4. Synchronizacja wątków

Aplikacje wielowątkowe często korzystają ze wspólnych zasobów. Dostęp do nich może powodować nieprzewidywane zachowania w działaniu aplikacji (błędne dane, modyfikacja zmiennych itp.). W programowaniu współbieżnym i równoległy, a także w programowaniu aplikacji w środowisku rozproszonym stosuje się wiele technik umożliwiających poprawną obsługę współdzielonych zasobów.

Przykładowa aplikacja

Poniżej znajduje się aplikacja odwołująca się do wspólnego zasobu klasy `Konto`.

```
public class Konto {
    private int stan;

    public Konto(int kwota) {
        this.stan = kwota;
    }

    public boolean zmienStan(int kwota) {
        int nowyStan = this.stan + kwota;

        try {
            Thread.sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {
           System.out.println("Przerwana transakcja.");
        }

        if (nowyStan < 0) {
            return false;
        } else {
            this.stan = nowyStan;
            return true;
        }
    }

    public String toString(){
        return ("Aktualny stan konta: " + this.stan);
    }

}
```

Do wspólnego zasobu dostęp będą miały obiekty klasy `Klient` implementujące interfejs `Runnable`.

```
public class Klient implements Runnable {
    private Konto konto;
    private int[] operacje;

    public Klient(Konto konto, int[] operacje) {
        this.konto = konto;
        this.operacje = operacje;
    }

    public void run() {

        for(int i = 0; i < operacje.length; i++) {

            konto.zmienStan(operacje[i]);

            System.out.println("Operację wykonuje: " +
```

```
            Thread.currentThread().getName() + " Kwota: " + operacje[i] + " " + konto);
             Thread.yield();
        }
    }
}
```

Przykładowe uruchomienie aplikacji będzie polegało na utworzeniu kilku wątków wykorzystujących wspólny zasób.

```
public class Zakupy {

    public static void main(String[] args) {

        Konto wspolneKonto = new Konto(1500);

        int[] operacjeMeza = {-100, -500, 300, 200, -800, 300, 145};
        int[] operacjeZony = {100, -345, 210, 180, -89, -345, -23};

        Thread maz = new Thread(new Klient(wspolneKonto, operacjeMeza), "Mąż");
        Thread zona = new Thread(new Klient(wspolneKonto, operacjeZony), "Żona");

        System.out.println(wspolneKonto);

        maz.start();
        zona.start();

        try {
            maz.join();
            zona.join();
        } catch (InterruptedException e) {
            System.out.println("Błąd: " + e.toString());
        }

        System.out.println(wspolneKonto);

    }
}
```

Przykładowe wyniki działania aplikacji:

```
Aktualny stan konta: 1500
Operację wykonuje: Żona Kwota: 100 Aktualny stan konta: 1600
Operację wykonuje: Mąż Kwota: -100 Aktualny stan konta: 1400
Operację wykonuje: Mąż Kwota: -500 Aktualny stan konta: 900
Operację wykonuje: Żona Kwota: -345 Aktualny stan konta: 1255
Operację wykonuje: Żona Kwota: 210 Aktualny stan konta: 1465
Operację wykonuje: Żona Kwota: 180 Aktualny stan konta: 1645
Operację wykonuje: Mąż Kwota: 300 Aktualny stan konta: 1200
Operację wykonuje: Mąż Kwota: 200 Aktualny stan konta: 1400
Operację wykonuje: Żona Kwota: -89 Aktualny stan konta: 1556
Operację wykonuje: Mąż Kwota: -800 Aktualny stan konta: 600
Operację wykonuje: Żona Kwota: -345 Aktualny stan konta: 1211
Operację wykonuje: Mąż Kwota: 300 Aktualny stan konta: 900
Operację wykonuje: Żona Kwota: -23 Aktualny stan konta: 1188
Operację wykonuje: Mąż Kwota: 145 Aktualny stan konta: 1045
Aktualny stan konta: 1045
```

Obserwując powyższe wyniki zauważyć można błędne dane dotyczące zmian na koncie. Kilkukrotne uruchomienie aplikacji powoduje otrzymanie różnych wyników.

Synchronizacja wątków

W sytuacjach związanych z współdzieleniem zasobów wymagana jest synchronizacja wątków czyli określenie pewnego rodzaju blokady na konkretnym zasobie. Dostęp do takiego zasobu (np. konto bankowe, plik, ekran itp.) w danym momencie posiadać może tylko jeden wątek, reszta musi czekać na zwolnienie blokady. Ochrona krytycznych elementów aplikacji odbywa się za pomocą słowa kluczowego `synchronized` i występuje jako modyfikator metody np.:

```
public synchronized void dodaj(int wartosc) {
  suma += wartosc;
}
```

lub fragment kodu (blok instrukcji) np.:

```
synchronized (object) {
  // instrukcje
}
```

Kod, który może zostać wykonany w danym momencie tylko przez jeden watek nazywa się sekcją krytyczną.

Modyfikacja kodu aplikacji do obsługi konta powinna wprowadzić sekcję krytyczną dla wszystkich wątków. W tym konkretnym przykładzie zmieniona zostanie metoda `zmienStan(kwota)`. Jej poprawny kod, odporny na błędy wygląda następująco:

```
public synchronized boolean zmienStan(int kwota) {
    int nowyStan = this.stan + kwota;

    try {
        Thread.sleep((int)(Math.random() * 1000));
    } catch (InterruptedException e) {
        System.out.println("Przerwana transakcja.");
    }

    if (nowyStan < 0) {
        return false;
    } else {
        this.stan = nowyStan;
        return true;
    }
}
```

Wyniki generowane przez zmodyfikowaną aplikację są już w porządku.

```
Aktualny stan konta: 1500
Operację wykonuje: Mąż Kwota: -100 Aktualny stan konta: 1400
Operację wykonuje: Żona Kwota: 100 Aktualny stan konta: 1500
Operację wykonuje: Mąż Kwota: -500 Aktualny stan konta: 1000
Operację wykonuje: Żona Kwota: -345 Aktualny stan konta: 655
Operację wykonuje: Mąż Kwota: 300 Aktualny stan konta: 955
Operację wykonuje: Żona Kwota: 210 Aktualny stan konta: 1165
Operację wykonuje: Mąż Kwota: 200 Aktualny stan konta: 1365
Operację wykonuje: Żona Kwota: 180 Aktualny stan konta: 1545
Operację wykonuje: Mąż Kwota: -800 Aktualny stan konta: 745
Operację wykonuje: Żona Kwota: -89 Aktualny stan konta: 656
Operację wykonuje: Mąż Kwota: 300 Aktualny stan konta: 956
Operację wykonuje: Mąż Kwota: 145 Aktualny stan konta: 1101
Operację wykonuje: Żona Kwota: -345 Aktualny stan konta: 756
Operację wykonuje: Żona Kwota: -23 Aktualny stan konta: 733
Aktualny stan konta: 733
```

Komunikacja między wątkami

Komunikacja między wątkami, czyli wymiana informacji o aktualnych blokadach, jest możliwa za pomocą następujących metod:

- `wait()` - oczekiwanie na spełnienie warunku (dokładnie na sygnał wysłany przez metody `notify()` lub `notifyAll()`),
- `notify()` - wysłanie sygnału do wątku oczekującego po wywołaniu metody `wait()`,
- `notifyAll()` - wysłanie sygnału do wszystkich wątków oczekujących po wywołaniu metody `wait()`.

Mówiąc inaczej `wait()` i `notify()` to nic innego jak wstawienie wątku do poczekalni (`wait()`) i oczekiwanie na powiadomienie go przez inny wątek (`notify()`) o możliwości dalszej pracy.

Przykładem aplikacji, która powinna korzystać z synchronizacji wątków jest rozwiązanie problemu typu **Producent-Konsument**. Producent produkuje jakieś dobra, a konsument je konsumuje. Wspólnym zasobem, który dotyczy obu wątków są, zatem dobra. Poniżej przykładowa aplikacja z brakiem synchronizacji między wątkami producenta i konsumenta. Dla uproszczenia przyjęto, że producent produkuje liczby od 1 do 10, które trafiają do pojemnika (klasa `Pojemnik`, metoda `put()`). Dostęp do pojemnika ma również wątek konsumenta (metoda `get()`).

```
class Pojemnik {
 private int n;
 private boolean pusty = true;
 int get() {
        System.out.println("Konsumpcja: " + n);
        if (pusty) System.out.println ("Skonsumowałem coś czego nie ma!");
        pusty = true;
        return n;
 }

 void put(int n) {
        this.n = n;
        System.out.println("Produkcja:  " + n);
        if (!pusty) System.out.println ("Wyprodukowałem coś niepotrzebnie!");
        pusty = false;
 }
}

class Producent implements Runnable {
 private Pojemnik p;

 Producent(Pojemnik p) {
        this.p = p;
        new Thread(this, "Producent").start();
 }

 public void run() {
        for(int i = 1; i<=10; i++) {
               p.put(i);
               try {
                      Thread.sleep((int)(Math.random() * 1000));
               } catch (InterruptedException e) {}
        }
 }
}

class Konsument implements Runnable {
 private Pojemnik p;

 Konsument(Pojemnik p) {
        this.p = p;
        new Thread(this, "Konsument").start();
```

```
  }

  public void run() {
        for(int i = 1; i<=10; i++)  {
              p.get();
              try {
                    Thread.sleep((int)(Math.random() * 1000));
              } catch (InterruptedException e) {}
        }
  }
}

public class ProducentKonsumentBS {
  public static void main(String args[]) {
        Pojemnik p = new Pojemnik();

        new Producent(p);
        new Konsument(p);
  }
}
```

Efekt działania aplikacji:

```
Produkcja:  1
Konsumpcja: 1
Konsumpcja: 1
Skonsumowałem coś czego nie ma!
Produkcja:  2
Konsumpcja: 2
Produkcja:  3
Konsumpcja: 3
Konsumpcja: 3
Skonsumowałem coś czego nie ma!
Produkcja:  4
Produkcja:  5
Wyprodukowałem coś niepotrzebnie!
Konsumpcja: 5
Produkcja:  6
Konsumpcja: 6
Produkcja:  7
Produkcja:  8
Wyprodukowałem coś niepotrzebnie!
Konsumpcja: 8
Produkcja:  9
Produkcja:  10
Wyprodukowałem coś niepotrzebnie!
Konsumpcja: 10
Konsumpcja: 10
Skonsumowałem coś czego nie ma!
```

Brak synchronizacji wątków doprowadził do wielu sytuacji niedopuszczalnych. Poniżej wersja poprawna wykorzystująca już synchronizację.

```
class Pojemnik {
  private int n;
  private boolean pusty = true;

  synchronized int get() {
        if(pusty){
              try {
                    System.out.println ("Konsument: czekam na produkcję...");
                    wait();
              } catch (InterruptedException e) { }
        }
        System.out.println("Konsumpcja: " + n);
        pusty = true;
        notify();
```

```
        return n;
  }

  synchronized void put(int n) {
        if( !pusty ) {
              try {
                    System.out.println ("Producent: czekam na konsumpcję...");
                    wait();
              } catch (InterruptedException e) {}
        }
        this.n = n;
        System.out.println("Produkcja:  " + n);
        pusty = false;
        notify();
  }
}

class Producent implements Runnable {
  private Pojemnik p;

  Producent(Pojemnik p) {
        this.p = p;
        new Thread(this, "Producent").start();
  }

  public void run() {
        for(int i = 1; i<=10; i++) {
              p.put(i);
              try {
                    Thread.sleep((int)(Math.random() * 1000));
              } catch (InterruptedException e) {}
        }
  }
}

class Konsument implements Runnable {
  private Pojemnik p;

  Konsument(Pojemnik p) {
        this.p = p;
        new Thread(this, "Konsument").start();
  }

  public void run() {
        for(int i = 1; i<=10; i++)  {
              p.get();
              try {
                    Thread.sleep((int)(Math.random() * 1000));
              } catch (InterruptedException e) { }
        }
  }
}

public class ProducentKonsumentZS {
  public static void main(String args[]) {
        Pojemnik p = new Pojemnik();

        new Producent(p);
        new Konsument(p);
  }
}
```

Efekt działania aplikacji:

```
Produkcja:  1
Konsumpcja: 1
Produkcja:  2
Konsumpcja: 2
```

```
Produkcja:  3
Konsumpcja: 3
Produkcja:  4
Konsumpcja: 4
Konsument: czekam na produkcję...
Produkcja:  5
Konsumpcja: 5
Konsument: czekam na produkcję...
Produkcja:  6
Konsumpcja: 6
Konsument: czekam na produkcję...
Produkcja:  7
Konsumpcja: 7
Produkcja:  8
Producent: czekam na konsumpcję...
Konsumpcja: 8
Produkcja:  9
Producent: czekam na konsumpcję...
Konsumpcja: 9
Produkcja:  10
Konsumpcja: 10
```

Po zsynchronizowaniu wątków wszystkie operacje przebiegają poprawnie.

Metody `wait()`, `notify()` oraz `notifyAll()` muszą być wywoływane z treści zsynchronizowanej.

Zadania

1. Zmodyfikuj kod zadania nr 4 z poprzednich ćwiczeń, tak aby wyniki pojawiały się na ekranie w prawidłowej wersji. Zastosuj synchronizację odpowiedniego zasobu (co jest wspólnym zasobem dla wszystkich wątków?)
2. Przeanalizuj aplikację dotyczącą konta bankowego. Sprawdź wyniki zarówno dla wersji nieodpornej na błędy związane z dostępem do wspólnego zasobu jak i dla wersji poprawionej.
3. Napisz klasę `Zmiennik` implementującą interfejs `Runnable`, która w metodzie `run()` będzie posiadała pętlę określoną (100 wykonań) . W pętli zostanie wywołana metoda `zmien()` poniższej klasy `Schowek`. Przy każdej iteracji wyświetl zawartość „schowka”. Utwórz cztery wątki pracujące na jednym zasobie (obiekt klasy `Schowek`), uruchom je, sprawdź otrzymywane wyniki. Czy wszystko jest w porządku?

```
public class Schowek {

    private int wartosc;

    public int zmien() {
        wartosc += 10;
        wartosc -= 10;
        return wartosc;
    }

    public String toString() {
        return ("Aktualna wartość przechowywana w schowku: " + wartosc);
    }
}
```

4. Masz daną klasę:

```
class Towar {
      private int kod;
      private double cena;
```

```
	public Towar() {
		kod = (int)(Math.random() * 100000);
		cena = (Math.random() * 1000);
	}
	public String toString() {
		return "Kod towaru:" + kod + " Cena: " + cena;
	}
}
```

Zmodyfikuj aplikację Producent-Konsument, tak by pojemnik przechowywał obiekty klasy `Towar` na liście powiązanej o rozmiarze określonym przez zmienną `max`. Wykorzystaj gotową klasę np. `LinkedList`. Producent będzie produkował towary, jeśli pojemnik jest pusty bądź też zawiera mniej niż maksymalny rozmiar pojemnika. Konsument będzie konsumował towary o ile tylko będą w pojemniku.

Co się stanie, gdy dodasz większą liczbę konsumentów? Gdzie jest błąd? Znajdź go i popraw kod aplikacji.

5. Napisz aplikację rozwiązującą problem śpiącego golibrody. 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. Zasady panujące w zakładzie fryzjerskim:
   - kiedy fryzjer kończy strzyc 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, 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.