

Współbieżność w środowisku Java

Wielowątkowość (obsługa wątków)

Zagadnienia

- Tworzenie wątków
- Stany wątków i ich zmiana
- Demony
- Grupy wątków
- Synchronizacja wątków
 - ↳ wzajemne wykluczanie
 - ↳ oczekiwanie na zmiennych warunkowych
 - ↳ pakiet `java.util.concurrent`

Klasa `java.lang.Thread`

Interfejs `java.lang.Runnable`

- Wątek reprezentowany jest w procesie na JVM przez obiekt klasy **Thread** (w szczególności jej pochodnej).
- Programem głównym wątku jest metoda `run()` klasy wywiedzionej z **Thread** lub dowolnej klasy implementującej interfejs **Runnable**.

Tworzenie wątków

- Dziedziczenie z klasy **Thread**
 - ↳ definicja klasy pochodnej od **Thread**,
 - ↳ utworzenie obiektu zdefiniowanej klasy.
- Implementacja interfejsu **Runnable**
 - ↳ definicja klasy implementującej interfejs **Runnable**,
 - ↳ utworzenie obiektu zdefiniowanej klasy,
 - ↳ utworzenie obiektu klasy **Thread** z przekazaniem referencji do utworzonego obiektu klasy implementującej **Runnable**.

Tworzenie wątków poprzez dziedziczenie z klasy **Thread**

- Zdefiniowanie klasy **MThread** wywiedzionej z klasy **Thread** (implementacja w tej klasie metody `run()`, która zawiera program wątku)

```
class MThread extends Thread {
    void run(){
        ...
    }
}
```

- Utworzenie obiektu `th` zdefiniowanej klasy **MThread**

```
MThread th;
th = new MThread();
```

Współbieżność w środowisku Java

5

Tworzenie wątków poprzez implement. interfejsu **Runnable**

- Zdefiniowanie klasy **MClass** implementującej **Runnable** oraz implementacja w tej klasie metody `run()`, która zawiera program wątku

```
class MClass implements Runnable {
    void run(){
        ...
    }
}
```

- Utworzenie obiektu `obj` zdefiniowanej klasy **MClass**

```
MClass obj = new MClass();
```

- Utworzenie obiektu `th` klasy **Thread**

```
Thread th = new Thread(obj);
```

Współbieżność w środowisku Java

6

Definicja interfejsu **Runnable**

```
public interface Runnable {  
    public abstract void run();  
}
```

Współbieżność w środowisku Java

7

Implementacja metody **run()** w klasie **Thread**

```
private Runnable target;  
public void run() {  
    if (target != null)  
        target.run();  
}
```

Referencja **target** ustawiana jest w konstruktorze, jeśli zostanie przekazany parametr klasy implementującej **Runnable**.

Współbieżność w środowisku Java

8

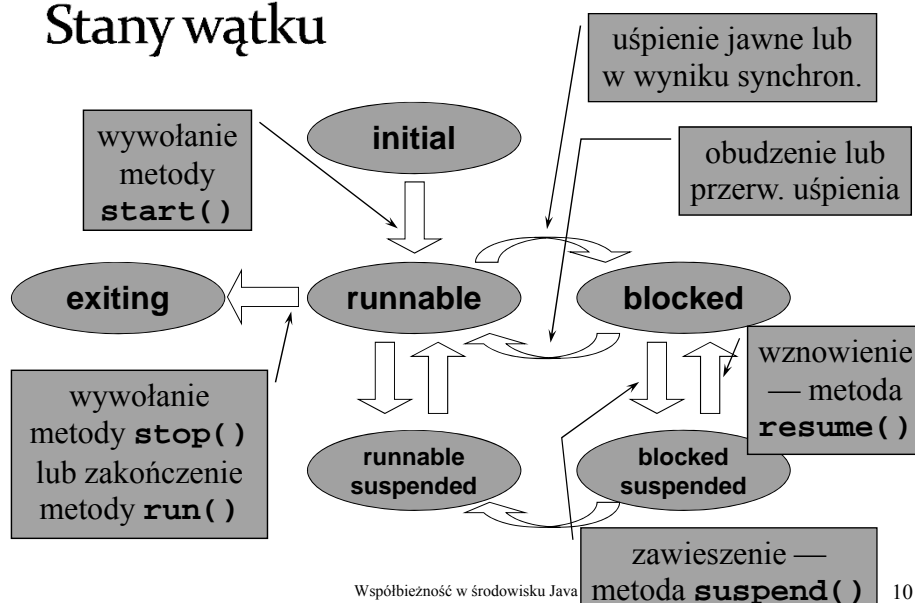
Konstruktory klasy Thread

- `Thread();`
- `Thread(Runnable target);`
- `Thread(String name);`
- `Thread(Runnable target, String name);`

Współbieżność w środowisku Java

9

Stany wątku



Współbieżność w środowisku Java

10

Klasa **Thread** — zmiana stanu wątków

- **void start()** — uruchomienie wątku,
- **void stop()** — zakończenie działania wątku,
- **void run()** — metoda wykonywana przez wątek (główny program wątku),
- **void suspend()** — zawieszenie wątku (wątek nie zwalnia blokad),
- **void resume()** — wznowienie wykonywania zawieszonoego wątku,
- **void interrupt()** — przerwanie oczekiwania wątku w stanie zablokowania.

Współbieżność w środowisku Java

11

Klasa **Thread** — zmiana stanu wątków (metody statyczne)

- **static void sleep(long milsec [, int nanosec])** — uśpienie wątku na podany okres czasu,
- **static void yield()** — „oddanie procesora” innemu wątkowi o tym samym priorytecie.
- Zmiana stanu następuje w wątku wywołującym (wątek wywołuje te metody w celu zmiany własnego stanu)

Współbieżność w środowisku Java

12

Przerywanie wątków

- Przerwanie — wywołanie metody `interrupt()` na obiekcie wątku — przerywa oczekiwanie wątku w (np. `sleep`, `join`, `wait`) poprzez zgłoszenie wyjątku `InterruptedException`.
- Jeśli wątek nie jest w stanie *blocked*, fakt przerywania jest odnotowywany poprzez ustawienie odpowiedniej flagi (*interrupted status*).
- Metoda statyczna `Thread.interrupted()` zwraca `true` jeśli flaga jest ustawiona (dla bieżącego wątku) i ją kasuje.
- Metoda `isInterrupted()` (na obiekcie wątku) zwraca tę informację dla danego wątku, ale nie kasuje flagi.

Współbieżność w środowisku Java

13

Klasa `Thread` — nadawanie nazw wątkom

- `void setName(String name)` — przypisanie nazwy do wątku,
- `String getName()` — odczytanie przypisanej nazwy.
- Z punktu widzenia systemu nazewnictwo wątków nie ma żadnego znaczenia, jest również raczej mało istotne dla użytkownika.

Współbieżność w środowisku Java

14

Klasa **Thread** — priorytety wątków

- `void setPriority(int priority)` — ustawianie priorytetu wątku,
- `int getPriority()` — odczytanie priorytetu wątku.
- Stałe (**final**) w klasie **Thread**:
 - ↳ `Thread.MIN_PRIORITY`
 - ↳ `Thread.MAX_PRIORITY`
 - ↳ `Thread.NORM_PRIORITY`
- Większa wartość oznacza wyższy priorytet.

Współbieżność w środowisku Java

15

Klasa **Thread** — inne metody

- `void join([long milsec [, int nanosec]])` — oczekiwanie na zakończenie wątku (można podać czas oczekiwania),
- `boolean isAlive()` — sprawdzenie, czy wątek działa (zwraca **true** jeśli wątek został uruchomiony przez `start()`, ale nie zakończył jeszcze działania — wykonywanie metody `run()` nie dobiegło końca).

Współbieżność w środowisku Java

16

Klasa **Thread** — inne metody statyczne

- **static Thread currentThread()** — zwraca obiekt reprezentujący aktualnie wykonywany wątek,
- **static int enumerate(Thread threadArray[])** — zwraca obiekty reprezentujące wszystkie wątki procesu,
- **static int activeCount()** — zwraca liczbę aktywnych wątków procesu.

Współbieżność w środowisku Java

17

Demony

- Demon jest takim wątkiem, który kończy swoje działanie po zakończeniu ostatniego wątku użytkownika.
- **void setDaemon(boolean on)** — w zależności od wartości parametru **on** zmienia wątek użytkownika na wątek-demon lub odwrotnie,
- **boolean isDaemon()** — sprawdza, czy wątek jest demonem.

Współbieżność w środowisku Java

18

Grupowanie wątków

- Łączenie wątków w grupy ma na celu ułatwienie zarządzania zbiorami logicznie powiązanych ze sobą wątków (np. grupa wątków w serwerze do obsługi określonego klienta na połączeniu siec.)
- Wątek musi zostać przypisany do grupy w momencie tworzenia i pozostaje w niej do końca swego istnienia.
- Grupy tworzą hierarchię wynikającą z zawierania się jednych grup w innych (każda nowo tworzona grupa jest częścią innej grupy).

Konstruktory klasy **Thread** z uwzględnieniem grupowania

```
Thread(ThreadGroup group, Runnable target);  
Thread(ThreadGroup group, String name);  
Thread(ThreadGroup group, Runnable target, String name);
```

Tworzenie grupy wątków

- Grupa wątków reprezentowana jest przez obiekt klasy **ThreadGroup**.
- Konstruktory klasy **ThreadGroup**:
 - ↳ **ThreadGroup(String name)** — utworzenie nowej grupy, która jest podgrupą grupy wątku bieżącego,
 - ↳ **ThreadGroup(ThreadGroup parent, String name)** — utworzenie nowej grupy, która jest podgrupą grupy wskazanej.

Operowanie na grupach wątków

- **void stop()** — zakończenie działania wszystkich wątków w grupie,
- **void suspend()** — zawieszenie wszystkich wątków w grupie,
- **void resume()** — wznowianie wykonywania zawieszonych wszystkich wątków w grupie.

Wyliczanie wątków w grupie

- `int enumerate(Thread list[])`
- `int enumerate(Thread list[], boolean recurse)`
- `int activeCount()`
- `int enumerate(ThreadGroup list[])`
- `int enumerate(ThreadGroup list[], boolean recurse)`

Współbieżność w środowisku Java

23

Usuwanie grupy wątków

- Do usuwania grupy wątków służy metoda `destroy()`.
- Metoda `destroy()` jest skuteczna, jeśli wszystkie wątki w grupie i podgrupach zostały zakończone.
- Metoda `destroy()` rekurencyjnie usuwa również wszystkie podgrupy grupy usuwanej.

Współbieżność w środowisku Java

24

Synchronizacja

Koordinacja wątków, spójność danych

Synchronizacja

- Synchronizacja procesów/wątków
 - ↳ koordynacja realizacji poszczególnych instrukcji (kroków, faz)
 - ↳ kontrola przepływu sterowania
- Synchronizacja danych
 - ↳ utrzymanie spójności danych
 - ↳ gwarancja dostępu do najświeższych wartości zmiennych/stanów obiektów (uwzględnienie wyników ostatnich modyfikacji)

Mechanizmy synchronizacji

- Poziom architektury systemu komputerowego
 - ↳ zapis/odczyt współdzielonych zmiennych (tzw. współdzielone rejestry)
 - ↳ złożone operacje realizowane niepodzielnie , np. `test&set`, `exchange`
- Poziom systemu operacyjnego
 - ↳ zarządzanie procesami/wątkami (ich stanem), integracja z mechanizmami przydziału procesora (szeregowania), np. semafony, zamki, zmienne warunkowe
- Poziom języka programowania
 - ↳ strukturalne mechanizmy synchronizacji udostępniające konstrukcje do wyrażania zależności i ograniczeń w dostępie do współdzielonych zasobów (monitory, regiony krytyczne)

Synchronizacja wątków

- Mechanizmy „niskopoziomowe”
 - ↳ Wzajemne wykluczanie blok/metoda `synchronized`
 - ↳ Oczekiwanie na spełnienie warunku `wait()`, `notify()`, `notifyAll()`
 - ↳ Blok `synchronized` oraz metody `wait()`, `notify()` i `notifyAll()` mogą być realizowane na dowolnym obiekcie (obiekcie klasy `Object`).
- Mechanizmy „wysokopoziomowe” — pakiet `java.util.concurrent` (od wersji 1.5)
 - ↳ Atomowe operacja na obiektach
 - ↳ Zamki (`Lock`) i zmienne warunkowe (`Condition`)
 - ↳ Semafony (`Semaphore`), bariery (`CyclicBarrier`) itp.
 - ↳ Współbieżnie dostępne kolekcje (`ConcurrentHashMap`, `ConcurrentLinkedQueue`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`)

Metoda/blok **synchronized**

- Blok **synchronized** na danym obiekcie zajmuje zamek związany (integralnie) z tym obiektem
`synchronized (obj){`
`...`
`}`
- Metoda typu **synchronized** zajmuje zamek związany z obiektem, dla którego jest wywoływana, będzie zatem wykluczać wykonanie innych metod typu **synchronized** lub bloków **synchronized** na tym obiekcie.

Współbieżność w środowisku Java

29

Przykład metod **synchronized**

```
public class Konto {
    private float kwota;
    public synchronized boolean wyplata(float k) {
        if (k <= kwota) {
            kwota -= k;
            return true;
        }
        return false;
    }
    public synchronized void wplata (float k) {
        kwota += k;
    }
}
```

Współbieżność w środowisku Java

30

Wzajemne wykluczanie fragmentu kodu metody

- Jeśli tylko fragment kodu metody ma się wykluczać z innymi metodami typu **synchronized**, można to osiągnąć przez utworzenie bloku **synchronized** na referencji **this** w implementacji tej metody.

```
{
    ...
    synchronized (this) {
        ...
    }
}
```

Współbieżność w środowisku Java

31

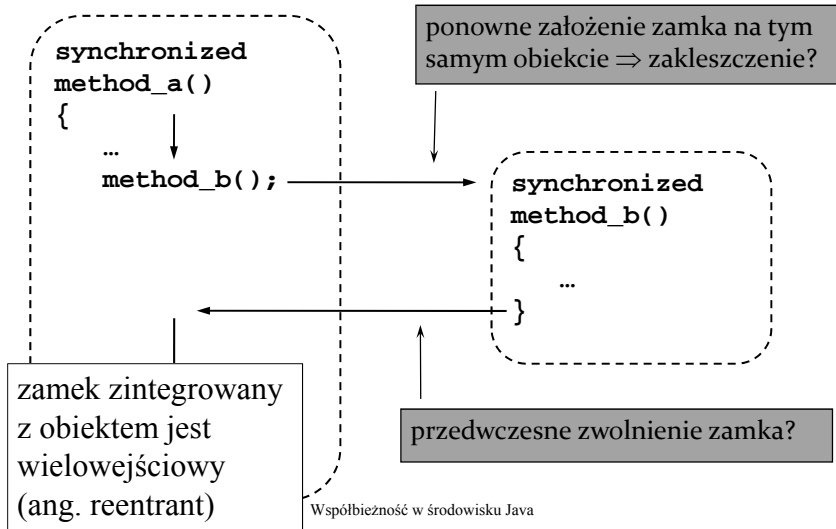
Pytanie

- Co się stanie, jeśli metoda typu **synchronized** zostanie wywołana z innej metody typu **synchronized** (czyli przez ten sam wątek)?
- Czy nastąpi zakleszczenie, a jeśli nie, to czy nie nastąpi przedwczesne zwolnienie blokady obiektu?

Współbieżność w środowisku Java

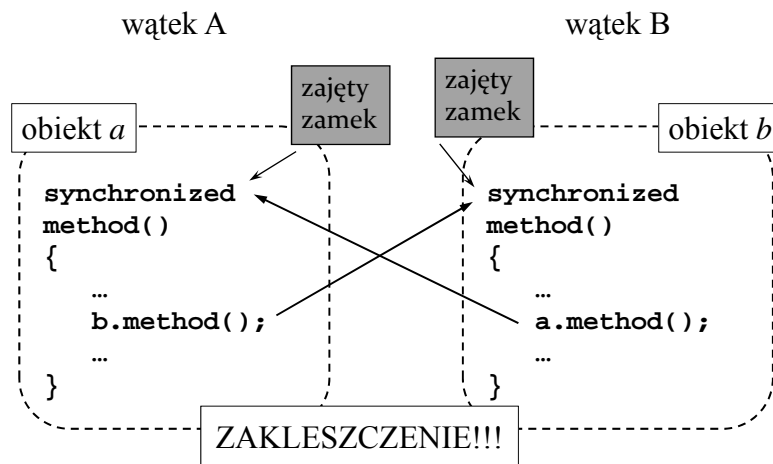
32

Przykład zagnieżdżonych blokad



33

Przykład zagnieżdżonego wzajemnego wywołania metod przez różne wątki



34

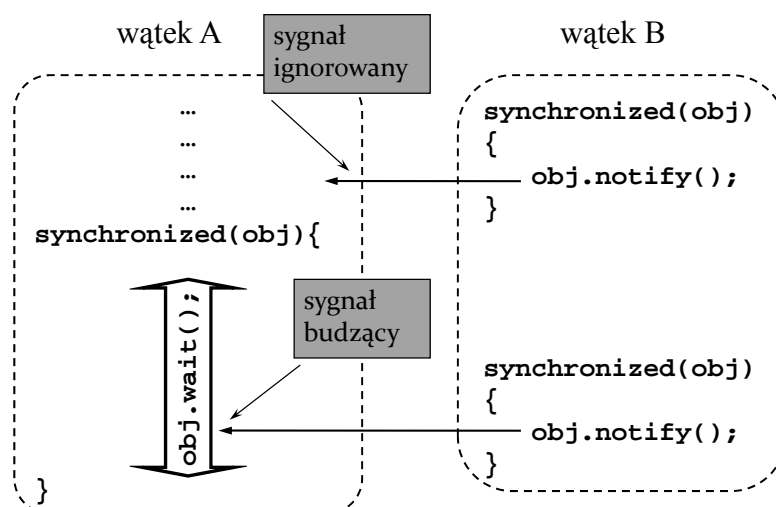
Oczekiwanie na spełnienie warunku

- `void wait([long milsec [, int nanosec]])` — czeka na spełnienie warunku (na sygnał wysyłany przez `notify()` lub `notifyAll()`),
- `void notify()` — wysyła sygnał do wątku oczekującego po wywołaniu metody `wait()` danego obiektu,
- `void notifyAll()` — wysyła sygnał do wszystkich wątków oczekujących po wywołaniu metody `wait()`,
- Metody `wait()`, `notify()` i `notifyAll()` muszą być wywoływane w bloku (metodzie) `synchronized` na tym samym obiekcie, w przeciwnym przypadku zgłaszany jest wyjątek `NotOwnerException`.

Współbieżność w środowisku Java

35

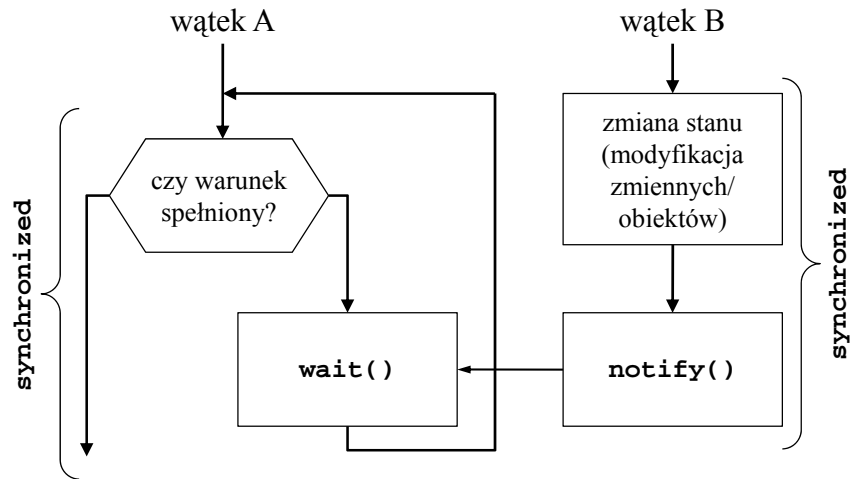
Budzenie wątków oczekujących na spełnienie warunku



Współbieżność w środowisku Java

36

Schemat synchronizacji na zmiennych warunkowych



Współbieżność w środowisku Java

37

Pytanie

- Czy przy pomocy mechanizmów synchronizacji w Java'ie da się zbudować monitor?

Współbieżność w środowisku Java

38

Atomowe operacja na obiektach

- Atomowe zmienne: `AtomicBoolean`,
`AtomicInteger`, `AtomicLong`,
`AtomicReference`
- Operacje (atomowe) na zmiennej atomowej:
`addAndGet`, `getAndAdd`, `compareAndSet`,
`decrementAndGet`, `getAndDecrement`,
`incrementAndGet`, `getAndIncrement`