Wykład 6

Procesy

Wątki

Synchronizacja

Blokady

Egzekutory

Zbiory wątków

Procesy i wątki

`proces` — to cały program, lista instrukcji, pamięć, zasoby, zmienne lokalne kilka procesów - kilka osobno działających programów lub jedne program

`wątek` - tylko lista instrukcji do wykonania, główny sposób realizowanie współbieżności w Javie

wszystkie wątki działają na wspólnych danych, pamięć jest współdzielona

```
problem – synchronizacja np. nie chcemy, aby jednocześnie w danym obszarze pamięc
zostały zapisane jakieś dane, aby nie powstał bałagan.
```

Program w Javie uruchamiany jest w ramach pojedynczego procesu JVM. Z tego powodu i**mplementacja współbieżności koncentruje się głównie na obsłudze wątków**. Niemniej Java udostępnia także mechanizmy umożliwiające uruchamianie procesów systemu operacyjnego. Proces można stworzyć korzystając z metody *Runtime.exec()* lub klasy *ProcessBuilder* z pakietu java.lang.

Przykładowe użycie procesów

Sposób I

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class ProcessExample {
 public static void main(String[] args) {
        try {
            String s;
            Process ps = Runtime.getRuntime().exec("ls -l");
            BufferedReader bri = new BufferedReader(new InputStreamReader(
                                                    ps.getInputStream()));
            BufferedReader bre = new BufferedReader(new InputStreamReader(
                                                    ps.getErrorStream()));
            while ((s = bri.readLine()) != null)
                System.out.println(s);
            bri.close();
            while ((s = bre.readLine()) != null)
                System.out.println(s);
            bre.close();
            ps.waitFor();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Gotowe.");
    }
}
```

NIE DA SIĘ UTWORZYĆ PROCESU ZA POMOCĄ KONSTRUKTORA!
Dlatego do tworzenia procesu używamy `getRuntime()` – metoda statyczna która zwraca obiekt reprezentujący proces JVM

na obiekcie typu **Runtime** wywołujemy `exec()` , który w środku zawiera instrukcję, która ma się wykonać – możemy wpisać cokolwiek co może się wykonać z linii komend terminala np. java [nazwa klasy]

`waitFor()` — znajduje się w klasie *Process* oraz służy do oczekiwania na zakończenie danego procesu i zwraca kod wyjścia tego procesu.

Sposób II

```
ProcessBuilder builder = new ProcessBuilder("ls", "-l");
 builder.directory(new File("."));
 builder.redirectErrorStream(true); //zmiana ścieżki do strumienia
 builder.redirectOutput(Redirect.INHERIT);
 Process ps;
 try {
    ps = builder.start();
    ps.waitFor();
 } catch (IOException | InterruptedException e) {
    e.printStackTrace();
 }
 System.out.println("Gotowe.");
```

Klasa `ProcessBuilder` pozwala precyzyjniej określić środowisko, w którym działa proces. Czyli, daje większą kontrolę programistom, co dokładnie ten proces zrobi.

Używanie wątków

Wątki są czasami nazywane lekkimi procesami. Zarówno procesy, jak i wątki zapewniają środowisko wykonawcze, ale utworzenie nowego wątku wymaga mniej zasobów niż utworzenie nowego procesu.

Wątki istnieją w procesie — każdy proces ma co najmniej jeden. Wątki współdzielą zasoby procesu, w tym pamięć i otwarte pliki. Dzięki temu komunikacja jest wydajna, ale potencjalnie problematyczna.

Wykonywanie wielowątkowe jest istotną cechą platformy Java. Każda aplikacja ma co najmniej jeden wątek — lub kilka, jeśli liczyć wątki „systemowe”, które wykonują takie czynności, jak zarządzanie pamięcią i obsługa sygnałów. Jednak z punktu widzenia programisty aplikacji **zaczynasz od jednego wątku, zwanego wątkiem głównym.** Ten wątek może tworzyć dodatkowe wątki, co pokażemy w następnej sekcji.

Istnieją dwa podstawowe sposoby tworzenia wątków.

- Rozszerzenie klasy `java.lang.Thread`
- Implementacja interfejsu `java.lang.Runnable`

Użycie `Thread` :

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Witam z wątku");
    }

    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
        System.out.println("Witam z programu");
    }
}
```

Musimy nadpisać metodę `run` oraz umieścić w niej kod który ma sie wykonać przez wątek.

> Sama **klasa Thread implementuje Runnable**, chociaż jej metoda uruchamiania nic nie robi

Czyli jest to swego rodzaju **szablon pustego wątku**

Klasa Thread **definiuje szereg metod przydatnych do zarządzania wątkami**. Należą do nich metody statyczne, które dostarczają informacji o wątku wywołującym metodę lub wpływają na stan wątku.

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Witam z watku");
    }

    public static void main(String args[]) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
        System.out.println("Witam z programu");
    }
}
```

↑ Ten przypadek jest ogólniejszy (i zalecany), gdyż klasa implementująca wątek może rozszerzać inną klasę.

> konstruktor **Thread()** z argumentem *new HelloRunnable()*, czyli z czymś co implementuje interfejs *Runnable*

Bardziej zwięzłe metody zastosowania

```
Runnable task = new Runnable(){
    public void run() {
        System.out.println("Witam z watku");
    }
}
Thread t = new Thread(task);
t.start();
//-------------------------------------------------------------------------
Thread t = new Thread(new Runnable(){
    public void run() {
        System.out.println("Witam z watku");
    }
});
t.start();
//-------------------------------------------------------------------------
Thread t = new Thread(() -> { System.out.println("Witam z watku");} ); //wyrażenie lambda
t.start();
```

Metody klasy Thread

- `Thread.sleep()`
  powoduje, że bieżący wątek zawiesza wykonywanie na określony czas.

Dostępne są dwie przeciążone wersje uśpienia:

1. określająca czas uśpienia w milisekundach
2. określająca czas uśpienia w nanosekundach
   Nie ma jednak gwarancji, że te czasy uśpienia będą dokładne, ponieważ są ograniczone funkcjami zapewnianymi przez podstawowy system operacyjny.Nie można zakładać, że wywołanie trybu uśpienia spowoduje zawieszenie wątku na dokładnie określony czas.

```
public class InterruptExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(10000); // wstrzymanie na 10 sek.
        } catch (InterruptedException e) {
            System.out.println("interupted");
        }
    }
    public static void main(String[] args) throws InterruptedException{
        Thread t = new Thread(new InterruptExample());
        t.start();
        Thread.sleep(5000);
        System.out.println("budzenie");
        t.interrupt();
    }
}
```

`t.interrupt()` – przerywa sen w wątku *InterruptExample* i wywołuje wyjątek InterruptedException

> Do metody **run** nie możemy dodać `throws ExampleException` ponieważ nie byłaby ona wtedy metodą interfejsu *Runnable* (nie spełniałaby specyfikacji)

Kolejność wywołania:

> t.start() → run() → sleep(10000) → Thread.sleep(5000) [usypia wątek w którym się znajduje] → “budzenie” → t.interrupt() → “interupted”

- `join()`
  powoduje, że bieżący wątek wstrzymuje wykonywanie do czasu zakończenia wątku na którym została wywołana metoda.

```
public class JoinExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("watek");
    }
    public static void main(String[] args) throws InterruptedException{
        Thread t = new Thread(new JoinExample());
        t.start();
        t.join(); // czekamy na zakonczenie t
        System.out.println("teraz ja");
    }
}
```

Podobnie jak *sleep()*, `join()` odpowiada na przerwanie, kończąc z wyjątkiem **InterruptedException**.

Kolejność wywołania:

1. z join():

> t.start() → sleep(5000) → t.join() [czeka na zakończenie wątku JoinExample] → ‘’watek’’ → “teraz ja”

2. bez join():

> t.start() → sleep(5000) → ‘’teraz ja” → ‘’watek’’

Kryterium	**`Runnable`**	**`Thread`**
Tworzenie wątków	Implementacja interfejsu `Runnable`	Rozszerzenie klasy `Thread`
Dziedziczenie	Nie ogranicza dziedziczenia	Ogranicza dziedziczenie
Metoda `run()`	Metoda `run()` jest jedyną metodą zadeklarowaną w tym interfejsie.	Zawiera metodę `run()`, ale jest ona domyślnie pusta. Definiuje szereg metod przydatnych do zarządzania wątkami.

Kryterium	`Runnable`	`Thread`
Uruchamianie wątku	Tworzony jest nowy obiekt `Thread` z instancją `Runnable` jako argumentem, a następnie wywołanie metody `start()` na tym obiekcie `Thread`.	Wątek jest uruchamiany przez utworzenie instancji klasy rozszerzającej `Thread` i wywołanie metody `start()` na tej instancji.
Wielowątkowość	Gdy implementujesz `Runnable`, wiele wątków może współdzielić ten sam obiekt.	Gdy rozszerzasz `Thread`, każdy wątek tworzy unikalny obiekt i jest z nim powiązany.

Synchronizacja

Wątki mogą się komunikować przez dowolne współużytkowane zasoby (np. referencje do obiektów). Jest to bardzo efektywne, jednak może powodować problemy, gdy kilka wątków korzysta z jednego zasobu. jest to mocno niepożądane zachowanie podczas pisania programów.

Synchronizacja metod

Problem ten rozwiązuje się używając **synchronizacji**. Oznaczenie metod słowem `synchronized` powoduje, że w danej chwili może być wykonywana tylko jedna z nich (i tylko przez jeden wątek).

```
class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Jeśli jakiś wątek wywoła na instancji naszej klasy *SynchronizedCounter* metodę oznaczoną słówkiem `synchronized`, **to żaden inny wątek nie może wywołać dowolnej tej albo innej metody**

oznaczonej słówkiem *synchronized* w tym samym czasie.

Blokada wewnętrzna

```
public void addName(String s) {
    synchronized(this) {
        name = s;
        counter++;
    }
    nameList.add(name);
}
```

Z każdym obiektem w javie jest związana **blokada wewnętrzna** . (Można to porównać do takiego prywatnego semafora dla każdej instancji obiektu) Może w ten sposób zablokować dostęp wątków do swoich zasobów. Tak działa właśnie `synchronized(this)` . Na powyższym przykładzie: ograniczamy dostęp wątków do *name* = *s* i *counter*++.
WAŻNE: istnieje dokładnie jedna taka blokada wewnętrzna związana z danym obiektem. Każda osobna instancja może mieć osobną blokadę `synchronized(this)` .

Blokada drobnoziarnista

Polega na stworzeniu "dummy" obiektów aby otrzymać **niezależne semafory** do stworzenia **dwóch niezależnych sekcji synchronized**

Przykład:

```
public class FineGrainedLockEx {
    private long c1 = 0, c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Dostęp do *c1* i *c2* musi być synchronizowany niezależnie - nie chcemy blokować na raz obu liczników.

Czyli jeden wątek może wykonać *inc1*, a drugi inny wątek może wykonać *inc2* w tym samym czasie.

> gdyby było `public synchronized void inc1()` i `public synchronized void inc2()` , to jeden wątek wykonuje metodę *inc1*, ale drugi nie może już *inc2* w tym samym czasie.

Operacje atomowe

Operacja atomowa to taka operacja, która jest wykonywana od początku do końca bez przerwy. Wyobraź sobie, że jesteś w kolejce do kasy w sklepie. Gdy już dojdziesz do kasy i zaczynasz płacić, nikt nie może Cię przeszkodzić ani przerwać tego procesu. Dopiero po zakończeniu płatności następna osoba może zacząć swoją transakcję. To jest podobne do operacji atomicznej - raz zaczęta, musi być dokończona bez przerwy.

W Javie, operacje atomowe odnoszą się do operacji, które są gwarantowane jako niepodzielne (niepodzielne, nie do przerwania) w kontekście wielowątkowości. Oznacza to, że operacja atomowa zostanie wykonana w całości, a żaden inny wątek nie może przerwać jej wykonania w trakcie trwania. To zapewnia spójność danych w wielowątkowym środowisku.

> Typowo dostęp do zmienne/referencji nie jest realizowany jako pojedyncza operacja. Aby takie operacje (odczytu/zapisu zmiennej) były atomowe należy oznaczyć ją słowem `volatile` .

Uwaga:

```
private volatile long c1 = 0;

 public void inc1() {
    c1 += 5; // nie jest atomowa bo atomowy jest odczyt i zapis wartosci
             //a nie jej zwiekszanie
 }
```

Główne cechy `volatile` :

1. **Brak blokady (No Locking)**:
   To trochę jak w sztafecie biegowej. Kiedy jeden biegacz przekazuje pałeczkę następnemu, nie ma gwarancji, że pałeczka nie upadnie w trakcie przekazywania.
   W Javie, kiedy używamy `volatile` , nie ma gwarancji, że dwie linie kodu nie będą próbować zmienić tej samej wartości w tym samym czasie (to jest tzw. *“problem świadomości”*). Ale volatile zapewnia, że jeśli do tego dojdzie, to każda linia kodu zobaczy najnowszą wartość.
2. **Widoczność (Visibility)**:
   Masz dwie osoby - Jana i Annę. Jan pisze list do Anny, ale zamiast wysłać go od razu, zostawia go na swoim biurku. Anna nie może zobaczyć tego listu, dopóki Jan go nie przekaże. Teraz, jeśli Jan zdecyduje się zmienić coś w liście, Anna nie będzie o tym wiedzieć, dopóki Jan nie da jej zaktualizowanej wersji.
   To jest podobne do tego, jak działa `volatile` w Javie. Gdy jedna linia kodu (Jan) zmienia wartość zmiennej, wszystkie inne linie kodu (Anna) natychmiast widzą tę zmianę. Nie ma opóźnień ani nieporozumień - wszyscy widzą tę samą, najnowszą wartość.

> Słowo kluczowe volatile stosuje się do zmiennych, oznaczając, że wartość tej zmiennej może być zmieniana przez różne wątki. Gwarantuje, że zapisy do zmiennej są widoczne dla innych wątków natychmiast po zapisie. Jednakże, w przypadku operacji, które nie są atomowe (czyli operacje, które wymagają kilku kroków), volatile sam w sobie nie jest wystarczające, ponieważ nie gwarantuje, że te operacje będą niepodzielne.

Typowe problemy

- `zakleszczenia (deadlock)` — wątek blokują wzajemnie zasoby potrzebne im do dalszego działania (pięciu filozofów).
- `zagłodzenia (starvation)` — jeden wątek przez cały blokuje zasób potrzebny innym wątkom.
- `livelock` —(“odwrotność” deadlocka) — wątek reaguje na zachowanie drugiego wątku, które jest reakcją na zachowanie pierwszego wątku.

Przykład zakleszczenia

```
public class Deadlock {
    static class Worker {
        public String name;
        public Worker(String name) {
            this.name = name;
        }
        public synchronized void doWork(Worker w){
            System.out.println(this.name + " pracuje z " + w.name);
            try {
                Thread.sleep(1000); // pracujemy
            } catch (InterruptedException e) { }
            w.release();
        }
        public synchronized void release() {
            System.out.println(this.name + " jest znowu gotowy");
        }
    }
    public static void main(String[] args) {
        final Worker w1 = new Worker("w1");
        final Worker w2 = new Worker("w2");
        new Thread(new Runnable() {
            public void run() { w1.doWork(w2); }
        }).start();

        new Thread(new Runnable() {
            public void run() { w2.doWork(w1); }
        }).start();
    }
}
```

w1 czeka na opuszczenie blokady w2 i na odwrót, występuje zakleszczenie na metodzie release(), gdy w1 i w2 naraz chcą ją wykonać.

wait() i notify()

`wait()`

- zdefiniowana w klasie ***Object***.
  Co robi?:

- wątek zatrzymuje swoje wykonywane operacje
- zwalnia blokadę narzuconą przez słówko `synchronized`
- Po wznowieniu wątku wykonywane są operacje nastepne operacje po `wait()`

`noitfy()` / `notifyAll()`

- zdefiniowana w klasie ***Object***.

  Co robi?:
- `noitfy()` powoduje obudzenie jednego wątku, który wstrzymał swoje działanie
- `notifyAll()` powoduje obudzenie wszystkich wątków, które wstrzymały swoje działanie

  Konsekwencja tego co robi:
- dany wątek który znowu zaczął się wykonywać po intrukcji *wait()* podniesie blokadę dla reszty wątków

Przykład:

```
public synchronized consume() {
    while(!available) {
        try {
            wait(); // wstrzymuje dzialanie watku i zwalnia blokadę
        } catch (InterruptedException e) {}
    }
    System.out.println("Skonsumowane");
    available = false;
}
public synchronized produce() {
    doProduce();
    available = true;
    notifyAll(); // powiadamia (budzi) wszystkie czekajace watki
}
```

Synchronizacja cd.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockObjects {
    static class Worker {
        public Lock lock = new ReentrantLock();
        public String name;
        public Worker(String name) {
            this.name = name;
        }
        public boolean tryWorking(Worker w) {
            boolean myLock = lock.tryLock();
            boolean wLock = w.lock.tryLock();
            if (!(myLock && wLock)) { // zwalniamy blokady
                if (myLock)
                    lock.unlock();
                if (wLock)
                    w.lock.unlock();
            }
            return myLock && wLock;
        }
        public synchronized void doWork(Worker w) {
            boolean done = false;
            while (!done) {
                if (tryWorking(w)) {
                    System.out.println(name + ": pracuje z " + w.name);
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) { }
                    w.release();
                    this.lock.unlock();
                    done = true;
                } else {
                    System.out.println(name+": jestem zajety wiec czekam");
                    try {
                        wait();
                    } catch (InterruptedException e) { }
                    System.out.println(this.name + ": probuje znowu");
                }
            }
        }
        public synchronized void release() {
```

```
            System.out.println(this.name + ": jestem znowu gotowy");
            this.lock.unlock();
            notifyAll();
        }
    }
    public static void main(String[] args) {
        final Worker w1 = new Worker("w1");
        final Worker w2 = new Worker("w2");

        new Thread(new Runnable() {
            public void run() { w1.doWork(w2); }
        }).start();

        new Thread(new Runnable() {
            public void run() { w2.doWork(w1); }
        }).start();
    }
}
```

Można to porównać do naszego tricku z *"dummy"* obiektami. `lock` stanowią osobne blokady, ale nie w formie semafora, a osobnych obiektów na których można pracować. Dzięki temu instrukcje *trylock()*, *unlock()* działają atomowo.

Egzekutory

> We wszystkich poprzednich przykładach istnieje ścisłe powiązanie między zadaniem wykonywanym przez nowy wątek, zdefiniowanym przez jego obiekt *Runnable*, a samym wątkiem, zdefiniowanym przez obiekt *Thread*. Działa to dobrze w przypadku małych aplikacji, ale w aplikacjach na dużą skalę sensowne jest oddzielenie zarządzania i tworzenia wątków od reszty aplikacji.

`Egzekutory` w Javie to narzędzia, które pomagają w zarządzaniu i kontrolowaniu wątków, oddzielają wątek od zarządzania nim.

Oto kilka kluczowych punktów:

1. **Egzekutory i pule wątków:**
   Egzekutory ułatwiają uruchamianie wątków bez konieczności tworzenia nowej instancji dla każdego wątku. Zamiast tego, możemy użyć puli wątków, która pozwala na wielokrotne użycie wątków.
2. **Interfejsy egzekutorów:** — w pakiecie *java.util.concurrent*

- `Executor` — prosty interfejs wspierający uruchamianie nowych zadań
- `ExecutorService` — podinterfejs *Executora*, który dodaje funkcje pomagające zarządzać cyklem życia, zarówno poszczególnych zadań, jak i samego executora
- `ScheduledExecutorService` — podinterfejs *ExecutorService*, wspiera przyszłe i/lub okresowe wykonywanie zadań

> ***Thread Pools* są najpopularniejszym rodzajem implementacji egzekutorów**

W JDK 7 wprowadzono *ForkJoinPool*, przeznaczony do równoległego wykonywania obliczeń na wielu procesorach

Executor

zapewnia pojedynczą metodę wykonania, zaprojektowaną jako bezpośredni zamiennik popularnego sposobu tworzenia wątków. Jeśli *r* jest obiektem *Runnable*, a *e* jest obiektem *Executor*, to wyrażenie:

```
(new Thread(r)).start();
```

możesz zastąpić:

```
e.execute(r);
```

Zastąpione wyrażenie tworzy nowy wątek i natychmiast go uruchamia. W zależności od implementacji Executora, „execute” może zrobić to samo, ale jest bardziej prawdopodobne, że użyje istniejącego wątku roboczego do uruchomienia r lub umieści r w kolejce w celu oczekiwania na udostępnienie wątku roboczego.

ExecutorService

Uzupełnia metodę *execute* przy użyciu podobnej, ale bardziej wszechstronnej metody przesyłania. Podobnie jak wykonanie, przesyłanie akceptuje obiekty *Runnable*, ale akceptuje także obiekty *Callable*, które pozwalają zadaniu zwrócić wartość. Metoda przesyłania zwraca obiekt *Future*, który służy do pobierania zwracanej wartości *Callable* i zarządzania stanem zadań *Callable* i *Runnable*.

ScheduledExecutorService

Uzupełnia metody swojego nadrzędnego interfejsu *ExecutorService* o szablon, który wykonuje zadanie uruchamialne lub wywoływalne po określonym opóźnieniu. Dodatkowo w interfejsie

zdefiniowano szablony `scheduleAtFixedRate` i `scheduleWithFixedDelay`, które wykonują określone zadania wielokrotnie, w określonych odstępach czasu.