулі навіть, якщо вони простоюють).

Продемонструємо приклад використання фабрики Executors для Callable-завдання, що повертає результат:

```
public class MyTestCallable implements Callable<String> {
    private int workNumber;
    public MyTestCallable(int workNumber) {
        this.workNumber = workNumber;
    }
    @Override
    public String call() {
        for (int i = 0; i < 5; i++) {
            System.out.printf("Work-%d: %d\n", workNumber,
                              i);
            try{
                Thread.sleep((long) (Math.random() * 1000));
            }catch(InterruptedException ex){
                System.out.println("Sleep interrupted.");
            }
        }
        return "work-" + workNumber;
    }
}
```

Завдання полягає у виведенні 5 разів номера завдання з номером ітерації циклу виведення, при цьому кожна наступна ітерація затримується на випадковий час від 0 до 1с. Повертає завдання рядок зі своїм номером.

Далі у деякому класі:

```
public static void main(String[] args) {
    int numOfWorks = 20;
    ExecutorService pool = Executors.newFixedThreadPool(4);
    MyTestCallable[] works = new MyTestCallable[numOfWorks];
    Future[] futures = new Future[numOfWorks];
    for (int i = 0; i < numOfWorks; ++i) {
```

```
            works[i] = new MyTestCallable(i + 1);
            futures[i] = pool.submit(works[i]);
        }
        for (int i = 0; i < numOfWorks; ++i) {
            try {
                System.out.println(futures[i].get() + " ended");
            } catch (InterruptedException | ExecutionException
                                                          ex) {
                ex.printStackTrace();
            }
        }
        pool.shutdown();
}
```

Пул з максимум 4 потоків створюється фабричним методом `newFixedThreadPool(4)` класу Executors. Запуск завдань на виконання здійснюється методом `submit`, що викликається з виконавця з фіксованим пулом потоків. Результат виконання кожного завдання розміщується в масиві об'єктів Future. Після чого в наступному циклі масив об'єктів Future опитується і методом `get` отримуються результати виконання завдань (вони закінчуються словом ended):

```
Work-2: 0
Work-4: 0
Work-3: 0
Work-1: 0
Work-1: 1
Work-3: 1
Work-4: 1
Work-2: 1
Work-4: 2
Work-1: 2
Work-4: 3
Work-2: 2
Work-3: 2
Work-4: 4
Work-2: 3
Work-3: 3
Work-1: 3
Work-1: 4
Work-5: 0
Work-5: 1
Work-6: 0
work-1 ended
…
```

Синхронізація потоків за допомогою `synchronized` та вбудованого механізму `wait/notify` має деякі функціональні обмеження:

- потік може очікувати доступу до критичної секції невизначений час і відсутні засоби впливу на це;
- не підтримується справедливість доступу потоків до критичної секції (у тому порядку, в якому вони його просили);
- неможливо перервати потік, який очікує блокування;
- неможливо опитувати або намагатися отримати блокування, не будучи готовим до довгого очікування.

В пакеті java.util.concurrent.locks розміщені інтерфейси та класи, що забезпечують блокування (доступ до критичних секцій коду) та очікування умов, відмінних від вбудованого механізму синхронізації `wait/notify` та моніторів. Ці засоби надають набагато більшу гнучкість у використанні блокувань та умов (Рис. 5).

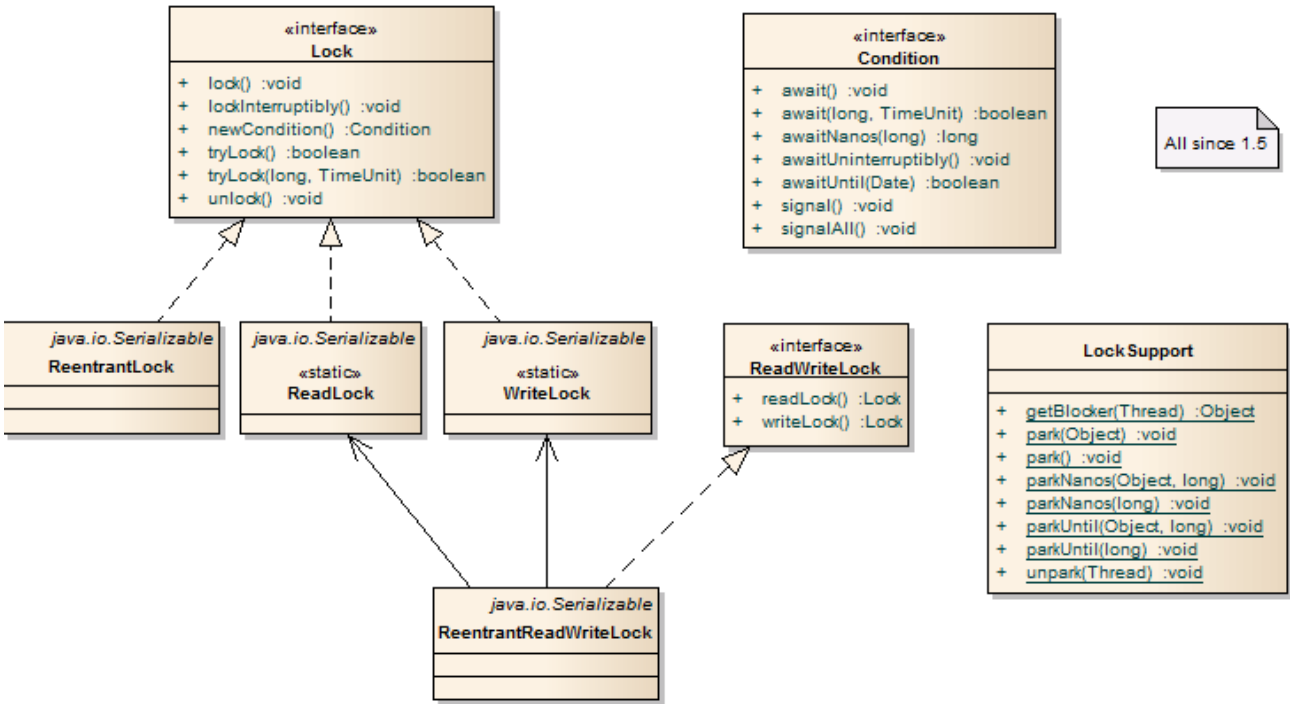

Рис. 5. Інтерфейси та класи пакету java.util.concurrent.locks

Lock - базовий інтерфейс, що надає більш гнучкий підхід щодо обмеження доступу до ресурсів/критичних секцій коду, ніж при використанні `synchronized`. Так, при використанні декількох об’єктів Lock, порядок їх вивільнення може бути довільним. Також є можливість використання альтернативного сценарію, якщо об’єкт Lock вже захоплений.

Condition - інтерфейс, який описує пов'язані з об’єктами Lock змінні, які можуть виконувати функції об'єкта, чий монітор використовується для доступу до критичних секцій кода, але пропонують розширені можливості. Зокрема, кілька об'єктів Condition можуть бути пов'язані з одним об’єктом Lock. Щоб уникнути проблем сумісності, імена методів Condition відрізняються від відповідних методів Object: `await/signal/signalAll`.

ReadWriteLock - інтерфейс підтримує пару пов'язаних блокувань: одну для читання і одну для запису, використовується для створення об’єктів ReentrantReadWriteLock. Такі об’єкти надзвичайно корисні, коли в системі багато операцій читання і мало операцій запису.

Найважливіші класи-реалізації описаних інтерфейсів це:

ReentrantLock - забезпечує можливість взаємного виключення потоків з такою самою базовою поведінкою та семантикою, що і блокування за допомогою монітора об’єкта, доступ до якого здійснюється за допомогою синхронізованих методів та операторів, але з розширеними можливостями. Тільки один потік може зайти в захищений блок. Клас підтримує *справедливий* (*fair*) або *несправедливий* (*non-fair*) *доступ* до критичних секцій. При справедливому блокуванні дотримується порядок звільнення потоків, що викликають `lock()`. При несправедливому розблокуванні порядок звільнення потоків не гарантується, але, як бонус,

таке розблокування працює швидше. За замовчуванням, використовується несправедливе розблокування.

ReentrantReadWriteLock - дуже часто використовується в багатопотокових сервісах і кешах, показуючи дуже хороший приріст продуктивності в порівнянні з блоками `synchronized`. По суті, клас працює в двох взаємовиключних режимах: багато читачів одночасно читають дані і тільки один записувач записує дані.

ReentrantReadWriteLock.ReadLock - ReadLock для читачів, одержуваний через `readWriteLock.readLock()`.

ReentrantReadWriteLock.WriteLock - Write lock для writer'ов, одержуваний через `readWriteLock.writeLock()`.

LockSupport - Призначений для побудови класів з об’єктами Lock. Містить методи для парковки потоків замість застарілих методів `Thread.suspend()` і `Thread.resume()`.

Блокуванням з повторним входом (*reentrant*) - це блокування з підрахунком кількості отриманих блокувань потоком (якщо потік, який утримує блокування, знову його отримує, кількість блокувань збільшується, і тоді для реального розблокування потрібно два рази зняти блокування). Це аналогічно семантиці `synchronized`: якщо потік входить в синхронний блок, захищений монітором, який вже належить потоку, потоку буде дозволено подальше функціонування, і блокування не буде знято, коли потік вийде з другого (або наступного) блоку `synchronized`, воно буде зняте тільки коли він вийде з першого блоку `synchronized`, в який він увійшов під захистом монітора.

Методи інтерфейсу Lock, реалізовані в класі ReentrantLock:

`void lock()` - запитує і отримує блокування критичної секції коду (початок якої відзначено даним методом) потоком, в якому з об'єкта ReentrantLock викликається метод `lock()`. Якщо блокування не доступне, то поточний потік буде призупинений до тих пір, поки блокування буде звільнене іншими потоками;

`void lockInterruptibly()` - запитує і отримує блокування критичної секції коду потоком, в якому з об'єкта ReentrantLock викликається даний метод, поки поточний потік не буде перерваний перериванням або блокування не буде звільнене. Якщо блокування не доступне, то поточний потік буде призупинений до тих пір, поки блокування буде звільнене іншими потоками або поки якийсь інший потік не викличе метод `interrupt()` з поточного потоку;

`boolean tryLock()` - метод запитує і отримує блокування так само, як і `lock()`, але якщо блокування недоступне, потік, в якому з об'єкта ReentrantLock викликається даний метод, в стан сну не переводиться, а продовжує працювати (метод повертає `true` при отриманні блокування і `false` якщо блокування недоступне);

`boolean tryLock(long time, TimeUnit unit)` - аналогічний `tryLock()`, але в разі недоступності блокування потік, в якому з об'єкта ReentrantLock викликається даний метод, засинає на зазначений як параметр час (протягом цього часу він зможе захопити блокування, яке звільниться).

`void unlock()` - єдиний метод звільнення блокування критичної секції потоком, в якому з об'єкта ReentrantLock викликається даний метод.

`Condition newCondition()` - повертає об’єкт-реалізацію інтерфейса Condition, пов’язаний з об’єктом ReentrantLock. Перед викликом методів об’єкта-реалізації інтерфейсу Condition, пов’язаний об’єкт ReentrantLock повинен бути отриманий поточним потоком. Виклик метода `void await()` об’єкта-реалізації інтерфейсу Condition заставить поточний потік очікувати до виклику в ньому методів `void signal()` (`void signalAll()`) об’єкта-реалізації інтерфейсу Condition (або до переривання його методом `interrupt()`).

Розглянемо синхронізацію за допомогою об’єкту ReentrantLock: Для початку нагадаємо, як виконується синхронізація за допомогою `synchronized` на прикладі класу з реалізацією завдання 4-разового інкременту лічильника з використанням синхронізованого з допомогою `synchronized` методу.

```
public class MyCounter {
    private int x = 1;
    public synchronized void increment() {
        try {
            for (int i = 0; i < 4; i++) {
            System.out.printf("%s %d \n",
                        Thread.currentThread().getName(), x);
                x++;
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Клас потоку з реалізацією завдання:

```
public class MyCounterThread extends Thread {
    MyCounter res;
    MyCounterThread(MyCounter res) {
        this.res = res;
    }
    @Override
    public void run() {
        res.increment();
    }
}
```

Тоді у якомусь класі виконаємо запуск 5 потоків, що реалізують завдання інкременту лічильника:

```
public static void main(String[] args) {
    MyCounter commonResource = new MyCounter();
```

```
    for (int i = 0; i < 5; i++) {
        Thread t = new MyCounterThread(commonResource);
        t.setName("Потік " + i);
        t.start();
    }
}
```

В консоль буде виведено:

Потік 0 1	Потік 4 5	Потік 3 9	Потік 2 13	Потік 1 17
Потік 0 2	Потік 4 6	Потік 3 10	Потік 2 14	Потік 1 18
Потік 0 3	Потік 4 7	Потік 3 11	Потік 2 15	Потік 1 19
Потік 0 4	Потік 4 8	Потік 3 12	Потік 2 16	Потік 1 20

Бачимо, що потоки отримують блокування (доступ до критичної секції) по черзі. Тепер виконаємо синхронізацію за допомогою об'єкту ReentrantLock. Клас завдання тепер буде виглядати так:

```
public class MyCounter {
    private int x = 1;
    ReentrantLock locker;
    public MyCounter(ReentrantLock locker) {
        this.locker = locker;
    }
    public void increment() {
        try {
          locker.lock();    //Отримання блокування
            for (int i = 0; i < 4; i++) {
                System.out.printf("%s %d \n",
                        Thread.currentThread().getName(), x);
                x++;
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
        } finally {
          locker.unlock();  //вивільнення блокування
        }
     }
}
```

Об'єкт ReentrantLock є атрибутом класу завдання і задається як параметр конструктора. На початку методу `increment()` з нього викликається метод `lock()`, відзначаючи початок критичної секції. При виклику `lock()` поточним потоком, потік отримує блокування для виконання критичної секції, що починається в місці виклику `lock()`, і інші потоки не зможуть увійти до критичної секції (отримати блокування). Якщо ж інший потік заблокував критичну секцію, то метод `lock()` переведе поточний потік, в якому викликається `lock()` в стан сну, поки інший потік не звільнить блокування критичної секції. В блоці `finally` блокування вивільнюється викликом метода `unlock()` з об'єкта

ReentrantLock по завершенню метода `increment()` або виникненню виключення. Місце виклику метода `unlock()` вказує на завершення критичної секції коду.

Класи `MyCounterThread` та `Main` не змінилися. Запуск програми на виконання демонструє аналогічний синхронізації за допомогою `synchronized` результат:

Потік 0 1	Потік 2 5	Потік 3 9	Потік 1 13	Потік 4 17
Потік 0 2	Потік 2 6	Потік 3 10	Потік 1 14	Потік 4 18
Потік 0 3	Потік 2 7	Потік 3 11	Потік 1 15	Потік 4 19
Потік 0 4	Потік 2 8	Потік 3 12	Потік 1 16	Потік 4 20

Але як і при використанні `synchronized`, використання методу `lock()` об'єкту ReentrantLock призводить до переведення потоків, які очікують блокування у неробочий стан. Це можна змінити використанням методу `tryLock()` об'єкту ReentrantLock:

```
public class MyCounter {
    private int x = 1;
    ReentrantLock locker;
    public MyCounter(ReentrantLock locker) {
        this.locker = locker;
    }
    public void increment() {
        try {
            boolean flag = locker.tryLock(3000,
                                          TimeUnit.MILLISECONDS);
            if(flag) {
                try {
                    for (int i = 0; i < 4; i++) {
                        System.out.printf("%s %d \n",
                        Thread.currentThread().getName(), x);
                        x++;
                        Thread.sleep(500);
                    }
                }finally {
                    locker.unlock();
                }
            }
            else{
                System.out.println(Thread.currentThread()
         .getName() +": Can't get lock. Contimue to work.");
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Тепер блокування отримується викликом з об’єкту ReentrantLock методу `tryLock(3000, TimeUnit.MILLISECONDS)`, в потоці, який намагається отримати блокування критичної секції. Якщо блокування можливе, потік його отримує в метод повертає `true`, якщо критична секція виконується іншим потоком, поточний потік переводиться у стан очікування доки не відбудеться одна з трьох подій:

- не звільниться та не буде отримане поточним потоком блокування критичної секції;
- інший потік не викличе метод `interrupt()` з поточного потоку;
- завершиться проміжок часу, вказаний у методі `tryLock` як параметр.

Враховуючи, що у кожній ітерації завдання потік "спить" 500 мс, менший 10 секунд проміжок часу повинен приводити к ситуації, коли потоки, що очікували впродовж 3 секунд, перейдуть у робочий стан і будуть виконувати іншу роботу, вказану у блоці `else` коду. Запуск програми виведе:

Потік 1 1
Потік 1 2
Потік 1 3
Потік 1 4
Потік 2 5
Потік 2 6
Потік 0: Can't get lock. Contimue to work.
Потік 3: Can't get lock. Contimue to work.
Потік 4: Can't get lock. Contimue to work.
Потік 2 7
Потік 2 8

Розглянемо використання об’єкту Condition разом з об’єктом ReentrantLock. Для початку згадаємо, як організовувати координацію потоків за допомогою механізму `wait/notify/notifyAll`. Нехай є класи-завдання постачальника товарів на склад Producer, клас покупця товарів зі складу Consumer та клас Store, який описує склад, що зберігає товари з методами отримання та видачі товарів зі складу (забезпечимо синхронізацію доступу цими методами до товарів за допомогою модифікатора `synchronized`):

```
public class Producer implements Runnable {

    Store store;

    public Producer(Store store) {
        this.store = store;
    }

    @Override
    public void run() {
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}

public class Consumer implements Runnable {

    Store store;

    public Consumer(Store store) {
        this.store = store;
```

```
    }
    @Override
    public void run() {
        for (int i = 1; i < 6; i++) {
            store.get();
        }
    }
}
public class Store {
    private int product = 0;
    public synchronized void get() {
        try {
           //поки немає доступних товарів на складі, очікуємо
            while (product < 1) {
                this.wait();
            }
            product--;
            System.out.println("Consumer bought 1 product");
            System.out.println("Goods in stock: " + product);
            //сигналізуємо про можливість блокування
            this.notifyAll();
        } catch (InterruptedException ex) {
            System.out.println(ex.getMessage());
        }
    }

    public synchronized void put() {
        try {
          //поки на складі 3 товари, чекаємо звільнення місця
            while (product >= 3) {
                this.wait();
            }
            product++;
            System.out.println("Producer added 1 product");
            System.out.println("Goods in stock: " + product);
            //сигналізуємо про можливість блокування
            this.notifyAll();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Доки на складі товари відсутні, потік з завданням споживача буде очікувати внаслідок виклику метода `wait()` з поточного об’єкту Store. Якщо ж товари наявні споживач виконає декремент їх кількості та сповістить інші потоки про

можливість отримання блокування викликом методу `notifyAll()`. Аналогічна ситуація і в методі `put()`, який використовує продюсер, тільки очікування виникає, якщо кількість товарів на складі перевищує 3. Запуск програми демонструє роботу складу:

```
public static void main(String[] args) {
    Store store = new Store();
    Producer producer = new Producer(store);
    Consumer consumer = new Consumer(store);
    new Thread(producer).start();
    new Thread(consumer).start();
}
```

і виводить у консоль:

```
Producer added 1 product
Goods in stock: 1
Producer added 1 product
Goods in stock: 2
Producer added 1 product
Goods in stock: 3
Consumer bought 1 product
Goods in stock: 2
Consumer bought 1 product
Goods in stock: 1
Consumer bought 1 product
Goods in stock: 0
Producer added 1 product
Goods in stock: 1
Producer added 1 product
Goods in stock: 2
Consumer bought 1 product
Goods in stock: 1
Consumer bought 1 product
Goods in stock: 0
```

Те ж саме можна зробити з використанням об'єкту ReentrantLock та пов'язаного з ним об'єкту Condition. У цьому випадку клас Store

```
public class Store {

    private int product = 0;
    ReentrantLock locker;
    Condition condition;

    public Store() {
        locker = new ReentrantLock();
        condition = locker.newCondition();
    }
    public void get() {
        try {
            locker.lock();
           //поки немає доступних товарів на складі, очікуємо
            while (product < 1) {
                condition.await();
            }
            product--;
            System.out.println("Consumer bought 1 product ");
            System.out.println("Goods in stock: " + product);

            //сигналізуємо про можливість блокування
```

```
            condition.signalAll();
        } catch (InterruptedException ex) {
            System.out.println(ex.getMessage());
        } finally {
            locker.unlock();
        }
    }

    public void put() {
        try {
            locker.lock();
          //поки на складі 3 товари, чекаємо звільнення місця
            while (product >= 3) {
                condition.await();
            }
            product++;
            System.out.println("Producer added 1 product ");
            System.out.println("Goods in stock: " + product);
            //сигналізуємо про можливість блокування
            condition.signalAll();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            locker.unlock();
        }
    }
}
```

Запуск програми дасть аналогічний попередній програмі результат.

На останок наведемо приклад програми яка демонструє повторне отримання блокування потоком на тому ж об’єкті ReentrantLock. Нехай є клас з вкладеними методами які інкрементують загальну змінну - зовнішній `outerMethod()` два рази, а внутрішній `innerMethod()` – один:

```
public class NestedMethodsTask {
    int x = 1;
    ReentrantLock locker;

    public NestedMethodsTask(ReentrantLock locker) {
        this.locker = locker;
    }

    public void outerMethod() {
        try {
          locker.lock();
          System.out.println(Thread.currentThread().getName()
            + ": Lock acquired and " + locker.getHoldCount()
            + " lock hold in outerMethod().");
            for (int i = 0; i < 2; i++) {
```

```
                System.out.printf("%s %d \n",
                        Thread.currentThread().getName(), x);
                x++;
                Thread.sleep(500);
            }
            innerMethod();
        } catch (InterruptedException e) {
        } finally {
          locker.unlock();
          System.out.println(Thread.currentThread().getName()
             + ": Lock released and " + locker.getHoldCount()
             + " lock hold in outerMethod().");
        }
    }

    public void innerMethod() {
        try {
          locker.lock();
          System.out.println(Thread.currentThread().getName()
             + ": Lock acquired and " + locker.getHoldCount()
             + " lock hold in innerMethod().");
            System.out.printf("%s %d \n",
                        Thread.currentThread().getName(), x);
            x++;
            Thread.sleep(500);
        } catch (InterruptedException ex) {
        } finally {
            locker.unlock();
          System.out.println(Thread.currentThread().getName()
             + ": Lock released and " + locker.getHoldCount()
             + " lock hold in innerMethod().");
        }
    }
}
```

Додані повідомлення з інформацією про кількість блокувань, отриманих потоком (для цього використовується метод `getHoldCount()` класу ReentrantLock). Клас потоку, що використовує описане вище завдання виглядає так:

```
public class MyThread extends Thread {

    NestedMethodsTask res;
    MyThread(NestedMethodsTask res) {
        this.res = res;
    }

    @Override
    public void run() {
        try {
            sleep((long) (Math.random() * 1000));
```

```
            } catch (InterruptedException ex) {
            }
            res.outerMethod();
        }
}
```

Тоді у деякому класі створюються два потоки, що виконують завдання:

```
public static void main(String[] args) {
    ReentrantLock locker = new ReentrantLock();
    NestedMethodsTask commonResource =
                                new NestedMethodsTask(locker);
    for (int i = 0; i < 2; i++) {
        Thread t = new MyThread(commonResource);
        t.setName("Потік " + i);
        t.start();
    }
}
```

Запуск програми виводить таку інформацію:

```
Потік 0: Lock acquired and 1 lock hold in outerMethod().
Потік 0 1
Потік 0 2
Потік 0: Lock acquired and 2 lock hold in innerMethod().
Потік 0 3
Потік 0: Lock released and 1 lock hold in innerMethod().
Потік 0: Lock released and 0 lock hold in outerMethod().
Потік 1: Lock acquired and 1 lock hold in outerMethod().
Потік 1 4
Потік 1 5
Потік 1: Lock acquired and 2 lock hold in innerMethod().
Потік 1 6
Потік 1: Lock released and 1 lock hold in innerMethod().
Потік 1: Lock released and 0 lock hold in outerMethod().
```

Ми бачимо, що при виконанні вкладеного методу збільшується число блокувань, отриманих потоком, а при його завершенні - зменшується.

Використовуйте об’єкти ReentrantLock, коли нам потрібно щось, чого не забезпечує `synchronized`, наприклад, вивільнення для іншої роботи потоків, які не отримують блокування, очікування блокування у проміжок певного часу, організацію блокування, яке може бути перерване іншим потоком, використання кількох змінних умов блокування або організацію опитування блокування.

Крім об’єктів-реалізацій інтерфейсу Lock Java Concurrent API пропонує *об'єкти-синхронізатори* (*Synchronizers*) з високорівневими (відносно використанню моніторів) механізмами забезпечення синхронізації роботи потоків такі, як:

Семафор (Semaphore) - об'єкт, який здійснює доступ потоків до ресурсу за допомогою лічильника дозволів на доступ;

З*асувка зі зворотним відліком* (CountDownLatch) - об'єкт, що затримує всі потоки до тих пір, поки не буде виконано певну умову, при виконанні умови він звільняє всі потоки одночасно;

Циклічний бар'єр (CyclicBarrier) - об'єкт, що дозволяє синхронізувати потоки в загальній точці, в якій вказана кількість паралельних потоків зустрічається і блокується. Як тільки все потоки прибудуть до точки, виконується опціонна операція (або не виконується, якщо бар'єр був ініційований без неї), і, після того, як операція виконана, бар'єр ламається і потоки, що очікують, «звільняються»;

Обмінник (Exchanger<V>) - об'єкт, призначений для обміну даними між потоками в певній точці роботи обох потоків. Він є типізованим і типізуються типом даних, якими потоки обмінюються.

Фазер (Phaser) - дозволяє синхронізувати потоки, що представляють окрему фазу або стадію виконання загального дії. *Phaser* визначає об'єкт синхронізації, який чекає, поки не завершиться певна фаза. Потім *Phaser* переходить до наступної стадії або фазі і знову чекає її завершення.

На відміну від об’єктів Lock, які надають потоку, в якому викликаний метод `lock()` цього об’єкту (або його варіації), ексклюзивний доступ до критичної секції, об'єкт класу java.util.concurrent.Semaphore підтримує певну кількість дозволів на доступ до критичної секції, яка вказується як параметр конструктора при створенні семафора.

```
Semaphore semaphore = new Semaphore(5);
```

Створювані конструктором семафори не забезпечують справедливість при наданні доступу потокам до критичної секції. *Справедливе блокування* (*FairSync*) - це коли потоки отримують блокування в тому порядку, в якому вони його запитували, *несправедливе блокування* (*NonfairSync*) може допускати *"безлад"* (*barging*), коли потік може отримати блокування раніше іншого, який запитував її першим. При передачі в конструктор другого параметра `true` - буде забезпечуватися справедливе блокування (семафор буде використовувати FIFO чергу).

```
Semaphore sem = new Semaphore(5, true);
```

Кожен виклик методу `void acquire()` з семафора в потоці зменшує на 1 кількість дозволів на доступ до критичної секції, кожен виклик методу `void release()` - збільшує число дозволів на 1. Таким чином, кількість потоків, що одночасно виконують критичну секцію коду, обмежується кількістю дозволів семафора, а критична секція починається викликом `acquire()` і закінчується викликом `release()` (Рис. 6). Також можна використовувати методи семафору `void acquire(int permits)` та `void release(int permits)`, які зменшують та збільшують одночасно `permits` кількість дозволів. Окрім того, наявна група методів `boolean tryAcquire()`, `boolean tryAcquire(long timeout, TimeUnit unit)`, `boolean tryAcquire(int permits)` та `tryAcquire(int permits, long timeout, TimeUnit unit)`, які отримують дозвіл (або `permits` дозволів) з семафору тільки, якщо цей дозвіл наявний в момент виклику методу (або впродовж проміжку часу, вказаного як параметр). Вони повертають `true`, якщо дозвіл отримано, та `false` - в іншому випадку. Такі

методи дозволяють уникати (або зменшувати час) очікування потоків, дозволяючи їм виконувати іншу роботу.

Семафор з кількістю дозволів, що дорівнює 1, називається *бінарним семафором* або *взаємовиключаючим семафором* (*Mutex*-ом - *Mutually Exclusive Semaphore*).

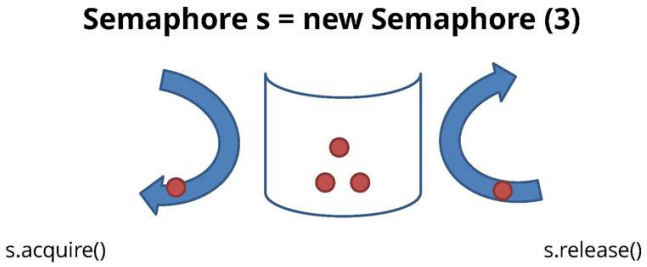

Рис. 6. Створення та робота семафору

Наведемо приклад використання семафору. Нехай є завдання, доступ до якого обмежується для граничної кількості потоків (вказаної в *RunTime*) за допомогою семафора.

```
public class Task implements Runnable {
    Semaphore semaphore;
    public Task(Semaphore semaphore) {
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        boolean permit = false;
        try {
            permit = semaphore.tryAcquire(3000,
                                          TimeUnit.MILLISECONDS);
            if (permit) {
                System.out.println(Thread.currentThread().
                                   getName() + ": Permit acquired");
                sleep(5000);
            } else {
              System.out.println(Thread.currentThread().
                    getName() + ": Could not acquire permit");
            }
        } catch (InterruptedException ex) {
        } finally {
            if (permit) {
                semaphore.release();
                System.out.println(Thread.currentThread().
                                   getName() + ": Permit released");
            }
        }
    }
```

```
}
```

Тоді у деякому класі:

```
public static void main(String[] args)
                                    throws InterruptedException {
    ExecutorService executor =
                              Executors.newFixedThreadPool(10);
    Semaphore semaphore = new Semaphore(3);
    Task task = new Task(semaphore);
    for (int i = 0; i < 10; i++) {
        executor.submit(task);
        /*Потрібно підбирати час, щоб побачити
          роботу семафору*/
        Thread.sleep(500);
    }
    executor.shutdown();
}
```

Запуск програми демонструє роботу семафору:

```
pool-1-thread-1: Permit acquired
pool-1-thread-2: Permit acquired
pool-1-thread-3: Permit acquired
pool-1-thread-4: Could not acquire permit
pool-1-thread-1: Permit released
pool-1-thread-5: Permit acquired
pool-1-thread-2: Permit released
pool-1-thread-6: Permit acquired
pool-1-thread-3: Permit released
pool-1-thread-7: Permit acquired
pool-1-thread-8: Could not acquire permit
pool-1-thread-9: Could not acquire permit
pool-1-thread-10: Could not acquire permit
pool-1-thread-5: Permit released
pool-1-thread-6: Permit released
pool-1-thread-7: Permit released
```

Бачимо, що після отримання трьох дозволів потоками 1, 2 та 3, потік 4 не дочекався вивільнення дозволу, а потоки 5, 6 та 7 - дочекались і отримали його. Інші потоки (8, 9 та 10) не дочекались дозволу.

Якщо Semaphore дозволяє потокам "заходити до критичної секції по одному", то *засувка зі зворотним відліком* (CountDownLatch) нагадує стартовий бар'єр на скачках. Цей клас затримує всі потоки до тих пір, поки не буде виконана певна умова. При виконанні умови він звільняє всі потоки одночасно.

Об’єкт CountDownLatch створюється конструктором з параметром, що ініціалізує лічильник засувки:

```
CountDownLatch counter = new CountDownLatch(5);
```

Метод `void await()` класу CountDownLatch переводить потік, у якому він викликаний, у стан очікування доки лічильник засувки не зменшиться до нуля

або потік не буде перерваний викликом з нього методу `interrupt()`. Метод `void countDown()` - зменшує лічильник засувки на 1, звільняючи разом всі потоки, що очікують, коли лічильник досягне 0.

Наведемо приклад використання об'єкту CountDownLatch. Нехай є клас завдання, в методі `run()` якого засувка переводить потік в стан очікування доки її лічильник не стане рівним 0:

```
public class Runner extends Thread {
    private CountDownLatch timer;
    public Runner(CountDownLatch timer, String name) {
        this.timer = timer;
        setName(name);
        System.out.println(getName()
                           + " ready and waiting to start");
    }
    @Override
    public void run() {
        try {
            timer.await();
        } catch (InterruptedException ex) {
            System.err.println("interrupted - can't start
running the race");
        }
        System.out.println(getName() + " started running");
    }
}
```

Тоді у деякому класі організуємо "забіг" трьох потоків:

```
public static void main(String[] args) {
    CountDownLatch counter = new CountDownLatch(5);
    new Runner(counter, "Carl").start();
    new Runner(counter, "Joe").start();
    new Runner(counter, "Jack").start();
    System.out.println("Starting the countdown ");
    /*Початкове значення лічильника*/
    long countVal = counter.getCount();
    while (countVal > 0) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
        }
        /*Поточне значення лічильника */
        System.out.println(countVal);
        if (countVal == 1) {
            System.out.println("Start");
        }
        counter.countDown();
```

```
            countVal = counter.getCount();
        }
    }
```

Запуск програми виводить імітацію "забігу" трьох учасників:

```
Carl ready and waiting to start
Joe ready and waiting to start
Jack ready and waiting to start
Starting the countdown
5
4
3
2
1
Start
Carl started running
Jack started running
Joe started running
```

Об’єкти CountDownLatch можуть бути використані в самих різних схемах синхронізації: наприклад, поки один потік виконує роботу, змусити інші потоки чекати або, навпаки, щоб змусити потік чекати доки інші потоки не виконають свою роботу.

Об’єкти *циклічного бар'єр*у (CyclicBarrier) є точкою синхронізації, в якій вказана кількість паралельних потоків зустрічається і блокується. Як тільки всі потоки прибули, виконується опціонна операція (або не виконується, якщо бар'єр був ініціалізований без неї), і, після виконання операції, бар'єр ламається і потоки, що очікують, «звільняються». В конструктор бар'єру (`CyclicBarrier(int parties)` і `CyclicBarrier(int parties, Runnable barrierAction)`) передається кількість сторін, які повинні «зустрітися», і, опціонально, дія, яка має відбутися, коли сторони зустрілися, але перед тим коли вони будуть «відпущені». Метод `int await()` об’єкту CyclicBarrier вказує потоку, з якого був викликаний цей метод, що він «підійшов» до бар’єра. Цей потік переводиться в стан очікування до моменту, коли всі інші потоки, кількість яких вказана у конструкторі як параметр `parties` досягнуть бар’єру (в них буде викликаний метод `int await()` об’єкту CyclicBarrier). Існує метод `int await(long timeout, TimeUnit unit)` об’єкту CyclicBarrier, який переводить поточний потік в стан очікування доки всі інші потоки не досягнуть бар’єру або не спливе проміжок часу, вказаний як параметр.

Розглянемо роботу об’єкта CyclicBarrier у програмі, яка емулює паромну переправу. Паром може переправляти одночасно по три автомобіля. Щоб не ганяти паром зайвий раз, потрібно відправляти його, коли біля переправи збереться мінімум три автомобілі. Є клас-завдання Car, що описує автомобіль та клас-завдання FerryBoat, що описує паром:

```
public class Car implements Runnable {
    private int carNumber;
```

```
    public Car(int carNumber) {
        this.carNumber = carNumber;
    }
    @Override
    public void run() {
        try {
            System.out.printf("Car №%d drove up
                                to the ferry.\n", carNumber);
            Ferry.BARRIER.await();
            System.out.printf("Car №%d continued to move.\n",
                                carNumber);
        } catch (Exception e) {
        }
    }
}

public class FerryBoat implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(500);
            System.out.println("FerryBoat ferrying cars!");
        } catch (InterruptedException e) {
        }
    }
}
```

Клас Car містить виклик методу `await()` з об'єкту CyclicBarrier, який створюється в *RunTime*-класі Ferry. В його конструкторі параметрами вказуються 3 потоки (автомобілі), які будуть очікувати зняття бар'єру, та дія - створення об'єкту FerryBoat (парому), який буде перевозити автомобілі:

```
public class Ferry {
    static final CyclicBarrier BARRIER =
                        new CyclicBarrier(3, new FerryBoat());
    public static void main(String[] args)
                                throws InterruptedException {
        for (int i = 0; i < 9; i++) {
            new Thread(new Car(i)).start();
            Thread.sleep(400);
        }
    }
}
```

Запуск програми виводить таку інформацію:

```
Car №0 drove up to the ferry.
Car №1 drove up to the ferry.
Car №2 drove up to the ferry.
```

```
Car №3 drove up to the ferry.
FerryBoat ferrying cars!
Car №2 continued to move.
Car №1 continued to move.
Car №0 continued to move.
Car №4 drove up to the ferry.
Car №5 drove up to the ferry.
Car №6 drove up to the ferry.
FerryBoat ferrying cars!
Car №3 continued to move.
Car №5 continued to move.
Car №4 continued to move.
Car №7 drove up to the ferry.
Car №8 drove up to the ferry.
FerryBoat ferrying cars!
Car №8 continued to move.
Car №7 continued to move.
Car №6 continued to move.
```

Бачимо що паром перевозить по 3 автомобілі, а інші чекають. CyclicBarriers корисні в програмах, що включають фіксовану групу потоків, які час від часу повинні чекати один одного. Бар'єр називається циклічним, оскільки його можна використовувати повторно після випуску потоків, що очікували.

Об’єкти *обмінника* (Exchanger<V>) можуть обмінюватись даними між двома потоками в певній точці роботи обох потоків. Обмінник - узагальнений клас, він параметризований типом об'єкта для обміну. Обмінник є точкою синхронізації пари потоків: потік, що викликає у обмінника метод `V exchange(V x)` блокується і чекає інший потік. Коли інший потік викличе той же метод, відбудеться обмін об'єктами: кожен з них отримає аргумент методу `V exchange(V x)`, викликаного в іншому потоці. Варто відзначити, що обмінник підтримує передачу `null` значення. Це дає можливість використовувати його для передачі об'єкта в одну сторону, або, просто як точку синхронізації двох потоків.

Розглянемо наступний приклад. Є дві вантажівки: одна їде з пункту A в пункт D, друга - з пункту B в пункт C. Дороги AD і BC перетинаються в пункті E. З пунктів A і B потрібно доставити посилки в пункти C і D. Для цього вантажівки в пункті E повинні зустрітися і обмінятися відповідними посилками. Є клас-завдання Truck, що описує вантажівку:

```
public class Truck implements Runnable {

    private int number;
    private String dep;
    private String dest;
    private String[] parcels;

    public Truck(int number, String departure,
                      String destination, String[] parcels) {
        this.number = number;
        this.dep = departure;
```

```
        this.dest = destination;
        this.parcels = parcels;
    }
    @Override
    public void run() {
        try {
            System.out.printf("The truck №%d was loaded with:
                %s и %s.\n", number, parcels[0], parcels[1]);
            System.out.printf("The truck №%d went from point
                      %s to point %s.\n", number, dep, dest);
            Thread.sleep(1000 + (long) Math.random() * 5000);
            System.out.printf("The truck №%d arrived at point
                                              E.\n", number);
            parcels[1] =
                     Delivery.EXCHANGER.exchange(parcels[1]);
            System.out.printf("The parcel for point %s was
                       moved to truck №%d.\n", dest, number);
            Thread.sleep(1000 + (long) Math.random() * 5000);
            System.out.printf("Truck №%d arrived at point %s
            and delivered the parcels: %s and %s.\n", number,
                                dest, parcels[0], parcels[1]);
        } catch (InterruptedException e) {
        }
    }
}
```

та клас програми Delivery:

```
public class Delivery {
    static final Exchanger<String> EXCHANGER =
                                              new Exchanger<>();
    public static void main(String[] args)
                                throws InterruptedException {
        String[] p1 = new String[]{"{parcel A->D}",
                           "{parcel A->C}"};  //for 1-st truck
        String[] p2 = new String[]{"{parcel B->C}",
                           "{parcel B->D}"};  //for 2-nd truck
        new Thread(new Truck(1, "A", "D", p1)).start();
        Thread.sleep(100);
        new Thread(new Truck(2, "B", "C", p2)).start();
    }
}
```

Запуск програми виводить таку інформацію:

```
The truck №1 was loaded with: {parcel A->D} и {parcel A->C}.
The truck №1 went from point A to point D.
The truck №2 was loaded with: {parcel B->C} и {parcel B->D}.
The truck №2 went from point B to point C.
The truck №1 arrived at point E.
```

The truck №2 arrived at point E.
The parcel for point D was moved to truck №1.
The parcel for point C was moved to truck №2.
Truck №1 arrived at point D and delivered the parcels: {parcel A->D} and {parcel B->D}.
Truck №2 arrived at point C and delivered the parcels: {parcel B->C} and {parcel A->C}.

Як ми бачимо, коли перша вантажівка (перший потік) приїжджає в пункт E (досягає точки синхронізації), він чекає поки друга вантажівка (другий потік) приїде в пункт E (досягне точки синхронізації). Після цього відбувається обмін посилками (типу String) і обидві вантажівки (потоки) продовжують свій шлях (роботу).

Клас Phaser дозволяє синхронізувати потоки, що представляють окрему фазу або стадію виконання загального дії. Phaser визначає об'єкт синхронізації, який чекає, поки не завершиться певна фаза. Потім Phaser переходить до наступної фази і знову чекає її завершення. Для створення об'єкта Phaser використовується один з конструкторів:

```
Phaser()
Phaser(int parties)
Phaser(Phaser parent)
Phaser(Phaser parent, int parties)
```

Параметр `parties` вказує на кількість сторін (потоків), які повинні виконувати всі фази дії. Перший конструктор створює об'єкт Phaser без будь-яких сторін. Другий конструктор реєструє передану у конструктор кількість сторін. Третій і четвертий конструктори додатково встановлюють батьківський об'єкт Phaser. Основні методи класу Phaser:

`int register()` - реєструє нову сторону, яка виконує фази, і повертає номер поточної фази (зазвичай фаза 0);

`int arrive()` - повідомляє, що сторона завершила фазу і повертає номер поточної фази, при виклику даного методу потік не припиняється, а продовжує виконуватися;

`int arriveAndAwaitAdvance()` - аналогічний методу `arrive()`, тільки при цьому змушує об’єкт Phaser очікувати завершення фази усіма іншими сторонами;

`int awaitAdvance(int phase)` - якщо `phase` дорівнює номеру поточної фази, призупиняє потік, у якому викликаний цей метод, до закінчення поточної фази. Повертає номер наступної фази, або аргумент, якщо він негативний, або (негативна) поточна фаза, якщо вона завершена;

`int arriveAndDeregister()` - повідомляє про завершення всіх фаз стороною і знімає її з реєстрації. Повертає номер поточної фази або негативне число, якщо синхронізатор Phaser завершив свою роботу;

`int getPhase()` - повертає номер поточної фази.

При роботі з класом Phaser зазвичай спочатку створюється його об'єкт. Далі треба зареєструвати всі сторони (потоки), які беруть участь у виконанні фаз. Для

реєстрації у кожній зі сторін викликається метод `register()` (можна обійтися і без цього методу, передавши потрібну кількість сторін в конструктор Phaser).

Потім кожна сторона (потік) виконує певний набір дій, які становлять фазу. А синхронізатор Phaser чекає, поки всі сторони (потоки) не завершать виконання фази. Щоб повідомити синхронізатор про те, що фаза завершена, сторона (потік) повинна викликати метод `arrive()` або `arriveAndAwaitAdvance()`. Після цього синхронізатор переходить до виконання наступної фази.

Розглянемо наступний приклад. Є п'ять зупинок. На перших чотирьох з них можуть стояти пасажири і чекати автобуса. Автобус виїжджає з парку і зупиняється на кожній зупинці на деякий час. Після кінцевої зупинки автобус їде в парк. Нам потрібно забрати пасажирів і висадити їх на потрібних зупинках. Є клас-потік Passenger, що описує пасажира - сторону фаз:

```
public class Passenger extends Thread {

    int departure;
    int destination;

    public Passenger(int departure, int destination) {
       this.departure = departure;
       this.destination = destination;
       System.out.println(this + " waiting at the bus stop №"
                          + this.departure);
    }

    @Override
    public void run() {
        try {
            System.out.println(this + " got on the bus.");
            while (Bus.PHASER.getPhase() < destination) {
                /*Поточний потік завершив фазу, очікуємо
                  завершення фази іншими потоками*/
                Bus.PHASER.arriveAndAwaitAdvance();
            }
            Thread.sleep(1);
            System.out.println(this + " left the bus.");
            /*Всі фази поточний потік завершив*/
            Bus.PHASER.arriveAndDeregister();
        } catch (InterruptedException e) {
        }
    }

    @Override
    public String toString() {
        return "Passenger{" + departure + " -> "
                                          + destination + '}';
    }
}
```

та клас програми Bus, у якому за допомогою створеного об’єкту Phaser виконується синхронізація сторін:

```
public class Bus {
    static final Phaser PHASER = new Phaser(1); //Реєстрація
  //потоку Main. Фази 0 та 6 - автобусний парк, 1 - 5 зупинки
    public static void main(String[] args)
                                throws InterruptedException {
        ArrayList<Passenger> passengers = new ArrayList<>();
        for (int i = 1; i < 5; i++) { //Генерація пасажирів
            if ((int) (Math.random() * 2) > 0) {
                passengers.add(new Passenger(i,
                 i + 1));     //виходить на наступній зупинці
            }
            if ((int) (Math.random() * 2) > 0) {
                passengers.add(new Passenger(i, 5));
                               //виходить на кінцевій зупинці
            }
        }
        for (int i = 0; i < 7; i++) {
            switch (i) {
              case 0:
                System.out.println("The bus left the park.");
                /*Потік Main (автобус) завершив фазу 0*/
                PHASER.arrive();
                break;
              case 6:
                System.out.println("The bus went to park.");
                /*Потік Main (автобус) завершив всі фази*/
                PHASER.arriveAndDeregister();
                break;
              default:
                int currentBusStop = PHASER.getPhase();
                System.out.println("Bus stop № "
                                          + currentBusStop);
                /*Перевіряємо, чи є пасажири на зупинці*/
                for (Passenger p : passengers) {
                    if (p.departure == currentBusStop) {
                        /*Якщо на зупинці є пасажири,
                        реєструємо новий потік*/
                        PHASER.register();
                        p.start();
                    }
                }
```

```
                    /*Поточний потік завершив фазу, очікуємо
                      її завершення іншими потоками*/
                    PHASER.arriveAndAwaitAdvance();
                } //кінець switch
            } //кінець for
        }
}
```

Запуск програми демонструє первезення пасажирів автобусом:

```
Passenger{2 -> 3} waiting at the bus stop № 2
Passenger{4 -> 5} waiting at the bus stop № 4
Passenger{4 -> 5} waiting at the bus stop № 4
The bus left the park.
Bus stop № 1
Bus stop № 2
Passenger{2 -> 3} got on the bus.
Bus stop № 3
Passenger{2 -> 3} left the bus.
Bus stop № 4
Passenger{4 -> 5} got on the bus.
Passenger{4 -> 5} got on the bus.
Bus stop № 5
Passenger{4 -> 5} left the bus.
Passenger{4 -> 5} left the bus.
The bus went to the park.
```

Атомарними (*Atomic*) вважаються операції, які можна виконувати в багатопотоковій програмі без застосування синхронізації і блокування. В основі таких операцій лежить алгоритм *порівняння з обміном* (*Compare and Swap - CAS*), ідея якого полягає в тому, що замість блокування монітора і взаємовиключення доступу декількох потоків до критичної секції коду (ресурсу) (*песимістичний підхід*), декільком потокам не забороняється отримувати одночасний доступ до ресурсу (відсутнє блокування) в надії, що взаємного їх впливу не буде (*оптимістичний підхід*). Якщо ж такий вплив виявляється, то останній потік дізнається про це і не зможе змінити ресурс, але буде намагатися це зробити пізніше. Неблокуючі (атомарні) операції мають менший негативний вплив на продуктивність (тому що не змушують потоки очікувати).

Алгоритм *порівняння з обміном* (*Compare and Swap - CAS*) порівнює вміст комірки пам'яті V з деяким значенням цієї комірки A, яке було прочитане потоком востаннє і, тільки якщо вони однакові, змінює вміст цієї комірки пам'яті на нове значення B. Це виконується в єдиній атомарної операції, атомарність гарантує, що нове значення розраховується на основі актуальної інформації. Якщо значення в пам'яті одночасно намагатиметься оновити інший потік, він зазнає невдачі, але потік повторно намагатиметься виконати оновлення. Результат операції вказує, чи виконується це заміщення змінної в пам'яті.

Розберемо всю операцію *CAS* покроково. Нехай спочатку V = 10 і є потоки 1 та 2, які хочуть прочитати і інкрементувати значення в комірці пам’яті V:

1) $V = 10$, $A_1 = 0$, $B_1 = 0$, $A_2 = 0$, $B_2 = 0$

2) Потоки 1 і 2 хочуть збільшити значення V, вони обидва читають значення:
$V = 10, A_1 = 10, B_1 = 0, A_2 = 10, B_2 = 0$

3) Потоки 1 і 2 збільшують прочитане значення на 1 у своїх локальних змінних (запам'ятовуючи також і попередні значення):
$V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

4) Нехай потік 1 отримує доступ до комірки пам'яті першим і порівнює значення V з останнім прочитаним значенням:
$V = 10, A_1 = 10, B_1 = 11, A_2 = 10, B_2 = 11$

```
if A1 = V
    V = B1
else
    operation failed
return V
```

V буде замінено як 11:
$V = 11, A_1 = 11, B_1 = 11, A_2 = 10, B_2 = 11$

5) Коли потік 2 отримає доступ до комірки пам'яті, то він виконує аналогічну операцію:

```
if A2 = V
    V = B2
else
    operation failed
return V
```

У цьому випадку $V = 11$ не дорівнює $A_2 = 10$, тому значення не замінюється і повертається поточне значення $V = 11$. Потік 2 оновляє ним останнє прочитане значення у A_2:
$V = 11, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 11$

5) Тепер потік 2, знову повторить операцію інкременту зі значеннями:
$V = 11, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 12$

6) Коли тепер потік 2 отримає доступ до комірки і ніякий інший потік не змінив за цей час її значення, то он виконавши CAS-алгоритм, замінить значення комірки на своє інкрементоване (т.к. $A_2 = 11$ дорівнювало $V = 11$). Нові значення будуть:
$V = 12, A_1 = 11, B_1 = 11, A_2 = 11, B_2 = 12$

Треба зауважити, що сучасні багатоядерні процесори мають підтримку операцій CAS на рівні мікропроцесорних команд (які виконуються дуже швидко, команда CMPXCHG для процесорів архітектури x86), таким чином, Java 5 надала доступ до функціоналу мікропроцесора, який забезпечує багатопотоковість.

Пакет java.util.concurrent.atomic містить класи які дозволяють атомарно змінювати примітивні змінні (boolean, int, long) та об'єкти, а також масиви та поля об'єктів примітивних змінних (int, long) та об'єктів (Рис. 7):

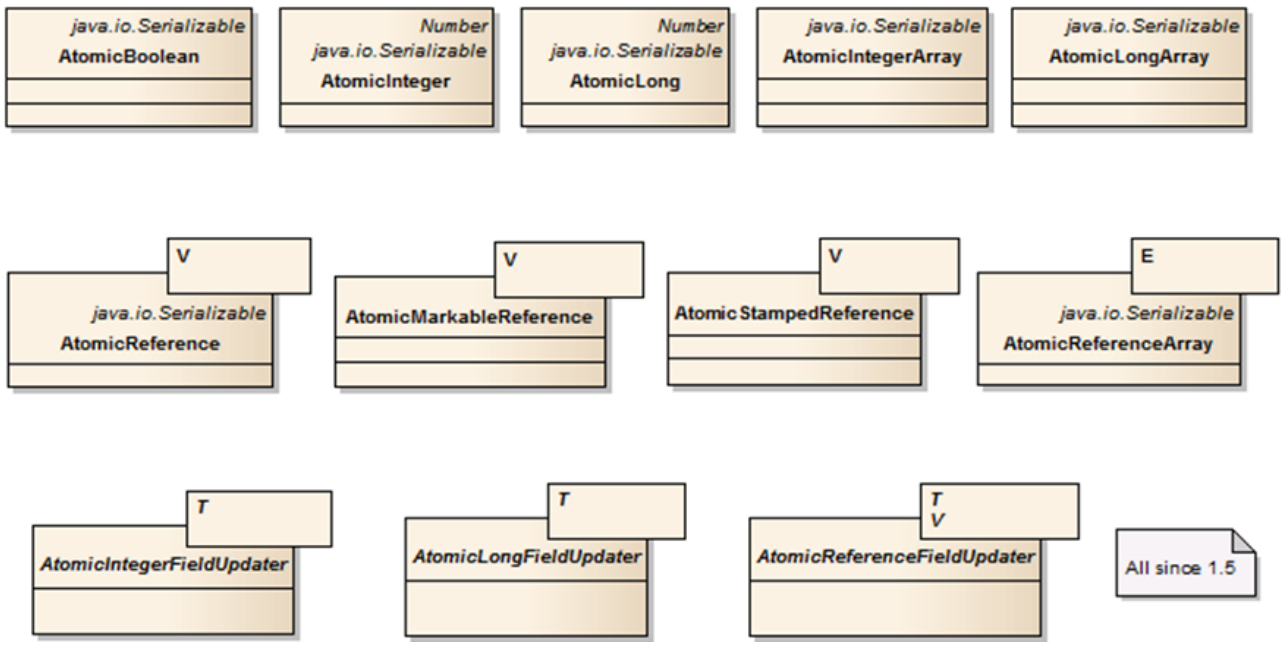

Рис. 7. Класи пакету java.util.concurrent.atomic

У класах AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference існує метод `boolean compareAndSet(type expect, type update)`, що приймає два аргументи відповідних типів: передбачуване поточне і нове значення. Метод атомарно встановлює об'єкту нове значення, якщо поточне значення дорівнює передбачуваному, і повертає `true`. Якщо поточне значення змінилося, то метод поверне `false` і нове значення не буде встановлене. Крім цього, класи мають метод `type getAndSet(type newValue)`, який атомарно безумовно встановлює нове значення і повертає старе.

Класи AtomicInteger і AtomicLong мають також методи інкремента/декремента:

`type getAndIncrement()` - атомарний інкремент поточного значення та повернення старого значення (еквівалентно операції `i++`);

`type incrementAndGet()` - атомарний інкремент поточного значення та повернення старого значення після збільшення (еквівалентно операції `++i`);

`type getAndDecrement()` - атомарний декремент поточного значення та повернення старого значення (еквівалентно операції `i--`);

`type decrementAndGet()` - атомарний декремент поточного значення та повернення старого значення після зменшення (еквівалентно операції `--i`);

та додавання нового значення:

`type addAndGet(type delta)` – атомарне додавання значення-аргумента до поточного, повертає нове значення після додавання;

`type getAndAdd(type delta)` – атомарне додавання значення-аргумента до поточного, повертає старе значення.

Також всі вказані класи мають методи отримання поточного значення `type get()` та безумовного встановлення заданого значення `void set(type newValue)`.

Класи AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray містять методи для роботи з елементами масивів, аналогічні методам класів AtomicInteger, AtomicLong, AtomicReference. Відміна цих методів - у доданні додаткового аргументу, що вказує на індекс елемента у масиві `i`, наприклад, метод `boolean compareAndSet(int i, type expect, type update)`, метод `type getAndIncrement(int i)` і т.і.

Класи AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater містять методи для оновлення значень полів об’єктів за їхніми іменами з використанням reflection, аналогічні методам класів AtomicInteger, AtomicLong, AtomicReference. Заміщення значень полів для CAS операцій визначається в конструкторі і кешується. Сильного падіння продуктивності через використання reflection не спостерігається. Відміна цих методів - у доданні додаткового аргументу, що вказує об’єкт `obj`, чиє поле оновлюється, наприклад, метод `boolean compareAndSet(T obj, type expect, type update)`, метод `type getAndIncrement(T obj)` і т.і.

Клас AtomicMarkableReference підтримує посилання на об'єкт разом із бітом позначки, який можна оновити атомарно.

Клас AtomicStampedReference підтримує посилання на об'єкт разом із цілим числом "штампом", яке може бути оновлене атомарно.

Розглянемо приклад коду, у якому порівняємо значення трьох лічильників, що інкрементуються декількома потоками: лічильника, значення якого зберігається у звичайній загальній змінній, лічильника, значення якого зберігається у волатильній загальній змінній, та лічильника, значення якого зберігається у загальному об’єкті атомарного класу і змінюється атомарними операціями. Ці лічильники - поля класу `MyCounter`:

```
public class MyCounter {
    public int cnt1;
    public volatile int cnt2;
    public AtomicInteger cnt3 = new AtomicInteger(0);
}
```

Клас потоку, об’єкти якого будуть інкрементувати лічильники:

```
public class MyCountThread extends Thread {
    MyCounter m;
    /*Кількість змін лічильників*/
    int n;

    public MyCountThread(MyCounter m, int n) {
        this.m = m;
        this.n = n;
    }


    @Override
    public void run() {
        for (int i = 0; i < n; i++) {
```

```
            m.cnt1++;
            m.cnt2++;
            m.cnt3.getAndIncrement();
        }
    }
}
```

Тоді, у деякому класі:

```
public static void main(String[] args) {
    int numOfThreads = 100;
    int countUpperBound = 1_000_000;
    MyCounter m = new MyCounter();
    MyCountThread[] tg = new MyCountThread[numOfThreads];
    for(int i = 0; i < numOfThreads; i++) {
        tg[i] = new MyCountThread(m, countUpperBound);
        tg[i].start();
    }
    for (MyCountThread t : tg) {
        try {
            t.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    System.out.printf("int: %s, volatile: %s, atomic: %s%n",
                                        m.cnt1, m.cnt2, m.cnt3);
}
```

Запуск програми виведе такі дані, наприклад:

```
int: 68292017, volatile: 60866787, atomic: 100000000
```

Бачимо, що і звичайна змінні, і змінна з модифікатором `volatile`, що містять інкрементоване значення лічильника, оказують менше значення, ніж кількість ітерацій (оскільки відсутнє блокування критичної секції коду і кілка потоків одночасно змінюють лічильники протягом однієї ітерації - див попередню лабораторну роботу). Звичайно, синхронізація блоку коду, що інкрементує лічильник, за допомогою `synchronized` виправить ситуацію. Але порівняння часу виконання операції інкременту лічильника 10000 разів 100 потоками дає 3-хкратний виграш у часі у разі використання атомарних змінних та операцій (зауважимо, що при менших кількостях перевага атомарних операцій зникає).

Пакет java.util.concurrent включає інтерфейси і класи, що розширюють Java Collection Framework і забезпечують додаткові можливості по багатопотоковості використання об'єктів цього фреймворка.

Нагадаємо, що клас java.util.Collections містить методи:

```
public static <T> Collection<T>
                    synchronizedCollection(Collection<T> c),
public static <T> List<T> synchronizedList(List<T> list),
public static <T> Set<T> synchronizedSet(Set<T> s),
```

```
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m),
public static <T> SortedSet<T>
                        synchronizedSortedSet(SortedSet<T> s),
public static <K,V> SortedMap<K,V>
                       synchronizedSortedMap(SortedMap<K,V> m),
```

які повертають синхронізовані колекції, які забезпечують безпеку потоків завдяки використанню моніторів та синхронізації доступу потоків до методів колекцій. Такі колекції є статичними вкладеними класами класу Collections і мають поля - посилання на колекцію, яка синхронізується (параметр відповідного методу з наведених вище) та об'єкт, монітор якого використовується для синхронізації, наприклад:

```
static class SynchronizedCollection<E>
                       implements Collection<E>, Serializable {
    private static final long serialVersionUID =
                                          3053995032091335093L;
    final Collection<E> c;  // Backing Collection
    final Object mutex;     // Object on which to synchronize
    SynchronizedCollection(Collection<E> c) {
        this.c = Objects.requireNonNull(c);
        mutex = this;
    }
…
    public boolean add(E e) {
        synchronized (mutex) {return c.add(e);}
    }
…
```

У прикладі наведений код методу `boolean add(E e)`, який використовує синхронізований блок, аналогічно побудовані і інші методи роботи з колекцією, яку огортає об'єкт синхронізованої колекції.

Безумовно ці засоби забезпечують потокобезпечну роботу з такими колекціями, але мають недолік у зменшенні ефективності, який був описаний вище.

Засоби потокобезпечних колекцій Java Concurrent API можна згрупувати:

1. Черги - які можна розділити на неблокуючі черги, засновані на зв'язних списках - ConcurrentLinkedQueue<E> і ConcurrentLinkedDeque<E> (JDK 7) і блокуючі черги (реалізації інтерфейсів BlockingQueue<E>, BlockingDeque<E> і TransferQueue<E> (JDK 7)), що дозволяють задати їх розмір та/або умови блокування (наприклад, в разі наближення до граничного розміру або видалення з порожньої черги).
2. Покращені реалізації HashMap, TreeMap з кращого підтримкою багатопоточності і масштабованості - реалізації інтерфейсів ConcurrentMap<K, V> і ConcurrentNavigableMap<K, V> - додані атомарні операції, наприклад додавання пари, тільки якщо ключ відсутній і видалення і заміна пари тільки, якщо ключ присутній у об'єкті-карті.

3. CopyOnWrite колекції - всі операції по зміні колекції (add, set, remove) призводять до створення нової копії внутрішнього масиву. Тим самим гарантується, що при проході ітератором по колекції не згенерується ConcurrentModificationException.

Типове використання блокуючої черги - додавання до неї об'єктів одним потоком і витягування іншим.

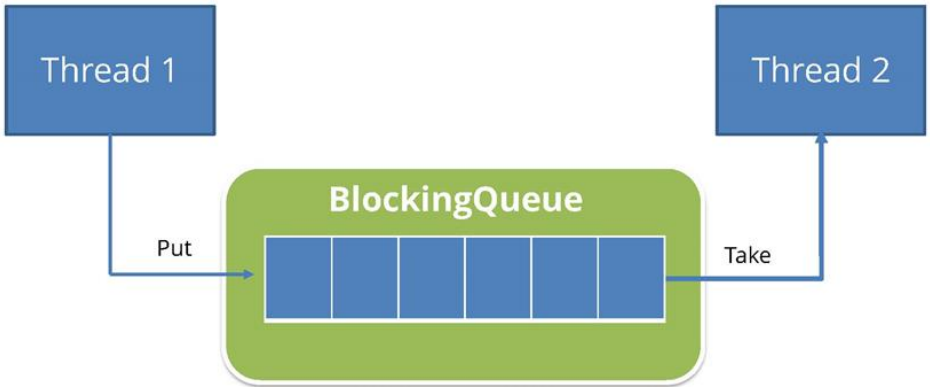

Рис. 8. Робота блокуючої черги

Можливі реалізації інтерфейсу BlockingQueue<E>:

ArrayBlockingQueue<E> - побудована на класичному кільцевому буфері. Крім розміру черги, доступна можливість керувати «чесністю» блокувань. Якщо `fair=false` (за замовчуванням), то черговість роботи потоків не гарантується.

DelayQueue<E extends Delayed> - дозволяє витягувати елементи з черги тільки після деякої затримки, визначеної в кожному елементі через метод `getDelay` інтерфейсу Delayed.

LinkedBlockingQueue<E> - черга на зв’язному списку, в ній реалізовані два блокування: одне - на додавання, інше - на витягування елемента. За рахунок цього, в порівнянні з ArrayBlockingQueue, даний клас показує більш високу продуктивність, але і витрати пам'яті у нього вище. Розмір черги задається через конструктор і за замовчуванням дорівнює Integer.MAX_VALUE.

PriorityBlockingQueue<E> - багатопотокова обгортка над PriorityQueue. При доданні елемента в чергу, його порядок визначається відповідно до логіки Comparator'a або імплементації Comparable інтерфейсу у класі елементів. Першим з черги виходить самий найменший елемент.

SynchronousQueue<E> - працює за принципом один увійшов, один вийшов. Кожна операція вставки блокує потік постачальника, доки потік споживача не витягне елемент з черги, і, навпаки, споживач чекатиме, поки постачальник не вставить елемент.

Розглянемо приклад використання блокуючої черги, що використовується об'єктами класів постачальника (Producer) та споживача (Consumer):

```
public class Producer implements Runnable {

    private final BlockingQueue<Long> queue;
    private long i;
```

```
    public Producer(BlockingQueue<Long> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()
                                                && i < 5) {
                queue.put(produce());
                System.out.println("Producer: There are "
                  + queue.size() + " elements in the queue");
                Thread.sleep(1000);
            }
        } catch (InterruptedException ex) {
        }
    }

    private Long produce() {
        long obj = i++;
        System.out.println("Produser: produced object: "
                                                    + obj);
        return obj;
    }
}
```

Об’єкт класу Producer постачає кожну 1 секунду об'єкти (5 всього) в блокуючу чергу з обмеженою кількістю елементів (вказується при створенні черги в RunTime). Поки поточний потік не перерваний, в чергу додаються об'єкти типу Long (кожен наступний - інкремент попереднього). Метод `void put(E e)` додає елемент в чергу, чекаючи, якщо вона зайнята повністю, і може викликати InterruptedException, якщо потік переривається під час очікування звільнення черги.

```
public class Consumer implements Runnable {

    private final BlockingQueue<Long> queue;

    public Consumer(BlockingQueue<Long> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        try {
            System.out.println("Consumer: Trying to take
                                        object from queue...");
            while (!Thread.currentThread().isInterrupted()) {
                consume(queue.poll(2, TimeUnit.SECONDS));
                System.out.println("Consumer: There are "
                  + queue.size() + " elements in the queue");
            }
```

```
            } catch (InterruptedException ex) {
            }
        }

        private void consume(Long dt) {
            if (dt == null) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Consumer: consumed object: "+dt);
        }
}
```

Об’єкт класу Consumer вилучає об'єкти з блокуючої черги з обмеженим числом елементів, доки поточний потік не буде перерваний. Метод `E take()` вилучає елемент з початку черги, чекаючи появи елемента, якщо елементи відсутні повністю, і може викликати InterruptedException, якщо потік переривається під час очікування звільнення черги. Метод `E poll(long timeout, TimeUnit unit)`, використаний у прикладі, вилучає елементи і повертає `null`, якщо протягом таймаута-параметра немає елементів.

Метод `void consume(Long dt)` виводить в консоль отримані з черги елементи.

Тоді у деякому класі демонструється робота блокуючої черги в реалізації ArrayBlockingQueue<E>.

```
public static void main(String[] args) {
    BlockingQueue<Long> q = new ArrayBlockingQueue<>(2);
    Producer p = new Producer(q);
    Consumer c = new Consumer(q);
    Thread thp = new Thread(p);
    Thread thc = new Thread(c);
    thp.start();
    thc.start();
    try {
        thp.join();
        thc.join();
    } catch (InterruptedException ex) {
    }
}
```

Запуск програми демонструє роботу черги:

```
Consumer: Trying to take object from queue...
Produser: produced object: 0
Producer: There are 1 elements in the queue
Consumer: consumed object: 0
Consumer: There are 0 elements in the queue
Produser: produced object: 1
Producer: There are 1 elements in the queue
Consumer: consumed object: 1
Consumer: There are 0 elements in the queue
Produser: produced object: 2
```

```
Producer: There are 1 elements in the queue
Consumer: consumed object: 2
Consumer: There are 0 elements in the queue
Produser: produced object: 3
Consumer: consumed object: 3
Producer: There are 0 elements in the queue
Consumer: There are 0 elements in the queue
Produser: produced object: 4
Consumer: consumed object: 4
Consumer: There are 0 elements in the queue
Producer: There are 0 elements in the queue
Consumer: consumed object: null
Consumer: There are 0 elements in the queue
```

Бачимо, що споживач витягує з черги елементи, які додає в неї постачальник, доки метод `poll` споживача не поверне `null` і потік споживача не буде перерваний.

Інтерфейс BlockingDequeue<E> реалізує тільки один клас - LinkedBlockingDeque<E>. Він містить методи, якими потоки безпечно можуть додавати і витягувати елементи з обох кінців, реалізуючи як LIFO, так і FIFO правило.

Ефективність роботи реалізацій інтерфейсів BlockingQueue<E> та BlockingDequeue<E> () забезпечується використанням об'єктів ReentrantLock та Condition.

Інтерфейс ConcurrentMap<K,V> реалізують два класи - ConcurrentHashMap та ConcurrentSkipListMap. Робота ConcurrentHashMap аналогічна HashMap, але підтримує потокобезпечність. У ConcurrentHashMap відмінність полягає у внутрішній структурі для зберігання пар `key-value`. ConcurrentHashMap має додаткову концепцію сегментів (Рис. 9).

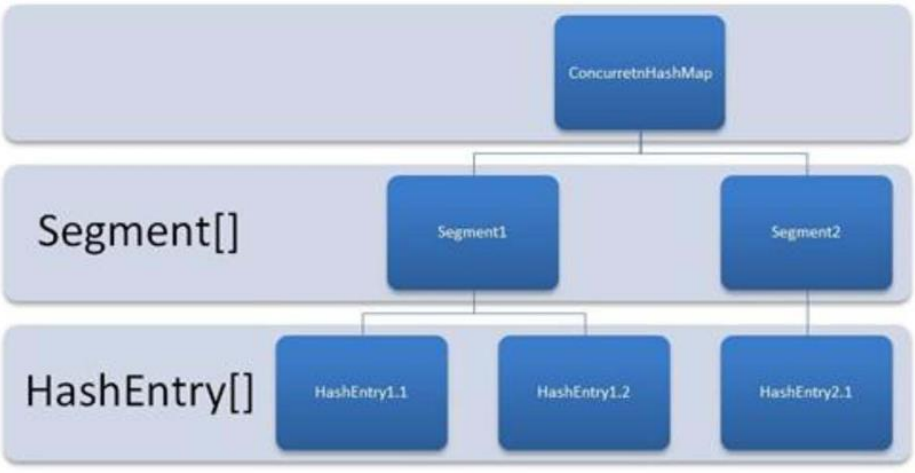

Рис. 9. Структура ConcurrentHashMap

Буде легко зрозуміти якщо уявити, що один сегмент еквівалентний одному HashMap (концептуально). ConcurrentHashMap розділена на декілька сегментів (за замовчуванням їх число дорівнює 16, максимальне значення обмежується 16-бітами і являє собою ступінь двійки) при ініціалізації. Між хеш-кодами ключів і

відповідними їм сегментами встановлюється залежність на основі застосування до старших розрядів хеш-коду бітової маски.

З огляду на псевдовипадковий розподіл хеш-кодів ключів всередині таблиці, можна зрозуміти, що збільшення кількості сегментів буде сприяти тому, що операції модифікації будуть зачіпати різні сегменти, що зменшить ймовірність блокувань під час роботи. Для кращої продуктивності використовуються операції CAS під час оновлення ConcurrentHashMap. Розглянемо приклад застосування ConcurrentHashMap:

```
public class TestConcHashMap extends Thread {

    private String name;
    private static Map<String, String> cmap =
                                    new ConcurrentHashMap<>();

    public TestConcHashMap(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        cmap.put(name + "1", "AA"); //Буде замінений
        cmap.put(name + "2", "B");
        cmap.put(name + "3", "C");
        cmap.put(name + "4", "D");
        cmap.put(name + "5", "E");
        cmap.put(name + "1", "A");  //Замінений
        System.out.println(name + " completed");
    }

    public static void main(String[] args) {
        TestConcHashMap th1 = new TestConcHashMap("One");
        TestConcHashMap th2 = new TestConcHashMap("Two");
        th1.start();
        th2.start();
        try {
            th1.join();
            th2.join();
        } catch (InterruptedException ex) {
        }
        System.out.println(cmap);
        System.out.println("main finished");
    }
}
```

Запуск програми виводить:

```
One completed
Two completed
{Two3=C, Two2=B, Two1=A, One3=C, One2=B, One5=E, One4=D, One1=A, Two5=E,
Two4=D}
main finished
```

Бачимо, що 2 потоки додали до колекції по 5 елементів (та по одному оновили). Всі 10 елементів знаходяться в ConcurrentHashMap.

Порівняння ефективності використання засобів Collection API: HashTable, SynchronizedMap та ConcurrentHashMap для додання та видалення елементів колекції демонструють підвищення швидкості роботи для останньої в 3-3,5 рази.

Робота класів CopyOnWriteArrayList<E> та CopyOnWriteArraySet<E> базується на тому, що всі операції, які змінюють колекцію (`add`, `set`, `remove`) призводять до створення нової копії внутрішнього масиву елементів. Тим самим гарантується, що при проході ітератором по колекції не згенерується ConcurrentModificationException. Наведемо вихідний код метода додання елемента у колекцію класу CopyOnWriteArrayList<E>, який ілюструє копіювання колекції:

```
public class CopyOnWriteArrayList<E> implements List<E>,
             RandomAccess, Cloneable, java.io.Serializable {
    final transient Object lock = new Object();
    private transient volatile Object[] array;
…
    public boolean add(E e) {
        synchronized(lock) {
            Object[] es = getArray();
            int len = es.length;
            es = Arrays.copyOf(es, len + 1);
            es[len] = e;
            setArray(es);
            return true;
        }
    }
…
```

Додавання елемента виконується в копію масиву. Відсутність взаємного впливу потоків при читанні і запису посилання на масив забезпечує модифікатор `volatile` для змінної, що зберігає посилання масиву. Слід пам'ятати, що при копіюванні масиву копіюються тільки посилання на об'єкти (shallow copy), доступ до полів елементів не потоконебезпечний. CopyOnWrite колекції зручно використовувати, коли операції запису досить рідкісні, наприклад при реалізації механізму реєстрації слухачів подій і проходу по ним.

Приклад використання CopyOnWriteArrayList<E>:

```
public static void main(String[] args) {
    CopyOnWriteArrayList<Integer> numbers =
    new CopyOnWriteArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(250);
```

```
            } catch (InterruptedException ex) {
            }
            numbers.add(10);
            System.out.println("Thread-0: numbers: "
                                                    + numbers);
        }
    });
    t.start();
    System.out.println("Main: iterate list...");
    for (int i : numbers) {
        System.out.println(i);
        try {
            Thread.sleep(100);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Після запуску програма виводить:

```
Main: iterate list...
1
2
3
Thread-0: numbers: [1, 2, 3, 4, 5, 10]
4
5
```

Реалізація ефективного паралельного алгоритму є нетривіальним завданням в більшості мов програмування: нам потрібно визначити, як розбити задачу, визначити оптимальний рівень паралелізму, і, нарешті, побудувати реалізацію з мінімальною синхронізацією. В JDK 7 з'явився новий *Fork/Join* фреймворк для вирішення рекурсивних задач. Принцип його роботи полягає у тому, що фреймворк спочатку *"розгалужує"* (*fork*) завдання, рекурсивно розбиваючи його на більш дрібні незалежні підзадачі, поки вони не стануть достатньо простими для асинхронного виконання. Після цього починається фаза *"з'єднання"* (*join*), у якій результати всіх підзадач рекурсивно об'єднуються в один результат, або в разі завдання, яке повертає `void`, програма просто чекає, поки не буде виконана кожна підзадача.

Для забезпечення ефективного паралельного виконання фреймворк Fork/Join використовує пул потоків ForkJoinPool, який керує робочими потоками типу ForkJoinWorkerThread. ForkJoinPool - реалізація ExecutorService, яка управляє робочими потоками і надає нам інструменти для отримання інформації про стан і продуктивність пулу потоків. Робочі потоки можуть одночасно виконувати тільки одну задачу (об’єкт ForkJoinTask(V)), але ForkJoinPool не створює окремий потік для кожної окремої підзадачі (також об’єкти ForkJoinTask(V)). Замість цього кожен потік в пулі має свою власну двосторонню чергу, в якій зберігаються завдання. Ця архітектура життєво важлива для балансування робочого

навантаження потоку за допомогою *алгоритму крадіжки роботи* (*work-stealing algorithm*). Даний алгоритм передбачає, що за замовчуванням робочий потік отримує завдання з заголовка своєї черги, але коли черга порожня, потік бере завдання з хвоста черги іншого зайнятого потоку або з глобальної вхідної черги, оскільки саме тут, ймовірно, будуть перебувати найбільші частини роботи. Такий підхід зводить до мінімуму ймовірність того, що потоки будуть конкурувати за завдання. Це також зменшує кількість пошуків роботи потоком, оскільки спочатку він буде працювати з найбільшими доступними частинами роботи (Рис. 10).

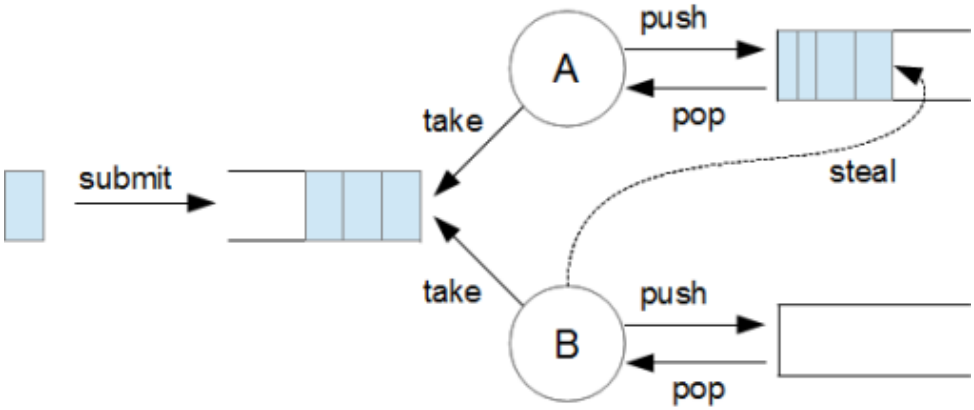

Рис. 10. Робота ForkJoinPool (алгоритм крадіжки роботи)

Об’єкт ForkJoinPool є точкою входу для запуску кореневих (main) ForkJoinTask(V) завдань. Підзадачі запускаються через методи завдання, від якої потрібно відстріл (fork). За замовчуванням створюється пул потоків з кількістю потоків рівною кількості доступних для JVM процесорів (cores).

ForkJoinTask(V) - абстрактний клас для всіх ForkJoin завдань. З його ключових методів можна відзначити: `ForkJoinTask<V> fork()` - додає завдання в чергу поточного потоку ForkJoinWorkerThread для асинхронного виконання; `V invoke()` - запускає завдання в поточному потоці; `V join()` - очікує завершення підзадачі з поверненням результату; `invokeAll(...)` - об'єднує всі три попередні операції, виконуючи два або більше завдань за один захід; `adapt(...)` - створює нову задачу ForkJoinTask з задачі Runnable або задачі Callable.

RecursiveTask(V) - абстрактний клас - спадкоємець ForkJoinTask(V), з оголошенням методу `abstract V compute()`, в якому слід виконувати асинхронну операцію.

RecursiveAction - абстрактний клас - спадкоємець ForkJoinTask(Void), відрізняється від RecursiveTask(V)тим, не повертає результат.

ForkJoinWorkerThread - використовується як потік, що управляється ForkJoinPool, виконує завдання ForkJoinTask(V).

Зміст підзадач-об'єктів RecursiveTask<V> вказується в перевизначеному методі `protected abstract V compute()`, в якому за допомогою виклику

методу `public final ForkJoinTask<V> fork()` виконується розподіл завдань, а за допомогою виклику методу `V join()` - повернення результатів обчислення.

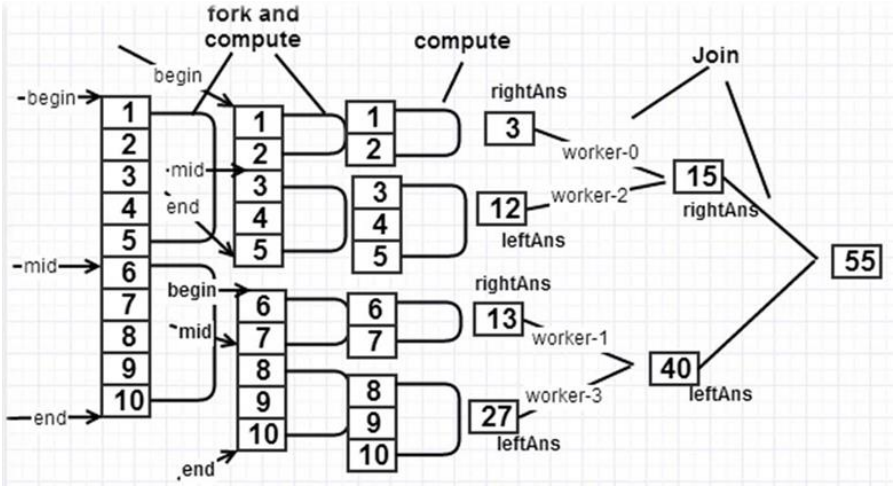

Рис. 11. Виконання рекурсивного завдання фреймворком Fork/Join

Наведемо приклад використання фреймворку Fork/Join для обчислення суми чисел від 1 до n = 1 млн десятьма потоками. Діапазон чисел ділиться навпіл, поки результуючий піддіапазон може бути оброблений потоком. Після того, як обчислення в піддіапазонах завершуються, вони зводяться в загальний результат.

```
public class SumOfNUsingForkJoin {

    private static final long N = 1_000_000L;
    private static final int NUM_THREADS = 10;

    static class RecSumOfN extends RecursiveTask<Long> {

        long from;
        long to;

        public RecSumOfN(long from, long to) {
            this.from = from;
            this.to = to;
        }

        @Override
        protected Long compute() {
            /*Знаходимо середину вихідного діапазону*/
            long mid = (from + to) / 2;
            System.out.printf("Forking computation into two "
                        +"ranges: %d to %d and %d to %d %n",
                        + from, mid, mid, to);
            /*Отримуємо підзадачу для 1 половини діапазону*/
            RecSumOfN firstHalf = new RecSumOfN(from, mid);
            /*Виділяємо в окрему підзадачу підсумовування
              у новому діапазоні*/
            firstHalf.fork();
            /*Отримуємо підзадачу для 2 половини діапазону*/
```

```
            RecSumOfN secondHalf = new RecSumOfN(mid+1, to);
            long resultSecond = secondHalf.compute();
            /*Очікуємо результат обчислень в 1 піддіапазоні і
              отримавши його додаємо до результату обчислення
              у 2 піддіапазоні і повертаємо загальний
              результат*/
            return firstHalf.join() + resultSecond;
    }
}

    public static void main(String[] args) {
        /*Створюємо ForkJoin пул з NUM_THREADS потоків*/
        ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);
        System.out.println("The pool is: "
                                        + pool.toString());
        /*Надсилаємо завдання RecSumOfN(0, 1_000_000) в пул*/
        long computedSum = pool.invoke(new RecSumOfN(0, N));
        System.out.println("The pool is: "
                                        + pool.toString());
        /*Формула підсумовування чисел в діапазоні 1 ... N*/
        long comapredFormulaSum = (N * (N + 1)) / 2;
        System.out.printf("Sum for range 1..%d;
                  computed sum = %d, formula sum = %d %n", N,
                  computedSum, comapredFormulaSum);
    }
}
```

Вкладений клас RecSumOfN успадковується від RecursiveTask<Long>, а не від RecursiveAction оскільки необхідно повернути результат. Властивостями класу є межі діапазону обчислень підзадачею. Метод `join()` змушує поточне завдання очікувати, поки не завершиться підзадача, запущена методом `fork()`.

Запуск програми виводить:

```
The pool is: java.util.concurrent.ForkJoinPool@12bb4df8[Running, parallelism = 4, size = 0,
active = 0, running = 0, steals = 0, tasks = 0, submissions = 0]
Forking computation into two ranges: 0 to 500000 and 500000 to 1000000
Forking computation into two ranges: 500001 to 750000 and 750000 to 1000000
Worker: ForkJoinPool-1-worker-3 is working!!!
        Summing of value range 750001 to 1000000 is 218750125000
Worker: ForkJoinPool-1-worker-3 is working!!!
        Summing of value range 500001 to 750000 is 156250125000
Forking computation into two ranges: 0 to 250000 and 250000 to 500000
Worker: ForkJoinPool-1-worker-3 is working!!!
        Summing of value range 250001 to 500000 is 93750125000
Worker: ForkJoinPool-1-worker-3 is working!!!
        Summing of value range 0 to 250000 is 31250125000
The pool is: java.util.concurrent.ForkJoinPool@12bb4df8[Running, parallelism = 4, size = 4,
active = 3, running = 3, steals = 1, tasks = 0, submissions = 0]
Sum for range 1..1000000; computed sum = 500000500000, formula sum = 500000500000
```

Як бачимо результат обчислень фреймворком Fork/Join та результат математичної функції співпадає. Використання фреймворку Fork/Join підвищує продуктивність для достатньо складних обчислювальних завдань. Для наведеного прикладу виграш спостерігається при кількості чисел N більше 1 млрд.

2. Завдання

1. Оберіть завдання у наведеній нижче таблиці. Номер варіанта визначте за формулою V = (№ mod 22) +1, де № - Ваш порядковий номер в журналі академгрупи. Виконайте завдання, використовуючи пул потоків із явним завданням кількості потоків та кількості місць для завдань у черзі. Обирайте тип завдання Runnable або Callable відповідно до Вашого завдання.

V	Завдання	V	Завдання
1	Напишіть програму, яка дозволить збільшити значення спільної змінної x від 5 до 10 двома потоками: Thread-0, який збільшує x на 1 та Thread-1, який збільшує x на 2.	12	Напишіть програму-таймер, до якої надходить період часу і яка щосекундно зменшує цей період до нуля. Запустіть такий таймер у трьох різних потоках для різних періодів часу
2	Напишіть програму із використанням декількох потоків, яка знаходить ціле число в діапазоні від 1 до 100000, яке має найбільшу кількість дільників. Побудуйте функцію залежності часу виконання завдання від кількості потоків.	13	Напишіть програму, яка запускає два потоки, перший з яких виводить ціл числа від 1 до 52, а другий - літери від A до Z, так, щоб програма виводила 12A34B ... 5152Z.
3	Напишіть програму із використанням декількох потоків, яка знаходить для масиву `double` елементів з 100000 випадковими значеннями від 0 включно до π виключно знайдіть індекс елемента з максимальним значенням синуса від значення елементу. Побудуйте функцію залежності часу виконання завдання від кількості потоків.	14	Напишіть програму, яка імітує підкидання монети та отримання орла або решки. Запустіть цю програму у 5 потоках та виведіть текст з ідентифікатором потоків, коли вони отримають 3 орли поспіль.
4	Напишіть програму із використанням декількох потоків так, що Потік 1 створює інший потік (Потік 2); Потік 2 створює Потік 3; і так далі, до Потік 50. Кожен потік повинен виводити у консоль рядок "Hello from Thread<num>!", де num - ідентифікатор потоку, але потоки повинні друкували свої привітання в зворотному порядку.	15	Напишіть програму, яка обробляє вхідні дзвінки від користувачів та імітує відповіді на них а між дзвінками імітує читання книги.
5	Напишіть програму із використанням декількох потоків, яка для файлу у форматі .txt переведе всі текстові символи у верхній регістр. Для текстового файлу 1 тому роману Л.М. Толстого "Війна та мир" виконайте таку заміну та побудуйте функцію залежності часу	16	Напишіть програму, яка імітує потік повідомлень, які виводяться з масиву повідомлень-рядків на консоль через певний проміжок часу. Основний потік періодично перевіряє чи друкуються повідомлення, і, якщо час перевірки

V	Завдання	V	Завдання
	виконання завдання від кількості потоків.		перевищує заданий граничний час, перериває роботу потоку повідомлень
6	Напишіть програму із використанням декількох потоків, яка виконує шифрування та дешифрування тексту шифром XOR (використовуйте псевдовипадкову послідовність байт). Виконайте шифрування та дешифрування для великого текстового файлу та побудуйте функцію залежності часу виконання завдання від кількості потоків.	17	Напишіть програму, яка розраховує добуток узгоджених матриць із використанням декількох потоків, а потім обраховує визначник результуючої матриці. Виконайте розрахунок для матриць розміром 256x512 елементів з випадковими цілими числами та побудуйте функцію залежності часу виконання завдання від кількості потоків.
7	Напишіть програму із використанням декількох потоків, яка підраховує суму елементів масиву випадкових цілих чисел у межах заданих як параметр нижнього та верхнього індексів. Побудуйте функцію залежності часу виконання завдання від кількості потоків.	18	Напишіть програму із використанням декількох потоків, яка виконує пошук елементів з максимальним значенням та виводить їх індекси. Виконайте пошук для масиву випадкових цілих чисел розміром 100000 елементів та побудуйте функцію залежності часу виконання завдання від кількості потоків.
8	Напишіть програму, яка дозволить отримати результат x=3, розрахований потоком Thread-0, якщо потік Thread-1 встановлює спільну змінну y = 1, а потік Thread-2 - встановлює спільну змінну z = 2.	19	Напишіть програму із використанням декількох потоків, яка виконує збільшення зображення, яке задається двовимірним масивом пікселів, кожен з яких має значення від 0 до 255 для трьох кольорів та каналу альфа (обирайте середнє значення кольорів та альфаканалу для кожного рядку пікселів). Виконайте збільшення зображення з 640x360 до 1280x720 та побудуйте функцію залежності часу виконання завдання від кількості потоків.
9	Напишіть програму із використанням декількох потоків, яка знаходить значення інтегралу функції $\int_0^1 \frac{dx}{\sqrt{x^3+1}}$ з точністю до 0,00001. Побудуйте функцію залежності часу виконання завдання від кількості потоків.	20	Напишіть програму із використанням декількох потоків, яка обраховує квадрати випадкових значень елементів (від 0 до 1000) масиву double а потім знаходить їх суму. Виконайте програму для масиву розміром 100000 елементів та побудуйте функцію залежності часу виконання завдання від кількості потоків.
10	Напишіть програму із використанням декількох потоків, яка знаходить прості числа в масиві випадкових цілих чисел з 10000 елементами. Побудуйте функцію залежності часу виконання завдання від кількості потоків.	21	Напишіть програму із використанням декількох потоків, яка імітує виробничу лінію. Лінія виготовляє деталь, яка збирається з деталі C і модуля, який, в свою чергу, складається з деталей A і B. Для виготовлення деталі A потрібна 1 секунда, B - дві секунди, C - три секунди.
11	Напишіть програму із використанням декількох потоків, яка виконує	22	Напишіть програму із використанням декількох потоків, яка знаходить суму

V	Завдання	V	Завдання
	сортування алгоритмом швидкого сортування масиву випадкових цілих чисел розміром у 100000 елементів. Побудуйте функцію залежності часу виконання завдання від кількості потоків.		елементів масиву з випадковими цілими значеннями від 0 до 1000. Знайдіть суму елементів для масиву розміром у 100000 та побудуйте функцію залежності часу виконання завдання від кількості потоків.

2. У разі, якщо Ваше завдання №1 є рекурсивним, розробіть програму з використанням Fork/Join фреймворка. Порівняйте час виконання завдання з попередньою реалізацією на пулі потоків. Підберіть кількість завдань, коли реалізація за допомогою Fork/Join фреймворка буде більш ефективною.
3. Оберіть завдання у наведеній нижче таблиці. Номер варіанта визначте за формулою V = (№ mod 22) +1, де № - Ваш порядковий номер в журналі академгрупи. Виконайте завдання, організуючи синхронізацію потоків за допомогою об'єктів класів-реалізацій інтерфейсів Lock та Condition.

V	Завдання	V	Завдання
1	Напишіть програму з використанням координації потоків, що імітує роботу перукарні, у якій є одне крісло для стрижки та N крісел для відвідувачів, які чекають. Коли приходить відвідувач і крісло для стрижки вільно, відвідувач сідає в нього і перукар починає його стригти. В іншому випадку відвідувач сідає в крісло для очікування. Якщо все крісла зайняті, відвідувач йде.	12	Напишіть програму з використанням координації потоків, що імітує спосіб обслуговування клієнтів в офісі. В офісі є N столів, де клієнти можуть бути обслужені одночасно. Клієнти беруть квиток з номером і чекають, поки їх номер не стане наступним для обслуговування на одному зі столів. Клієнти обслуговуються в порядку, зазначеному в їх квитку. Симуляція повинна припинитися після того, як певна кількість клієнтів отримають квитки і їх обслужать.
2	Напишіть програму з використанням координації потоків, що імітує атракціон "Американські гірки". Є N процесів-пасажирів і один процес-вагончик. Пасажири чекають черги проїхати у 8 вагончиках, що вміщують 32 людини.	13	Напишіть програму з використанням координації потоків, що імітує роботу порту. У порту працює N причалів, у одного причалу може стояти один корабель. Кораблі заходять в порт для розвантаження контейнерів, кожен корабель містить M контейнерів.
3	Напишіть програму з використанням координації потоків, що імітує пастуха, собаку та двох овець. Вівці пасуться на обмеженій ділянці пасовища і пересуваються випадковим чином. Пастух зазвичай спить. Собака слідкує, щоб вівці не виходили за заданий периметр. Якщо якась одна вівця покинула периметр, то собака заганяє її назад (протягом деякого випадкового часу). Якщо в цей момент ще одна вівця покинула периметр, то собака будить своїм гавкотом пастуха, і вони разом починають	14	Напишіть програму з використанням координації потоків, що імітує роботу готелю з N номерами. Клієнт приходить, заселяється в номер або чекає поки звільниться, якщо все зайнято. Якийсь час живе в номері, а потім його звільняє.

V	Завдання	V	Завдання
	заганяти овець на пасовище (кожен жене свою вівцю).		
4	Напишіть програму з використанням координації потоків, що імітує душову у квартирі гуртожитку, у якій у 9 кімнатах проживають 12 хлопців та 10 дівчат. Душовою можуть користуватися і хлопці і дівчата, але не одночасно. Одночасно в душовій може перебувати 4 особи.	15	Напишіть програму з використанням координації потоків, що імітує роботу бібліотеки. У бібліотеці доступні до читання N книг. Деякі з них можна видавати на руки деякі тільки в читальний зал. Відвідувачі можуть брати одночасно по кілька книг на руки і в читальний зал
5	Напишіть програму з використанням координації потоків, що імітує годування N пташенят їх матір'ю. Пташенята їдять із загальної миски, в якій спочатку знаходиться M порцій їжі. Кожне пташеня з'їдає порцію їжі, спить деякий час, потім знову їсть. Якщо вистачає їжі, то одночасно можуть їсти кілька пташенят. Коли закінчується їжа, пташеня, що їв останнім, кличе мати. Мати наповнює миску M порціями їжі і знову чекає, поки миска спорожніє. Ці дії повторюються без кінця.	16	Напишіть програму з використанням координації потоків, що імітує роботу буферу даних, який заповнюється одним потоком, а зчитується та обробляється декількома (на обробку даних потрібен певний час).
6	Напишіть програму з використанням координації потоків, що імітує ведмедя, який їсть мед, зібраний N бджолами. Бджоли збирають мед у горщик, що вміщує M порцій меду. Спочатку горщик порожній і ведмідь спить. Кожна бджола багаторазово збирає по одній порції меду і кладе її в горщик. Бджола, яка приносить останню порцію меду і заповнює горщик, будить ведмедя і він з'їдає увесь мед у горщику.	17	Напишіть програму з використанням координації потоків, що імітує роботу диспетчера подій. N потоків генерують об'єкти-події у яких фіксується ідентифікатор потоку та послідовний номер події, а один потік обробляє події та виводить їх послідові номери та ідентифікатори потоків-генераторів у напрямку зростання номерів подій.
7	Напишіть програму з використанням координації потоків, що реалізує три класи: сховище, лічильник та принтер. Клас Storage повинен зберігати ціле число. Клас Counter повинен створити потік, який починає відлік від 0 (0, 1, 2, 3 ...) до 1000000 і зберігає кожне значення в класі Storage. Клас Printer повинен створити потік, який продовжує читати значення в класі Storage і друкувати його рівно один раз.	18	Напишіть програму з використанням координації потоків, що імітує роботу системи передачі повідомлень. Один процес виробляє повідомлення, призначені для сприйняття і обробки іншим процесом. Процеси взаємодіють через деяку узагальнену область пам'яті.
8	Напишіть програму з використанням координації потоків, що імітує обслуговування відвідувачів піцерії, у якій 20 місць. Піч випікає по 5 піц за раз.	19	Напишіть програму з використанням координації потоків, що імітує систему запису-читання. Процеси-читачі зчитують одночасно інформацію з деякого

V	Завдання	V	Завдання
	Введіть граничний час очікування, після якого відвідувач уходить, не дочекавшись піци.		пристрою. Процеси-письменники записують інформацію в область і можуть робити це, тільки виключаючи як один одного, так і процеси-читачі.
9	Напишіть програму з використанням координації потоків, що імітує роботу автостоянки. Машина рухається через автостоянку, доки не виявляє першого пустого місця, стоїть на ньому деякий час і виїжджає зі стоянки. Через деякий час машина повертається до автостоянки. Якщо всі місця на стоянці зайняті, машина виїжджає.	20	Напишіть програму з використанням координації потоків, що імітує сервер, який обробляє клієнтські запити і зберігає дані-текстові рядки клієнтів у файл.
10	Напишіть програму з використанням координації потоків, що імітує рух по вузькому мосту. Машини, що рухаються в одному напрямку, можуть переїжджати міст одночасно, а в протилежному - ні.	21	Напишіть програму з використанням координації потоків, що імітує роботу лабораторії, у якій одночасно можуть обслуговувати N клієнтів за певний однаковий час кожного. Клієнти попадають до лабораторії за чергою.
11	Напишіть програму з використанням координації потоків, що імітує безконфліктний рух на перехресті двох двосторонніх доріг шириною у дві смуги кожна (тобто імітувати роботу світлофору, який буде перемикатися, оцінюючи обстановку на дорозі).	22	Напишіть програму з використанням координації потоків, що імітує роботу N кас супермаркету, які упродовж певного часу обслуговують покупців.

4. Наведіть код програм у звіт. Поясніть стратегію синхронізації, яку Ви обрали при вирішенні завдання.
5. У висновку вкажіть особливості роботи з використаними Вами засобами Java Concurrency API.

3. Контрольні питання

1. У якій версії Java був введений Concurrency API? Назвіть пакет, у якому розміщені його інструменти.
2. Надайте загальну характеристику інструментам Java Concurrency API.
3. Назвіть переваги використання пулів потоків та опишіть їх роботу.
4. Надайте характеристику інтерфейсам виконавців, які забезпечують функціонування пулів потоків, та опишіть їх основні методи.
5. Поясніть відміну завдань Runnable та Callable<T>, що передаються до пулу потоків. Опишіть призначення та методи інтерфейсу Future<V>.
6. Назвіть етапи організації виконання завдання Runnable у потоках, що управляються об’єктом ExecutorService.
7. Опишіть код програми, яка використовує ThreadPoolExecutor для запуску завдання Runnable. Опишіть параметри конструктора класу ThreadPoolExecutor.

8. Опишіть методи класу-фабрики Executors. Опишіть код програми, яка використовує Executors для запуску завдання Callable у пулі з фіксованою кількістю потоків.
9. Назвіть функціональні обмеження організації синхронізації потоків за допомогою `synchronized` та вбудованого механізму `wait/notify`.
10. Надайте загальну характеристику інтерфейсам пакету java.util.concurrent.locks.
11. Надайте загальну характеристику класам-реалізаціям інтерфейсів Lock та ReadWriteLock.
12. Що називають блокуванням з повторним входом (*reentrant*)? Охарактеризуйте методи інтерфейсу Lock, реалізовані в класі ReentrantLock. Коли рекомендується використовувати об'єкти ReentrantLock замість synchronized?
13. Порівняйте код програм, які використовують для синхронізації потоків `synchronized` метод та об'єкт ReentrantLock, відповідно.
14. Опишіть код програми, яка використовує метод об'єкта ReentrantLock для синхронізації, не змушуючи потоки чекати невизначений час блокування.
15. Порівняйте код програм, які використовують для координації потоків механізм `wait/notify/notifyAll` та об'єкти Condition разом з об'єктом ReentrantLock, відповідно.
16. Опишіть код програми, яка демонструє повторне отримання блокування потоком на тому ж об'єкті ReentrantLock.
17. Надайте загальну характеристику об'єктам-синхронізаторам.
18. Опишіть принцип роботи семафору (java.util.concurrent.Semaphore), параметри його конструктора та основні методи. Що називають вазємовиключаючим семафором (*Mute*x-ом)?
19. Опишіть код програми, яка демонструє використання семафору (java.util.concurrent.Semaphore).
20. Опишіть принцип роботи засувки зі зворотним відліком (java.util.concurrent.CountDownLatch), параметри його конструктора та основні методи.
21. Опишіть код програми, яка демонструє використання засувки зі зворотним відліком (java.util.concurrent.CountDownLatch).
22. Опишіть принцип роботи циклічного бар'єру (java.util.concurrent.CyclicBarrier), параметри його конструктора та основні методи.
23. Опишіть код програми, яка демонструє використання циклічного бар'єру (java.util.concurrent.CyclicBarrier).
24. Опишіть принцип роботи обмінника (java.util.concurrent.Exchanger<V>) та його основні методи.
25. Опишіть код програми, яка демонструє використання обмінника (java.util.concurrent.Exchanger<V>).
26. Опишіть принцип роботи об'єктів класу Phaser, його конструктори та основні методи.
27. Опишіть код програми, яка демонструє використання об'єктів класу Phaser.

28. Що називають атомарним (atomic) операціями? Охарактеризуйте песимістичний та оптимістичний підходи до доступу до загального ресурс декількох потоків.
29. Опишіть алгоритм порівняння з обміном (Compare And Swap).
30. Надайте загальну характеристику класам пакету java.util.concurrent.atomic.
31. Опишіть код програми, яка порівнює ефективність багатопотокової роботи з загальним ресурсом за допомогою `volatile`/`synchronized` та атомарних змінних та операцій.
32. Виконайте загальне порівняння синхронізованих колекцій класу java.util.Collections та колекцій Java Concurrent API.
33. Опишіть реалізації інтерфейсів BlockingQueue<E>, BlockingDequeue<E> () та їх призначення. Чим забезпечується ефективність їх роботи?
34. Опишіть код програми, яка демонструє використання блокуючої черги.
35. Опишіть реалізації інтерфейсу ConcurrentMap<K,V>. Яким чином реалізується потокобезпечність та ефективність роботи ConcurrentHashMap?
36. Опишіть код програми, яка демонструє використання ConcurrentHashMap.
37. Опишіть принцип роботи класів CopyOnWriteArrayList<E> та CopyOnWriteArraySet<E>. Яким чином реалізується їх потокобезпечність та ефективність роботи?
38. Опишіть код програми, яка демонструє використання CopyOnWriteArrayList<E>.
39. Надайте характеристику *Fork/Join* фреймворку. Які задачі він спроможний вирішувати? У якій версії Java він з'явився?
40. Поясніть алгоритм крадіжки роботи (work-stealing algorithm), який використовує *Fork/Join* фреймворк.
41. Надайте характеристику основним інструментам *Fork/Join* фреймворку, що забезпечують виконання у пулі потоків завдань робочими потоками, та їх основним методам.
42. Опишіть код програми, яка демонструє використання засобів *Fork/Join* фреймворку для розрахунку рекурсивного завдання.

4. Література

1. Блинов И.Н., Романчик В.С. Java. Методы программирования : уч.-мет. пособие - Минск : издательство "Четыре четверти", 2013. -896 с.
2. Гетц Б., Пайерлс Т., Блох Дж., Боубер Дж., Холмс Д., Ли Д. Java Concurrency на практике. — СПб.: Питер, 2020. — 464 с.
3. Concurrency. The Java™ Tutorials. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html (дата звернення 30.08.2020).
4. Обзор java.util.concurrent.*. URL: https://habr.com/ru/company/luxoft/blog/157273 (дата звернення 30.08.2020).
5. Справочник по синхронизаторам java.util.concurrent.*. URL: https://habr.com/ru/post/277669 (дата звернення 30.08.2020).

6. Java Compare and Swap Example – CAS Algorithm. URL: https://how-todoinjava.com/java/multi-threading/compare-and-swap-cas-algorithm/ (дата звернення 30.08.2020).
7. Руководство по Fork/Join Framework в Java. URL: https://www.codeflow.site/ru/article/java-fork-join (дата звернення 30.08.2020).