Programowanie współbieżne
Laboratorium nr 13

1 Synchronizatory

„Synchronizator [1] to dowolny obiekt koordynujący przepływ sterowania wątków na podstawie jego stanu."

„Wszystkie synchronizatory [1] maja pewne wspólne właściwości strukturalne. Hermetyzują stan, który określa, czy wątek przybywający do synchronizatora może przejść czy raczej musi zaczekać na wykonanie zadania; zawierają metody modyfikujące stan i sposoby wydajnego oczekiwania na zmianę stanu przez kod."

2 CountDownLatch

„Obiekt klasy CountDownLatch [2] zmusza wątki do oczekiwania, aż licznik dojdzie do zera. Zatrzask ten jest jednorazowego użytku, to znaczy, jeśli licznik dojdzie do zera, nie można go zwiększyć."

„Klasa CountDownLatch (ang. latch - zatrzask) [4] służy do synchronizowania jednego bądź wielu zadań przez wymuszanie oczekiwania na zakończenie zestawu operacji wykonywanych przez inne zadania."

Klasa java.util.concurrent.CountDownLatch (wybrane metody):

- public CountDownLatch(int count) - konstruktor tworzący zatrzask z parametrem określającym ile razy metoda countDown() musi zostać wywołana zanim wątki będą mogły "przejść przez metodę" await(),
- public void await(), public boolean await(long timeout, TimeUnit unit) - wywołanie metody powoduje, że bieżący wątek czeka na osiągnięcie przez zatrzask wartości zero; istnieje także przeciążona wersja tej metody posiadająca dodatkowe ograniczenie czasowe.,
- public void countDown() - metoda dekrementuje wartość wewnętrznego licznika zatrzasku, jeżeli nową wartością jest zero wszystkie oczekujące wątki są odblokowane.

Przykład: Egzaminator (klasa Examiner) przeprowadza egzamin na grupie studentów (klasa Student). Po przygotowaniu pytań przez egzaminatora wszyscy studenci równocześnie rozpoczynają egzamin. Każdy student może w dowolnej chwili zakończyć egzamin. Nie jest podany maksymalny czas trwania egzaminu. Po zakończeniu egzaminu przez wszystkich studentów egzaminator sprawdza prace. Przykład opracowano na podstawie: Dokumentacja JavaDoc 1.6

```
import java.util.Map;
import java.util.Random;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

class Student extends Thread {
    final private Examiner ex;

    Student(Examiner ex) {
        this.ex = ex;
    }

    @Override
    public void run() {
        try {
            String question = ex.getQuestion(this);
            String answer = exam(question);
            ex.returnAnswer(this, answer);
        } catch (InterruptedException e) {
            System.err.printf("Student %2d: the exam was cancelled%n", this.getId());
        }
    }
```

```
    private String exam(String question) throws InterruptedException {
        TimeUnit.SECONDS.sleep(new Random(hashCode() + getId()).nextInt(10));
        return "" + this.getName();
    }
}

public class Examiner {
    final private static int STUDENT_NUMER = 4;
    final private static CountDownLatch beginExam = new CountDownLatch(1);
    final private static CountDownLatch finishExam = new CountDownLatch(STUDENT_NUMER);
    final private Map<Long, String> answers = new ConcurrentHashMap<Long, String>();

    String getQuestion(Student s) throws InterruptedException {
        Examiner.beginExam.await();
        System.err.printf("Student %2d began the exam at %12d%n", s.getId(), System.nanoTime());
        return "What is your name?";
    }

    void returnAnswer(Student s, String answer) {
        System.err.printf("Student %2d finished the exam at %12d%n", s.getId(), System.nanoTime())↩
            ;
        this.answers.put(s.getId(), answer);
        Examiner.finishExam.countDown();
    }

    private void beginExam() {
        Examiner.beginExam.countDown();
        System.err.printf("Examiner began the exam at %12d%n", System.nanoTime());
    }

    private void finishExam() throws InterruptedException {
        Examiner.finishExam.await();
        System.err.printf("Examiner finished the exam at %12d%n", System.nanoTime());
        System.err.println("Answers: " + answers);
    }

    public static void main(String[] args) {
        try {
            Examiner ex = new Examiner();
            for (int i = 0; i < STUDENT_NUMER; i++)
                new Student(ex).start();
            ex.beginExam();
            ex.finishExam();
        } catch (InterruptedException e) {
            System.err.printf("Examiner: the exam was cancelled%n");
        }
    }
}
```

Listing 1: Przykład użycia klasy CountDownLatch {src/Examiner.java}

3 CyclicBarier

„Klasa CyclicBarier [4] wykorzystywana jest w sytuacjach, w których potrzeba utworzyć grupę zadań, uruchomić je współbieżnie, a potem oczekiwać na ich zakończenie warunkujące przejście do następnego etapu programu (coś w rodzaju join()). Wszystkie równolegle wykonywane zadania są wyrównywane na barierze uniemożliwiającej przedwczesne kontynuowanie procesów programu. Model klasy bardzo przypomina klasę CountDownLatch, z tym że obiekty klasy CountDownLatch to 'jednorazówki', a egzemplarze CyclicBarier można swobodnie regenerować."

Klasa java.util.concurrent.CyclicBarrier (wybrane metody):

- public CyclicBarrier(int parties) - konstruktor tworzący cykliczną barierę, jako parametr podaje się liczbę wątków konieczną do otworzenia bariery,
- public CyclicBarrier(int parties, Runnable barrierAction) - konstruktor tworzący cykliczną barierę, jako pierwszy parametr podaje się liczbę wątków konieczną do otworzenia bariery, w drugim parametrze można przekazać akcję bariery wykonywaną gdy wszystkie wątki dotrą do bariery, ale przed zwolnieniem blokady (akcja wykonywana jest przez watek, który dotarł jako ostatni),
- public int await() - wywołanie metody powoduje, że bieżący wątek czeka aż liczba wątków, które wywołają tą metodę osiągnie wartość podaną w konstruktorze; istnieje także przeciążona wersja tej metody posiadająca dodatkowe ograniczenie czasowe.

Przykład: Firma (klasa Company) składa się z kilku działów (klasa Department). Każdy dział na koniec miesiąca wylicza swoją sumaryczną sprzedaż (atrybut sales). Miesięczna premia, w każdym z działów, wyliczana jest jako suma:

- 10% z sumarycznej sprzedaży danego działu,
- 10% z sumarycznej sprzedaży wszystkich działów podzielonej przez liczbę działów.

```
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;

class Department extends Thread {

    private volatile double sales;
    final private Random r;
    final private Company c;
    final private int deptNo;

    Department(Company c, int deptNo) {
        this.c = c;
        this.deptNo = deptNo;
        r = new Random(hashCode() + getId());
    }

    double getDepartmentSales() {
        return sales;
    }

    @Override
    public void run() {
        try {
            while (true) {
                calculateSales();
                System.err.printf("Dept. No. %1d Sales : %4.0f (%15d)%n", this.deptNo, sales, ↩
                    System.nanoTime());
                double globalPremiumPart = c.getGlobalPremiumPart();
                System.err.printf("Dept. No. %1d Total premium: %4.0f (%15d)%n", this.deptNo, (0.1 ↩
                    * sales + globalPremiumPart), System.nanoTime());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

    void calculateSales() throws InterruptedException {
```

```
        TimeUnit.SECONDS.sleep(r.nextInt(10));
        sales = r.nextInt(1000);
    }
}

public class Company implements Runnable {

    final private static int NUMBER_OF_DEPARMENTS = 3;
    final private Department[] departments = new Department[NUMBER_OF_DEPARMENTS];
    final private CyclicBarrier barrier;
    private double globalPremiumPart;
    private int monthNo;

    public static void main(String[] args) {
        new Company();
    }

    Company() {
        barrier = new CyclicBarrier(NUMBER_OF_DEPARMENTS, this);
        for (int i = 0; i < departments.length; i++) {
            departments[i] = new Department(this, i);
            departments[i].start();
        }
    }

    @Override
    public void run() {
        double globalSales = 0;
        for (int i = 0; i < departments.length; i++) {
            double tmp = departments[i].getDepartmentSales();
            System.err.printf("Dept. No. %1d %4.0f%n", i, tmp);
            globalSales += tmp;
        }

        globalPremiumPart = 0.1 * globalSales / NUMBER_OF_DEPARMENTS;
        System.err.printf("Month No. %1d Global part of the premium: %4.0f%n", monthNo++, ↩
            globalPremiumPart);
    }

    double getGlobalPremiumPart() throws InterruptedException, BrokenBarrierException {
        barrier.await();
        return globalPremiumPart;
    }

}
```

Listing 2: Przykład użycia klasy CyclicBarier {src/Company.java}

4 Phaser

Klasa Phaser (można to przetłumaczyć jako fazer) [javadoc] jest barierą synchronizacyjną wielokrotnego użytku, o funkcjonalności zbliżonej do CyclicBarrier oraz CountDownLatch, ale umożliwiająca bardziej elastyczne użycie. Szczegółowe omówienie można znaleźć w artykule "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization" J. Shirako D. M. Peixotto V. Sarkar W. N. Scherer III http://www.cs.rice.edu/ vs3/PDF/SPSS08-phasers.pdf

Podstawowe cechy klasy Phaser [javadoc]:

- Rejestrowanie – liczba zarejestrowanych uczestników może zmieniać się w czasie. Początkowa liczba uczestników może zostać ustawiana w konstruktorze. Rejestracja jest możliwa poprzez wywołanie

metod: register() i bulkRegister (int). Natomiast wyrejestrowanie jest możliwe poprzez wywołanie metody arriveAndDeregister().

- Synchronizacja – podobnie jak dla klasa CyclicBarrier klasa Phaser umożliwia cykliczne oczekiwanie. Z każdą generacją powiązany jest numer fazy (ang. phase number). Istnieją dwa rodzaj metod, które mogą być wywoływane przez zarejestrowanych uczestników.
  - Przybycie (ang. Arrival) – do jego zgłaszania służą metody arrive() i arriveAndDeregister(). Nie są to metody blokujące i zwracają numer fazy przybycia (ang. arrival phase number). Kiedy ostatni uczestnik przybędzie wywoływana jest opcjonalna akcja (poprzez nadpisanie metody onAdvance(int, int)), a następnie faza jest zwiększana.
  - Oczekiwanie (ang. Waiting) – wywołanie metody awaitAdvance(int) powoduje, że nastąpi oczekiwanie na momentu gdy numer fazy zmieni się na inny niż podany w argumencie.
- Zakończenie (ang. Termination) – w tym stanie wszystkie metody służące do synchronizacji kończą się bez oczekiwania, natomiast próba rejestracji nie przynosi efektu. Stan ten jest wyzwalany gdy metoda onAdvance() zwróci true (domyślnie gdy liczba zarejestrowanych uczestników spadnie do zera).

  ```
  Phaser phaser = new Phaser() {
     protected boolean onAdvance(int phase, int parties) { return false; }
  }
  ```

  Istnieje także metoda forceTermination().
- Warstwy (ang. Tiering) – obiekt tej klasy może być zorganizowany w warstwy (np. w postaci drzewa), w celu uniknięcia problemów występujących w sytuacji gdy istnieje duża liczba uczestników.
- Monitorowanie – stan obiektu tej klasy może być odczytywany przez dowolny obiekt, służą do tego metody: getRegisteredParties(), getArrivedParties(), getUnarrivedParties(), getPhase(), toString().

Przykład dotyczy przechodzenia kandydatów przez proces rekrutacyjny składający się testów.

```
import java.util.Random;
import java.util.concurrent.Phaser;

class Candidate extends Thread{

    private int id;
    private Recruitment recruitment;
    private Random random = new Random();

    Candidate(int id, Recruitment recruitment) {
        this.id = id;
        this.recruitment = recruitment;
    }

    void applyForNextStep(){
        recruitment.applyNextStep();
    }

    @Override
    public void run(){
        recruitment.applyNextStep();
    }
}

public class Recruitment {

    Phaser selection = new Phaser();

    void applyNextStep(){
```

```
        System.out.println(Thread.currentThread().getId() + " " + selection.getArrivedParties() + ↩
            " (+) ");
        System.out.println(Thread.currentThread().getId() + " " + selection.getUnarrivedParties() ↩
            + " (-) ");
        selection.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getId() + " " + selection.getArrivedParties() + ↩
            " (+) ");
        System.out.println(Thread.currentThread().getId() + " " + selection.getUnarrivedParties() ↩
            + " (-) ");
    }

    void startNextStep(int candidatesNumber){
        selection.register();
        selection.bulkRegister(candidatesNumber);
        selection.arriveAndAwaitAdvance();
        System.out.println(Thread.currentThread().getId() + " " + selection.getArrivedParties() + ↩
            " (+) ");
        System.out.println(Thread.currentThread().getId() + " " + selection.getUnarrivedParties() ↩
            + " (-) ");
    }

    public static void main(String[] args) {
        Recruitment recruitment = new Recruitment();
        int candidatesNumber = 5;
        Candidate [] candidates = new Candidate[candidatesNumber];
        for(int i = 0; i < candidates.length; i++){
            candidates[i] = new Candidate(i, recruitment);
            candidates[i].start();
        }

        recruitment.startNextStep(candidatesNumber);
    }
}
```

Listing 3: Przykład użycia klasy Phaser {src/Recruitment.java}

5 Exchanger

„Obiekty klasy Exchanger [2] znajdują zastosowanie, gdy dwa wątki działają na dwóch egzemplarzach jednego bufora danych. Z reguły jeden z nich zapełnia bufor, a drugi pobiera te dane. Kiedy oba watki są gotowe, wymieniają się buforami."

„Klasa Exchanger [4] implementuje barierę, która wymienia obiekty pomiędzy dwoma zadaniami. Kiedy zadanie dociera do takiej bariery, posiada pewien obiekt, a kiedy opuszcza barierę, jego miejsce zajmuje inny obiekt, będący w posiadaniu innego zadania. Bariery wymiany są wykorzystywane, kiedy jedno z zadań wytwarza obiekty kosztowne w produkcji, a inne zadanie konsumuje te obiekty; typowo dochodzi wtedy niejako do recyclingu obiektów, co zmniejsza koszt wytwarzania."

Klasa java.util.concurrent.Exchanger<V>(wybrane metody):

- public Exchanger() - konstruktor bezargumentowy,
- public V exchange(V x) - wywołanie metody spowoduje, że aktualny wątek rozpocznie czekanie na inny wątek, po przybyciu drugiego wątku wątki wymienią się obiektami; istnieje także przeciążona wersja tej metody posiadająca dodatkowe ograniczenie czasowe.

Przykład dotyczy straży pożarnej (klasa FireBrigade), w której pracują strażacy (abstrakcyjna klasa FireFighter). Strażacy dzielą się na dwie kategorie:

- odpowiedzialnych za napełnianie wiaderek (klasa FireFighterFill),
- odpowiedzialnych za gaszenie pożaru czyli opróżnianie wiaderek (klasa FireFighterEmpty).

Podczas gaszenia pożaru strażacy pracują w parach wymieniając się wiaderkami (klasa Bucket). Strażak napełniający wiaderko podaje pełne wiaderko strażakowi gaszącemu pożar. Natomiast strażak gaszący pożar podaje puste wiaderko strażakowi napełniającemu wiaderka.

```
import java.util.Random;
import java.util.concurrent.*;

class Bucket {
    volatile boolean empty;
    private int number;

    Bucket(int number) {
        this.number = number;
    }

    @Override
    public String toString() {
        return "Bucket" + number + ": " + (empty ? "empty" : "filled");
    }
}

abstract class FireFighter extends Thread {
    protected final Random r;
    private final Exchanger<Bucket> exchanger;
    private Bucket b;

    FireFighter(Exchanger<Bucket> exchanger, Bucket initialBucket) {
        r = new Random(this.hashCode());
        this.exchanger = exchanger;
        this.b = initialBucket;
    }

    abstract void work(Bucket b) throws InterruptedException;

    @Override
    public void run() {
        try {
            while (true) {
                System.err.printf("%-17s start the work (%12d)%n", this, System.nanoTime());
                work(b);
                System.err.printf("%-17s end the work (%12d)%n", this, System.nanoTime());
                b = exchanger.exchange(b);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }
}

class FireFighterFill extends FireFighter {

    FireFighterFill(Exchanger<Bucket> exchanger, Bucket initialBucket) {
        super(exchanger, initialBucket);
    }

    @Override
```

```
    void work(Bucket b) throws InterruptedException {
        TimeUnit.SECONDS.sleep(r.nextInt(10));
        b.empty = false;
        System.err.println(b);
    }
}

class FireFighterEmpty extends FireFighter {

    FireFighterEmpty(Exchanger<Bucket> exchanger, Bucket initialBucket) {
        super(exchanger, initialBucket);
    }

    @Override
    void work(Bucket b) throws InterruptedException {
        TimeUnit.SECONDS.sleep(r.nextInt(10));
        b.empty = true;
        System.err.println(b);
    }
}

public class FireBrigade {

    private static Exchanger<Bucket> exchanger = new Exchanger<Bucket>();
    private static Bucket b0 = new Bucket(0);
    private static Bucket b1 = new Bucket(1);

    public static void main(String[] args) {
        FireFighter f0 = new FireFighterFill(exchanger, b0);
        f0.start();
        FireFighter f1 = new FireFighterEmpty(exchanger, b1);
        f1.start();
    }
}
```

Listing 4: Przykład użycia klasy Exchanger {src/FireBrigade.java}

6 Semaphore

„Z założenia semafor [2] służy do zarządzania pewną liczbą zezwoleń (ang. permit). Aby przejść obok semafora, wątek próbuje uzyskać zezwolenie, wywołując w tym celu metodę acquire. Liczba dostępnych zezwoleń jest ograniczona, co pozwala na kontrolę liczby wątków, które mogą przejść dalej. Inne wątki mogą wydawać zezwolenia za pomocą metody release. Nie istnieją żadne obiekty zezwoleń. Semafor przechowuje tylko licznik zezwoleń."

Klasa java.util.concurrent.Semaphore (wybrane metody):

- public Semaphore(int permits) - tworzy semafor z podaną liczbą zezwoleń,
- public void acquire() - próbuje uzyskać pozwolenie, jeżeli żadne pozwolenie nie jest dostępne blokuje się, istnieje także przeciążona wersja tej metody pozwalająca uzyskać więcej niż jedno pozwolenie,
- public int availablePermits() - zwraca ilość aktualnie dostępnych pozwoleń,
- public void release() - zwraca pozwolenie, istnieje także przeciążona wersja tej metody pozwalająca zwrócić więcej niż jedno pozwolenie.

Przykład dotyczy wypożyczalni samochodów (klasa Rental<Car>). Typ ogólny Rental<V> może zostać wykorzystany także z dowolną inną klasą. Przykład opracowano na podstawie: Dokumentacja JavaDoc 1.6 oraz Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.concurrent.Semaphore;

final class Rental<V> {

    private static class Entry<E> {
        boolean used;
        final E object;

        Entry(E object) {
            this.object = object;
        }
    }

    final private Semaphore permits;
    final private Collection<Entry<V>> entries = new HashSet<Entry<V>>();

    Rental(Collection<V> collection) {
        if (collection == null)
            throw new NullPointerException();
        for (V e : collection) {
            if (e == null)
                throw new NullPointerException();
            entries.add(new Entry<V>(e));
        }
        permits = new Semaphore(entries.size());
    }

    int getAvailable() {
        return permits.availablePermits();
    }

    public V rent() throws InterruptedException {
        permits.acquire();
        return getNext();
    }

    private synchronized V getNext() {
        for (Entry<V> e : entries)
            if (!e.used) {
                e.used = true;
                return e.object;
            }
        return null;
    }

    public void giveBack(V object) {
        if (tryReturn(object))
            permits.release();
    }

    private synchronized boolean tryReturn(V c) {
        for (Entry<V> e : entries)
            if (e.object == c) {
                if (!e.used)
                    return false;
                e.used = false;
                return true;
            }
```

```
        return false;
    }
}

class Car {
}

public class TestCarRental {
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Car> cars = new ArrayList<Car>();
        for (int i = 0; i < 4; i++)
            cars.add(new Car());

        Rental<Car> carRental = new Rental<Car>(cars);
        System.out.println("Available cars: " + carRental.getAvailable());

        Car c = carRental.rent();
        System.out.println("Rented car : " + c);
        System.out.println("Available cars: " + carRental.getAvailable());

        Car c2 = carRental.rent();
        System.out.println("Rented car : " + c2);
        System.out.println("Available cars: " + carRental.getAvailable());

        carRental.giveBack(c2);
        System.out.println("Available cars: " + carRental.getAvailable());

        carRental.giveBack(c2);
        System.out.println("Available cars: " + carRental.getAvailable());
        Car tmp = new Car();
        carRental.giveBack(tmp);
        System.out.println("Returned car : " + tmp);
        System.out.println("Available cars: " + carRental.getAvailable());
    }
}
```

Listing 5: Przykład użycia klasy Semaphore {src/TestCarRental.java}

7 SynchronousQueue

„Kolejka synchroniczna [2] to mechanizm dobierania w pary wątków producenta i konsumenta. Kiedy jeden wątek wywoła metodę put na rzecz obiektu typu SynchronousQueue, zostaje on zablokowany dopóki inny watek nie wywoła metody take i odwrotnie. W przeciwieństwie do klasy Exchanger, w tym przypadku dane są przesyłane tylko w jednym kierunku - od producenta do konsumenta. Mimo, iż klasa SynchronousQueue implementuje interfejs BlockingQueue, nie jest z założenia kolejką. Nie zawiera żadnych elementów, czyli jej metoda size zawsze zwraca wartość 0."

Klasa java.util.concurrent.SynchronousQueue<E>

extends AbstractQueue<E>

implements BlockingQueue<E>, Serializable.

Informacje na temat kolejki blokujące public interface BlockingQueue<E> extends Queue<E> można znaleźć na poprzednim wykładzie.

Przykład: Przykład dotyczy sztafety (klasa Relay). W sztafecie mamy czterech biegaczy (klasa Runner), którzy podczas zmiany (atrybut change typu ArrayList<SynchronousQueue<Baton«) przekazują między sobą pałeczkę sztafetową (klasa Baton).

Sztafeta

```
Change (zmiana) 0 1 2
Runner (biegacz) 0 1 2 3
```

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.concurrent.Semaphore;

final class Rental<V> {

    private static class Entry<E> {
        boolean used;
        final E object;

        Entry(E object) {
            this.object = object;
        }
    }

    final private Semaphore permits;
    final private Collection<Entry<V>> entries = new HashSet<Entry<V>>();

    Rental(Collection<V> collection) {
        if (collection == null)
            throw new NullPointerException();
        for (V e : collection) {
            if (e == null)
                throw new NullPointerException();
            entries.add(new Entry<V>(e));
        }
        permits = new Semaphore(entries.size());
    }

    int getAvailable() {
        return permits.availablePermits();
    }

    public V rent() throws InterruptedException {
        permits.acquire();
        return getNext();
    }

    private synchronized V getNext() {
        for (Entry<V> e : entries)
            if (!e.used) {
                e.used = true;
                return e.object;
            }
        return null;
    }

    public void giveBack(V object) {
        if (tryReturn(object))
            permits.release();
    }

    private synchronized boolean tryReturn(V c) {
        for (Entry<V> e : entries)
            if (e.object == c) {
                if (!e.used)
                    return false;
                e.used = false;
                return true;
            }
```

```
        return false;
    }
}

class Car {
}

public class TestCarRental {
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Car> cars = new ArrayList<Car>();
        for (int i = 0; i < 4; i++)
            cars.add(new Car());

        Rental<Car> carRental = new Rental<Car>(cars);
        System.out.println("Available cars: " + carRental.getAvailable());

        Car c = carRental.rent();
        System.out.println("Rented car : " + c);
        System.out.println("Available cars: " + carRental.getAvailable());

        Car c2 = carRental.rent();
        System.out.println("Rented car : " + c2);
        System.out.println("Available cars: " + carRental.getAvailable());

        carRental.giveBack(c2);
        System.out.println("Available cars: " + carRental.getAvailable());

        carRental.giveBack(c2);
        System.out.println("Available cars: " + carRental.getAvailable());
        Car tmp = new Car();
        carRental.giveBack(tmp);
        System.out.println("Returned car : " + tmp);
        System.out.println("Available cars: " + carRental.getAvailable());
    }
}
```

Listing 6: Przykład użycia klasy SynchronousQueue {src/TestCarRental.java}

Literatura

[1] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D., Java Współbieżność dla praktyków, Helion 2007

[2] Horstmann C.S., Cornell G., Java Podstawy, Helion, Wyd. VIII, 2009

[3] Horstmann C.S., Cornell G., Java Techniki zaawansowane, Helion, Wyd. VIII, 2009

[4] Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006.

[5] Bloch J.: Java Efektywne programowanie, Wyd. II, Helion, 2009.

[6] Brackeen D., B. Barker, L. Vanhelsuwe: Java Tworzenie gier, Helion, 2004.

[7] Silberschatz A., Galvin P. B., Gagne G.: Podstawy systemów operacyjnych, WNT, 2005

[8] Dokumentacja JavaDoc 1.6 http://java.oracle.com

[9] Dokumentacja JavaDoc 1.7 http://java.oracle.com

[10] The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf