Laboratorium 7

Wstęp

Jednym z podstawowych własności Javy jest wielowątkowość. Wiąże się to z możliwością współbieżnego wykonywania różnych operacji w ramach pojedynczej wirtualnej maszyny Javy (JVM).

Wielowątkowość realizowana jest poprzez umieszczenie kodu realizowanego współbieżnie w metodzie **run** klasy rozszerzającej klasę **Thread** lub implementującej interfejs **Runnable**

Przykład 1

```
class MyThread1 extends Thread {
    @Override
    public void run(){
        //kod realizowany współbieżnie
    }
}
```

Przykład 2

```
class MyThread2 implements Runnable {
    @Override
    public void run(){
        //kod realizowany współbieżnie
    }
}
```

Stworzenie klasy nie jest równoznaczne z wykorzystaniem wielowątkowości. Uzyskuje się ją dopiero poprzez inicjalizację wątku. Odpowiednio:

Do przykład 1

```
public static void main(String[] args) {
    Thread t = new MyThread1();
    t.start();
}
```

Do przykład 2

```
public static void main(String[] args) {
    MyThread2 myThread = new MyThread2();
    Thread t = new Thread(myThread);
}
```

Należy równocześnie pamiętać iż program główny również wykonywany jest w osobnym wątku.

Jednym z zagadnień związanych z operacją na wątkach jest synchronizacja wątków. Synchronizacja wątków to proces, w którym wymuszamy by wątki oczekują na swoje zakończenie, aby program

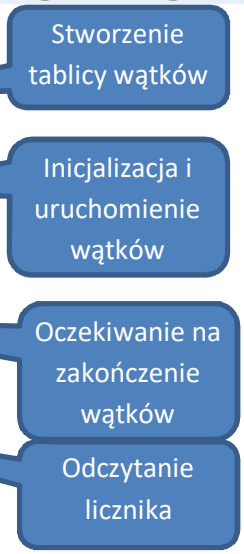
główny mógł kontynuować swoje działanie. Powyższy problem można zrealizować korzystając z metody **join()**

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new MyThread1(0);
        Thread t2 = new MyThread1(1);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Wątki " + t1.getName() + " i " +
                t2.getName() + " zakończone");
    }
}

class MyThread1 extends Thread {
    int id;
    public MyThread1(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        try {
            Thread.sleep((long) (300 * Math.random()));
        } catch (InterruptedException ex) { }
        System.out.println("Pozdrowienia z wątku: " + this.getName() +
                " nr wątku: "+ id);
    }
}
```

Osobnym zagadnieniem jest zapewnienie spójności danych i rozwiązanie problemu dostępu do danych współdzielonych. Aby zapewnić ich spójność Java oferuje słowo kluczowe **synchronized** . Wywołanie metody lub fragmentu kodu oznaczonego tym słowem powoduje ustawienie semafora na danym obiekcie, co jednocześnie blokuje dostęp innym wątkom do wykonania tzw. sekcji krytycznej kodu na tym obiekcie (sekcja krytyczna to fragment kodu wykonywany tylko przez jeden wątek).

Przedstawia to poniższy fragment:

```
public class Main1 {

    public static void main(String[] args) throws InterruptedException {
        System.out.println(Watek.licznik.get());
        int ileWatkow = 10;
        Watek watki[] = new Watek[ileWatkow];
        for (int i = 0; i < ileWatkow; i++) {
            watki[i] = new Watek(i + 1);
            watki[i].start();
        }
        for (int i = 0; i < ileWatkow; i++) {
            watki[i].join();
        }
        System.out.println(Watek.licznik.get());
    }
}
```



Klasa Watek

```
class Watek extends Thread {
    private int numer;
    static Licznik licznik = new Licznik();
    Watek(int id) {
        numer = id;
    }
    @Override
    public void run() {
        System.out.println("Watek: " + numer);
        for (int i = 0; i < 100000; ++i) {
            licznik.increment();
        }

    }
}
```

Klasa Licznik

```
class Licznik {
    int licznik;
    Licznik(){ licznik = 0; }
    synchronized void increment(){
        licznik++;
    }
    int get(){
        return licznik;
    }
}
```

W powyższym przykładzie wykorzystanie słówka **synchronized** gwarantuje nam że proces inkrementacji licznika odbywa się w sekcji krytycznej, tzn że inkrementacje może zrealizować tylko

jeden wątek. Jeśli określony wątek nie może zostać w danej chwili wykonany zasypia o oczekuje na możliwość wejścia do sekcji krytycznej.

W pewnych przypadkach może się zdarzyć, że dany wątek nie może wykonywać swoich operacji, gdyż występuje np. brak danych na których mógłby wykonywać operacje. Wówczas taki wątek powinien zasnąć. W tym celu wykorzystuje się metodę **wait(),** która wprowadza wątek w stan uśpienia, czyli oddaje swój czas procesora innym wątkom. Obudzenie dokonuje się po wywołaniu polecenia **notify()** lub **notifyAll().**

Przykładem powyższego problemu może być zagadnienie produenta i konsumenta. Tzn. producent może wytworzyć towar i wrzucić go do magazynu, ale tylko gdy magazyn jest pusty. Konsument natomiast pobiera towar z magazynu tylko jeśli on tam jest, a pobranie towaru z magazynu zwalnia miejsce na nowy towar. Powyższe przedstawia poniższy przykład:

Metoda uruchamieniowa:

```
public static void main(String[] args) throws InterruptedException {
    Magazyn m = new Magazyn();
    Thread t1 = new Thread(new Producent(m));
    Thread t2 = new Thread(new Konsument(m));
    t1.start();
    t2.start();
    Thread.sleep(10000);
    t1.interrupt();
    t2.interrupt();
    System.out.println("Koniec");
}
```

Czekamy 10 tyś milisekund

Wywołujemy przerwanie wątków. UWAGA działanie zależne od obsługi wyjątku!!!

Klasa Producent

```
class Producent implements Runnable {

    Magazyn m;
    int towarId;

    public Producent(Magazyn m) {
        this.m = m;
    }

    @Override
    public void run() {
        try {
            while (true) {
                Thread.sleep((long) (500*Math.random()));
                m.set("Towar id:" + towarId);
                towarId++;
            }
        } catch (InterruptedException ex) { }
    }
}
```

Producent w nieskończonej pętli dodaje towar. Jeśli nie może dodać to czeka

UWAGA wyjątek przechwycony po za pętlą, inaczej program nigdy by się nie skończył

Klasa Konsument

```
class Konsument implements Runnable {

    Magazyn m;

    public Konsument(Magazyn m) {
        this.m = m;
    }

    @Override
    public void run() {
        try {
            while (true) {
                Thread.sleep((long) (500*Math.random()));
                String s = m.get();
                System.out.println("Pobrałem towar: " + s);
            }
        } catch (InterruptedException ex) { }
    }
}
```

Pobranie towaru z magazynu, jeśli jest gotowy. Inaczej czekamy

UWAGA wyjątek przechwycony po za pętl
inaczej program nigdy by się nie skończył

Klasa Magazyn

```
class Magazyn {

    String towar;
    boolean jestTowar = false;

    synchronized void set(String towar) throws InterruptedException {
        while (jestTowar) {
            this.wait();
        }
        this.towar = towar;
        jestTowar = true;
        notifyAll();
    }

    synchronized String get() throws InterruptedException {
        while (!jestTowar) {
            this.wait();
        }
        jestTowar = false;
        notifyAll();
        return towar;
    }
}
```

Sprawdzamy czy towar już jest w magazynie, jeśli tak to czekamy

Jeśli towaru brak to wkładamy towar i budzimy wątki

Sprawdzamy czy towar już jest w magazynie, jeśli nie to czekamy

Jeśli towar jest to go pobieramy i budzimy wątki

Zadania

1. Stwórz dwa wątki poprzez implementacje po klasie Runnable. Każdy z wątków ma w pętli wypisywać liczby. Wątek 1 ma wypisywać liczby z przedziału 0 do 20, a wątek 2 liczby z przedziału 100 do 120. Wątek główny powinien poczekać na zakończenie wątków dodatkowych i zakończyć działanie.
2. Zmodyfikuj powyższy program tak, aby najpierw wypisały się liczby z wątku 2, a potem liczby z wątku 1. Dla uproszczenia możesz przekazać referencje dla tworzonych wątków miedzy sobą
3. Zaimplementuj przykład z licznikiem. Co się dzieje i dlaczego program nie działa jeśli nie skorzystamy z synchronizacji?
4. Zaimplementuj problem Producenta-Konsumenta
5. Napisz program z interfejsem graficznym działający w dwóch wątkach.
    a. Aplikacja z GUI działa w jednym wątku (tak jest zawsze)
    b. Stwórz drugi wątek w którym będzie wyznaczana wartość liczby PI za pomocą metody MonteCarlo – procedura ta odbywa się w nieskończonej pętli
    c. Wątek z aplikacją obliczeniową powinien wysyłać wystawiać do współdzielonego z interfejsem graficznym bufora aktualną wartością liczby PI co 10.000 iteracji
    d. W interfejsie graficznym aplikacja co 0.5s powinna weryfikować aktualny stan bufora i wyświetlać warość na ekranie (przy zmianie wartości zmień kolor czcionki wyświetlającej napis). Do realizacji zadania skorzystaj z Timera (jeśli robisz aplikację w technologii Swing jest to klasa javax.swing.Timer), który co 0.5s sprawdza i wyświetla aktualny stan liczby PI
    Obsługa timera:
    ```
    timer = new Timer(500, (e) -> {
        metodaObsługaZdarzenia();
    });
    ```
    Timer posiada metody start() – uruchomienie timera, stop() – zatrzymanie timera. Jego działanie sprowadza się do uruchomienia metody *metodaObsługiZdarzenia()* co 500ms.
    e. W metodzie obsługi timera powinna być pobierana aktualna wartość liczby PI, a następnie wartość ta powinna być wyświetlana.

6. Napisz program realizujący tzw problem fryzjera:
W zakładzie fryzjerskim znajduje się tylko jeden fotel oraz poczekalnia z określoną liczbą miejsc. Pracuje w nim jeden fryzjer. Do zakładu przychodzą klienci, którzy chcą się ostrzyc. Zaprogramuj funkcjonowanie zakładu fryzjerskiego zachowując następujące własności:
    - Kiedy fryzjer kończy strzyżenie klienta, klient wychodzi a fryzjer zagląda do poczekalni czy są kolejni klienci. Jeśli jakiś klient czeka w poczekalni wówczas zaprasza go na fotel i zaczyna strzyżenie. Jeśli poczekalnia jest pusta, wówczas fryzjer ucina sobie drzemkę na fotelu.
    - Klient, który przychodzi do zakładu i sprawdza, co robi fryzjer. Jeśli fryzjer strzyże, wówczas klient idzie do poczekalni, i jeśli jest wolne miejsce to je zajmuje i czeka. Jeśli nie ma wolnego miejsca, wówczas klient rezygnuje. Jeśli fryzjer śpi, to budzi go.