

Programowanie współbieżna

Język Java – wątki (streszczenie)

Paweł Rogaliński

Instytut Informatyki, Automatyki i Robotyki
Politechniki Wrocławskiej

pawel.rogalinski @ pwr.wroc.pl

Podstawowe pojęcia: procesy i wątki

Proces to wykonujący się program wraz z dynamicznie przydzielanymi mu przez system zasobami (np. pamięcią operacyjną, zasobami plikowymi). Każdy proces ma własną przestrzeń adresową.

Systemy wielozadaniowe pozwalają na równoległe (teoretycznie) wykonywanie wielu procesów, z których każdy ma swój kontekst i swoje zasoby.

Wątek to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)

Każdy proces ma co najmniej jeden wykonujący się wątek. W systemach wielowątkowych proces może wykonywać równoległe (teoretycznie) wiele wątków, które wykonują się jednej przestrzeni adresowej procesu.

Podstawowe pojęcia: procesy i wątki

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na krótki czas (kwant czasu), po czym „oddaje procesor” innemu wątkowi.

Zmiana wątku wykonywanego przez procesor może dokonywać się na zasadzie:

- **współpracy** (*cooperative multitasking*), wątek sam decyduje, kiedy oddać czas procesora innym wątkom,
- **wywłaszczenia** (*pre-emptive multitasking*), o dostępie wątków do procesora decyduje systemowy zarządca wątków, który przydziela wątkowi kwant czasu procesora, po upływie którego odsuwa wątek od procesora i przydziela kolejny kwant czasu innemu wątkowi.

Java jest językiem wieloplatformowym, a różne systemy operacyjne stosują różne mechanizmy udostępniania wątkom procesora. Programy wielowątkowe powinny być tak pisane, by działały zarówno w środowisku „współpracy” jak i „wywłaszczenia”

Tworzenie i uruchamianie wątków

Uruchamianiem wątków i zarządzaniem nimi zajmuje się klasa *Thread*.

Aby uruchomić wątek, należy utworzyć obiekt klasy *Thread*
i dla tego obiektu wywołać metodę *start()*.

Kod wykonujący się jako wątek – sekwencja działań wykonująca się równolegle z innymi działaniami programu – określany jest przez obiekt implementujący interfejs *Runnable*, który zawiera deklarację metody *run()*.

Metoda *run()* określa to co ma robić wątek.

Główne metody klasy *Thread*

1. Uruchamianie i zatrzymywanie wątków:

- *start* - uruchomienie wątku,
- *stop* – zakończenie wątku (metoda niezalecana),
- *run* - kod wykonywany w ramach wątku.

2. Identyfikacja wątków:

- *currentThread* - metoda zwraca identyfikator wątku bieżącego,
- *setName* - ustawienie nazwy wątku,
- *getName* - odczytanie nazwy wątku,
- *isAlive* - sprawdzenie czy wątek działa,
- *toString* - uzyskanie atrybutów wątku.

3. Priorytety i szeregowanie wątków:

- *getPriority* - odczytanie priorytetu wątku,
- *setPriority* - stawienie priorytetu wątku,
- *yield* - wywołanie szeregowania.

Główne metody klasy *Thread* c.d.

4. Synchronizacja wątków:

- *sleep* - zawieszenie wykonania wątku na dany okres czasu,
- *join* - czekanie na zakończenie innego wątku,
- *wait* - czekanie w monitorze,
- *notify* - odblokowanie wątku zablokowanego na monitorze,
- *notifyAll* - odblokowanie wszystkich wątków zablokowanych na monitorze,
- *interrupt* - odblokowanie zawieszonych wątków,
- *suspend* - zablokowanie wątku,
- *resume* - odblokowanie wątku zawieszonych przez *suspend*,
- *setDaemon* - ustanowienie wątku demonem,
- *isDaemon* - testowanie czy wątek jest demonem.

Tworzenie wątków

Możliwe są dwa sposoby tworzenia nowego wątku:

- poprzez dziedziczenie klasy *Thread*
- poprzez implementację interfejsu *Runnable*.

Sposób drugi stosujemy wówczas, gdy klasa reprezentująca wątek musi dziedziczyć po innej niż *Thread* klasie (Java nie dopuszcza dziedziczenia wielobazowego).

Tworzenie wątku – dziedziczenie klasy *Thread*

Aby utworzyć wątek jako klasa potomna od klasy *Thread* należy:

1. Utworzyć nową klasę (na przykład o nazwie *TKlasa*) jako potomną klasy *Thread* i nadpisać metodę *run()* klasy macierzystej. Metoda ta ma zawierać kod do wykonania w ramach tworzonego wątku.

```
class TKlasa extends Thread {  
    ....  
    void run() {  
        // Kod wątku  
    }  
    ....  
}
```

2. Utworzyć obiekt nowej klasy *TKlasa* (na przykład *thr*):

```
TKlasa thr = new TKlasa(...);
```

3. Wykonać metodę *start()* klasy *TKlasa* dziedziczoną z klasy macierzystej:

```
thr.start();
```


Tworzenie wątku – dziedziczenie klasy *Thread*

```
class NowyWatek extends Thread
{
    public void run()
    { System.out.println("    Nowy watek : POCZATEK");
      try { for(int i = 5; i > 0; i--)
            { System.out.println("    Nowy watek: " + i);
              Thread.sleep(500);
            }
        } catch (InterruptedException e) {}
      System.out.println("    Nowy watek : KONIEC");
    }
}
```

działanie
nowego wątku

```
class GlownyWatek
{
    public static void main(String args[])
    { System.out.println(" Glowny watek: POCZATEK");

      System.out.println(" Glowny watek: Tworze Nowy Watek");
      NowyWatek nowyWatek = new NowyWatek();

      System.out.println(" Glowny watek: Uruchamiam Nowy watek");
      nowyWatek.start();

      try { for(int i = 5; i > 0; i--)
            { System.out.println(" Glowny watek: " + i);
              Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}

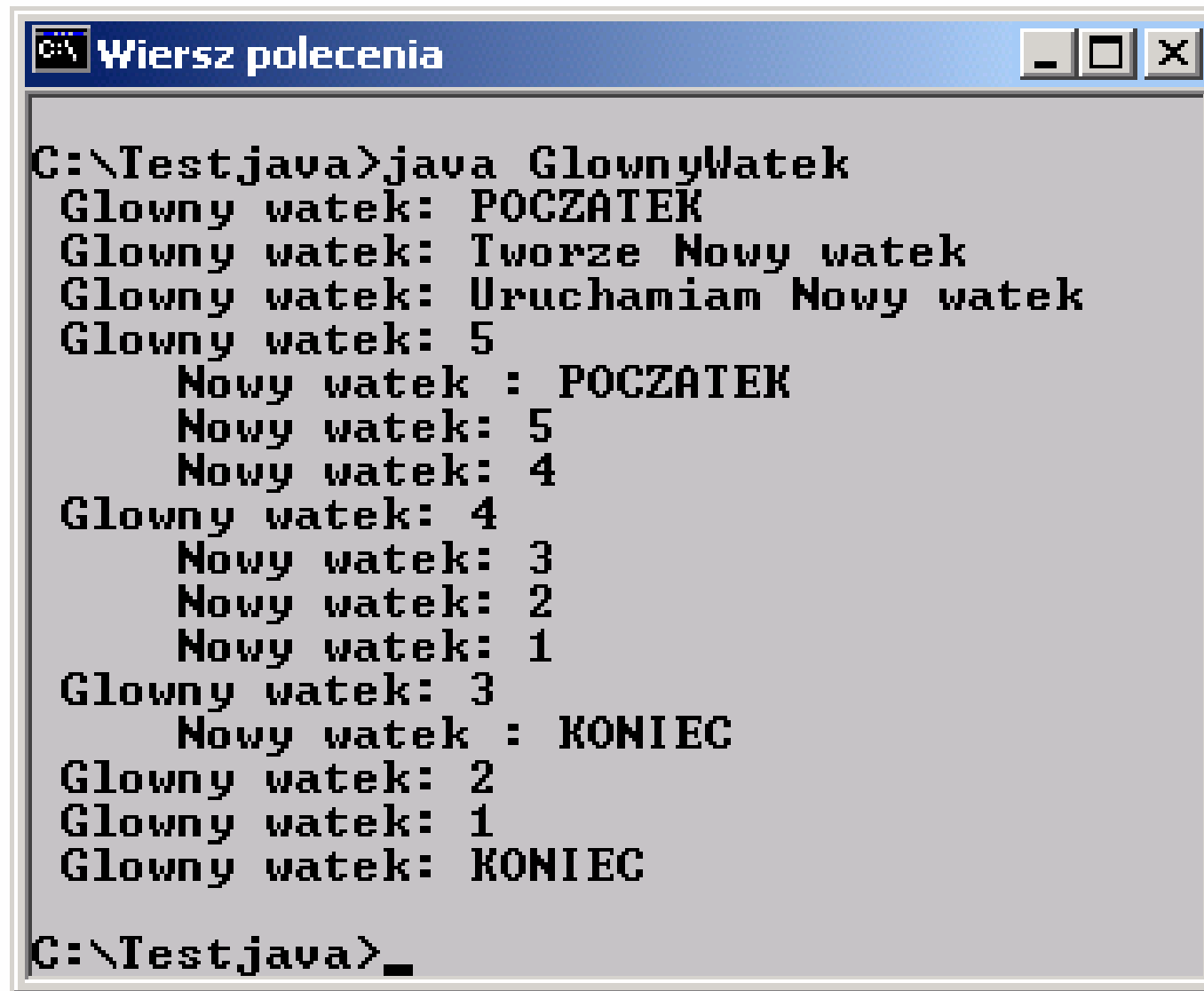
      System.out.println(" Glowny watek: KONIEC");
    }
}
```

tworzenie
nowego wątku

uruchomienie
nowego wątku

działanie
głównego
wątku

Tworzenie wątku – dziedziczenie klasy *Thread*



```
Wiersz polecenia
C:\Testjava>java GlownyWatek
Glowny watek: POCZATEK
Glowny watek: Tworze Nowy watek
Glowny watek: Uruchamiam Nowy watek
Glowny watek: 5
    Nowy watek : POCZATEK
    Nowy watek: 5
    Nowy watek: 4
Glowny watek: 4
    Nowy watek: 3
    Nowy watek: 2
    Nowy watek: 1
Glowny watek: 3
    Nowy watek : KONIEC
Glowny watek: 2
Glowny watek: 1
Glowny watek: KONIEC
C:\Testjava>_
```

Tworzenie wątku – implement. interfejsu *Runnable*

Aby utworzyć wątek korzystając z interfejsu *Runnable* należy:

1. Utworzyć nową klasę (np. o nazwie *RKlasa*) dziedziczącą po interesującej na innej klasie (np. o nazwie *InnaKlasa*) i implementującą interfejs *Runnable*. W ramach tej nowej klasy utworzyć metodę *run()*, która wykonywała będzie żądane czynności.

```
class RKlasa extends InnaKlasa implements Runnable {  
    public void run() {  
        // Zawartość metody run  
    }  
}
```

2. Utworzyć obiekt tej nowej klasy

```
RKlasa r1 = new RKlasa();
```

3. Utworzyć obiekt klasy *Thread* przekazując obiekt wcześniej utworzonej klasy jako parametr konstruktora klasy *Thread*.

```
Thread t1 = new Thread(r1);
```

4. Uruchomić wątek wykonując metodę *start()* klasy *Thread*.

```
t1.start();
```

Tworzenie wątku – implement. interfejsu *Runnable*

```
class NowyWatek implements Runnable
{
    public void run()
    { System.out.println("    Nowy watek : POCZATEK");
      try { for(int i = 5; i > 0; i--)
            { System.out.println("    Nowy watek: " + i);
              Thread.sleep(500);
            }
        } catch (InterruptedException e) {}
      System.out.println("    Nowy watek : KONIEC");
    }
}
```

działanie
nowego wątku

```
class GlownyWatek
{
    public static void main(String args[])
    { System.out.println(" Glowny watek: POCZATEK");

      System.out.println(" Glowny watek: Tworze Nowy watek");
      NowyWatek nowyWatek = new NowyWatek();
      Thread thread = new Thread(nowyWatek);

      System.out.println(" Glowny watek: Uruchamiam Nowy watek");
      thread.start();

      try { for(int i = 5; i > 0; i--)
            { System.out.println(" Glowny watek: " + i);
              Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
      System.out.println(" Glowny watek: KONIEC");
    }
}
```

tworzenie
nowego wątku

uruchomienie
nowego wątku

działanie
głównego
wątku

Kończenie pracy wątku

Wątek kończy pracę w sposób naturalny gdy zakończy się jego metoda `run()`.

Wykonanie metody `stop()` powoduje zatrzymanie innego wątku. Gdy zatrzymywany wątek znajduje się wewnątrz monitora to wejście do tego monitora zostaje odblokowane. Metoda ta jest wycofana (istnieje ale użycie jej nie jest zalecane). Powodem jest fakt że wywołując tę metodę nie wiemy w jakim stanie jest zatrzymywany wątek. Gdy wykonuje jakieś krytyczne operacje może pozostawić obiekt w nieprawidłowym stanie.

Jeśli chcemy programowo zakończyć pracę wątku, powinniśmy zapewnić w metodzie `run()` sprawdzanie warunku zakończenia (ustalanego programowo) i jeśli warunek ten jest spełniony, spowodować wyjście z metody `run()`. Warunek zakończenia może być formułowany w postaci jakiejś zmiennej, która jest ustalana przez inne fragmenty kodu programu (wykonywane w innym wątku).

Kończenie pracy wątku - przykład

```
class NowyWatek extends Thread
{
    boolean zakoncz = false;

    public void run()
    { System.out.println("    Nowy watek : POCZATEK");
      while (zakoncz==false)
        { try { sleep(200);
              } catch (InterruptedException e) {}
          System.out.print(" .");
        }
      System.out.println("\n    Nowy watek : KONIEC");
    }
}

class GlownyWatek
{
    public static void main(String args[])
    { System.out.println(" Glowny watek: POCZATEK");
      System.out.println(" Glowny watek: Tworze Nowy watek");
      NowyWatek nowyWatek = new NowyWatek();
      nowyWatek.start();
      try { Thread.sleep(10000);
            } catch (InterruptedException e) {}

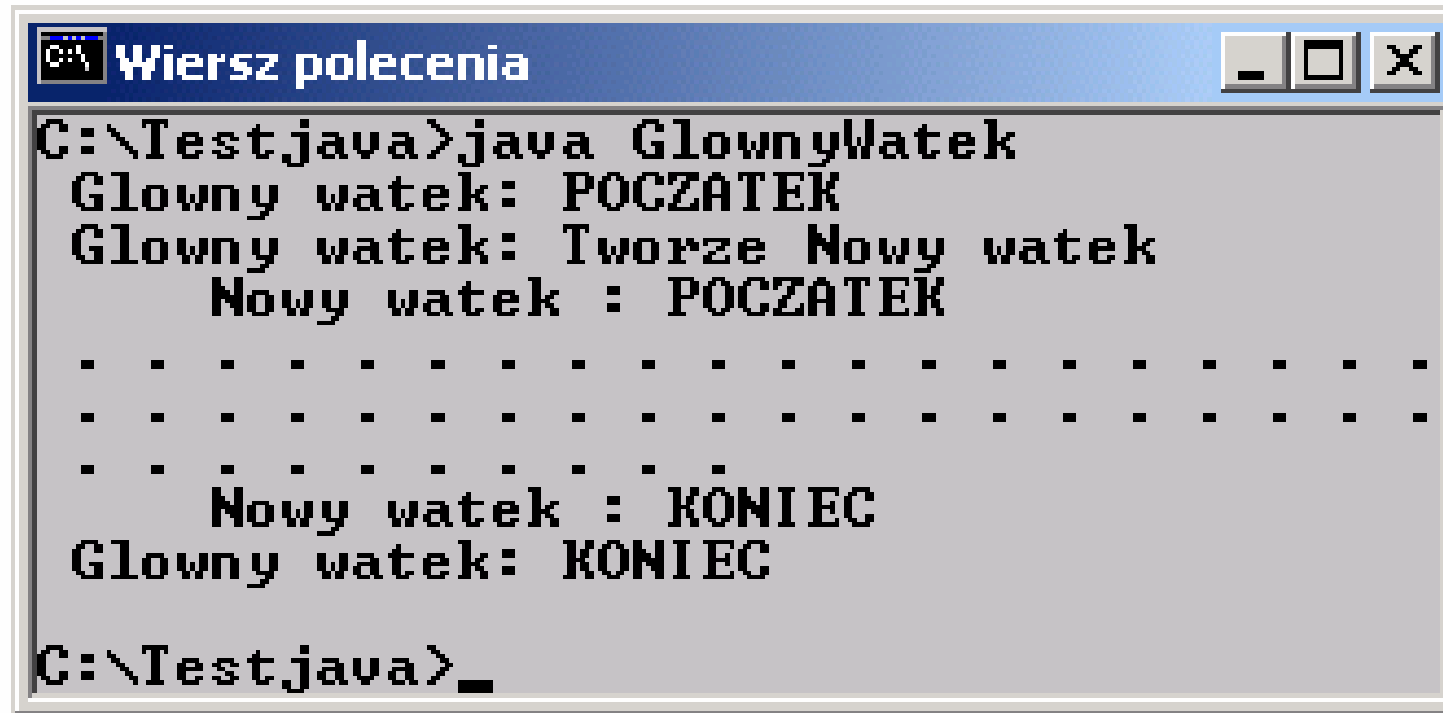
      nowyWatek.zakoncz = true;

      try { Thread.sleep(2000);
            } catch (InterruptedException e) {}
      System.out.println(" Glowny watek: KONIEC");
    }
}
```

testowanie warunku
zakończenia wątku

wymuszenie warunku
zakończenia wątku

Kończenie pracy wątku - przykład



```
C:\Testjava>java GłównyWątek
Główny wątek: POCZATEK
Główny wątek: Tworze Nowy wątek
    Nowy wątek : POCZATEK
. . . . .
. . . . .
. . . . .
    Nowy wątek : KONIEC
Główny wątek: KONIEC

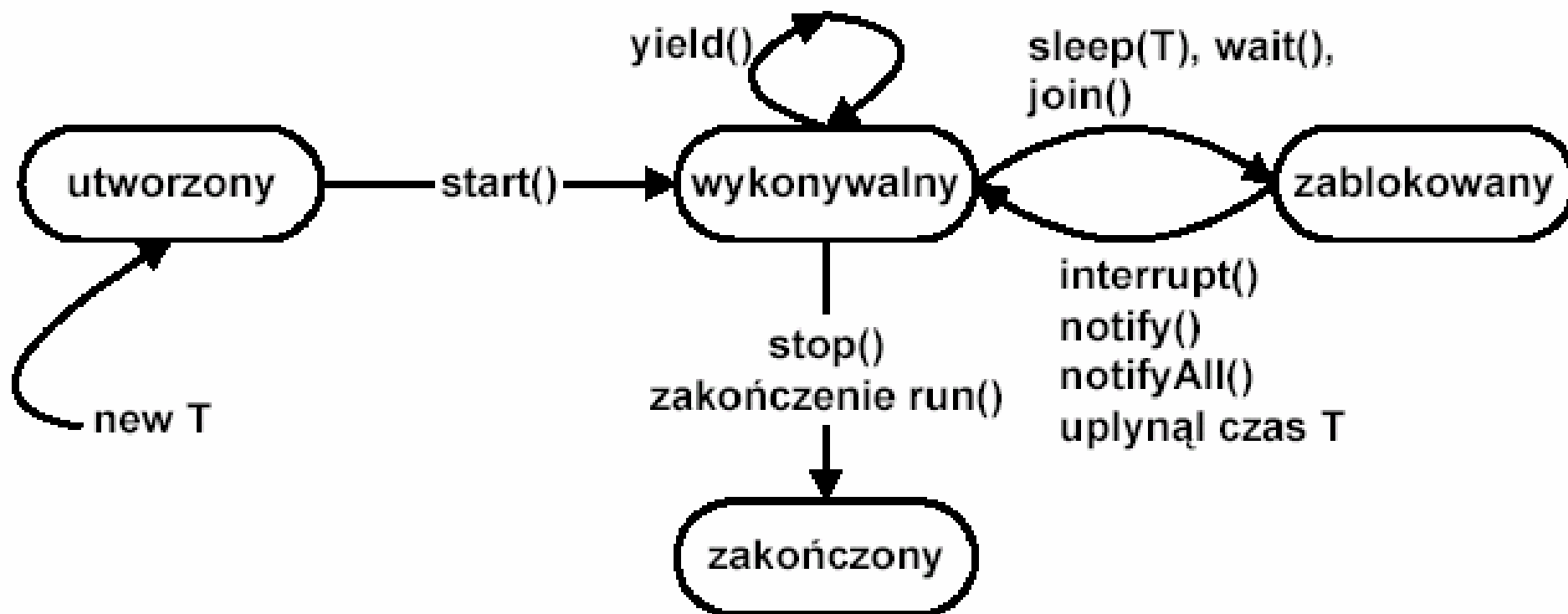
C:\Testjava>_
```

Stany wątków

Wątek może znajdować się w jednym z czterech stanów:

- utworzony (ang. *new thread*) - obiekt wątku został już utworzony ale nie wykonano metody *start()*, a więc wątek nie jest jeszcze szeregowany,
- wykonywalny (ang. *runnable*) - Wątek posiada wszystkie zasoby aby być wykonywany. Będzie wykonywany gdy tylko procedura szeregująca przydzieli mu procesor,
- zablokowany (ang. *blocked*) - Wątek nie może być wykonywany gdyż brakuje mu pewnych zasobów. Dotyczy to w szczególności operacji synchronizacyjnych (wątek zablokowany na wejściu do monitora, operacje *wait*, *sleep*, *join*) i operacji wejścia wyjścia,
- zakończony (ang. *dead*) - Stan po wykonaniu metody *stop()*. Zalecanym sposobem kończenia wątku jest zakończeni metody *run()*.

Stany wątków



Stany wątków

Przejście od stanu wykonywalny do zablokowany następuje gdy:

1. wątek chce wejść do zablokowanego monitora
2. wykonana została metoda `wait()`, `join()`, `suspend()`
3. wywołano metodę `sleep(...)`
4. wątek wykonał operację wejścia / wyjścia.

Powrót od stanu zablokowany do wykonywany następuje gdy:

1. monitor został odblokowany
2. inny wątek wykonał operacja odblokowania zablokowanego wątku – wywołał metodę `notify()`, `notifyAll()`, `resume()`, `interrupt()`
3. gdy wątek zakończył wykonywanie metody `sleep()` – upłynął zadany interwał czasu.
4. jeżeli wątek czekał na zakończenie operacji wejścia / wyjścia – operacja ta się zakończyła

Synchronizacja wątków - przykład

Pytanie:

Jaką wartość zwróci metoda `balance()` ?


```
class Balance
{
    private int number = 0;

    public int balance()
    {
        number++;
        number--;
        return number;
    }
}
```

Synchronizacja wątków – przykład c.d.

Zmiany wartości zmiennej *number* gdy wykonywany jest tylko jeden wątek:

Wartości zmiennej <i>number</i>	wykonywane instrukcje w wątku
0	...
0	<i>balance(){</i>
0	<i> number++;</i>
1	<i> number--;</i>
0	<i> return number;</i>
0	<i>}</i>
0	...



zwróci wartość 0

Synchronizacja wątków – przykład c.d.

Zmiany wartości zmiennej *number* gdy wykonywane są dwa wątki:

Wartości zmiennej <i>number</i>	wykonywane instrukcje w pierwszym wątku	wykonywane instrukcje w drugim wątku
0	<pre>... balance(){ number++; number--; return number; } ...</pre>	
0		
0		
1		
0		
0		
0		
0		
0		
0		
0		
0		<pre>... balance(){ number++; number--; return number; } ...</pre>
0		
0		
0		
1		
0		
0		
0		
0		
0		

zwróci wartość 0

wyłączenie pierwszego wątku

zwróci wartość 0

Synchronizacja wątków – przykład c.d.

Zmiany wartości zmiennej *number* gdy wykonywane są dwa wątki:

Wartości zmiennej <i>number</i>	wykonywane instrukcje w pierwszym wątku	wykonywane instrukcje w drugim wątku	
0	...		
0	<i>balance(){</i>		
0	<i>number++;</i>		wyłączenie pierwszego wątku
1		...	
1		<i>balance(){</i>	
1		<i>number++;</i>	
2		<i>number--;</i>	
1		<i>return number;</i>	zwróci wartość 1
1		}	
1		...	wyłączenie drugiego wątku
1	<i>number--;</i>		
0	<i>return number;</i>		
0	}		zwróci wartość 0
...	...		

Synchronizacja wątków – przykład c.d.

Zawsze musimy się liczyć z tym, że wątki operujące na współdzielonych zmiennych mogą być wyłączone w trakcie operacji (nawet pojedynczej) i wobec tego stan współdzielonej zmiennej może okazać się niespójny.

Testowanie programów wielowątkowych jest trudne, bowiem możemy wiele razy otrzymać wyniki, które wydają się świadczyć o poprawności programu, a przy kolejnym uruchomieniu okaże się, że wynik jest nieprawidłowy.

Wyniki uruchamiania programów wielowątkowych mogą być także różne na różnych platformach systemowych.

Synchronizacja wątków

Komunikacja między wątkami opiera się na wspólnej pamięci. W takim przypadku występuje zjawisko wyścigów.

Wyścigi (ang. *race conditions*) – wynik działania procedur wykonywanych przez wątki zależy od kolejności ich wykonania.

Gdy kilka wątków ma dostęp do wspólnych danych i przynajmniej jeden je modyfikuje występuje konieczność synchronizowania dostępu do wspólnych danych.

Synchronizacja jest mechanizmem, który zapewnia, że kilka wykonujących się wątków:

- nie będzie równocześnie działać na tym samym obiekcie,
- nie będzie równocześnie wykonywać tego samego kodu.

Kod, który może być wykonywany w danym momencie tylko przez jeden wątek, nazywa się sekcją krytyczną. W Javie sekcje krytyczne wprowadza się jako bloki lub metody synchronizowane.

Synchronizacja wątków - monitory

Każdy egzemplarz klasy *Object* i jej podklas posiada monitor (ang. *lock*), który ogranicza dostęp do obiektu. Blokowanie obiektów jest sterowane słowem kluczowym *synchronized*.

Synchronizacja w Javie może być wykonana na poziomie:

- metod – słowo kluczowe *synchronized* występuje przy definiowaniu metody:

```
public synchronized int balance()  
{...}
```

- instrukcji - słowo kluczowe *synchronized* występuje przy definiowaniu bloku instrukcji:

```
synchronized( number )  
{ number++;  
  number--;  
}
```

Synchronizacja wątków

Kiedy wątek wywołuje na rzecz jakiegoś obiektu metodę synchronizowaną, automatycznie zamykany jest monitor (obiekt jest zajmowany przez wątek). Inne wątki usiłujące wywołać na rzecz tego obiektu metodę synchronizowaną (niekoniecznie tą samą) lub usiłujące wykonać instrukcję *synchronized* z podaną referencją do zajętego obiektu są blokowane i czekają na zakończenie wykonywania metody lub instrukcji *synchronized* przez wątek, który zajął obiekt (zamknął monitor).

Dowolne zakończenie wykonywania metody synchronizowanej lub instrukcji *synchronized* zwalnia monitor, dając czekającym wątkom możliwość dostępu do obiektu.

Koordinacja wątków

Koordinacja wątków polega na zapewnieniu właściwej kolejności działań wykonywanych przez różne wątki na wspólnym zasobie. Do koordynacji wątków stosuje się następujące metody: *wait()*, *notify()*, *notifyAll()*.

Metoda `wait()`

```
public final void wait();
```

```
public final void wait(long timeout);
```

```
public final void wait(long timeout, int nanos)
```

```
throws InterruptedException
```

Wykonanie metody powoduje zawieszenie bieżącego wątku do czasu gdy inny wątek nie wykona metody `notify()` lub `notifyAll()` odnoszącej się do wątku, który wykonał `wait()`. Wątek wykonujący `wait(...)` musi być w posiadaniu monitora dotyczącego synchronizowanego obiektu. Wykonanie `wait(...)` powoduje zwolnienie monitora.

Metoda *notify()*

```
public final void notify();
```

Metoda powoduje odblokowanie jednego z wątków zablokowanych na monitorze pewnego obiektu poprzez *wait()*. Który z czekających wątków będzie odblokowany nie jest w definicji metody określone.

Odblokowany wątek nie będzie natychmiast wykonywany – musi on jeszcze poczekać aż zwolniona będzie przez bieżący wątek blokada monitora. Odblokowany wątek będzie konkurował z innymi o nabycie blokady monitora. Metoda może być wykonana tylko przez wątek, który jest właścicielem zajmuje monitor obiektu.

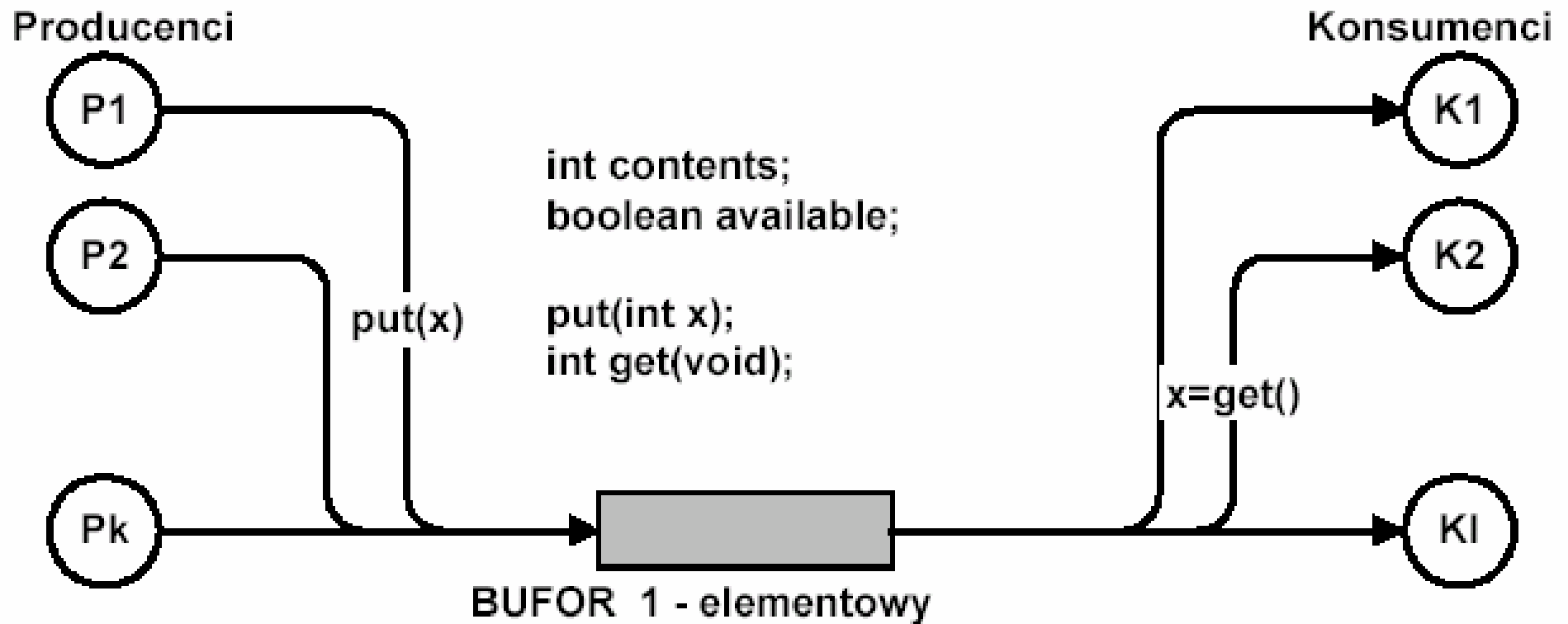
Metoda *notifyAll()*

```
public final void notifyAll()
```

Metoda powoduje odblokowanie wszystkich wątków zablokowanych na monitorze pewnego obiektu poprzez uprzednie wykonanie *wait()*.

Wątki będą jednak czekały aż wątek bieżący nie zwolni blokady monitora. Odblokowane wątki będą konkurowały o nabycie blokady monitora.

Przykład koordynacji: Problem producent - konsument.



Kilku producentów i konsumentów korzysta ze wspólnego zasobu jakim jest bufor.

- Każdy producent co pewien czas generuje liczby i umieszcza je w buforze.
- Każdy konsument co pewien czas pobiera liczbę z bufora i wyświetla ją na ekranie.

Przykład koordynacji: Problem producent - konsument.

Ograniczenia:

1. Z bufora może korzystać w każdej chwili tylko jeden producent lub konsument,
2. Producent może umieścić liczbę w buforze tylko wówczas, gdy bufor jest pusty.
W przeciwnym wypadku Producent musi czekać, aż konsument zwolni miejsce w buforze.
3. Konsument może pobrać liczbę z bufra tylko wtedy, gdy bufor nie jest pusty.
W przeciwnym wypadku konsument musi czekać aż jakiś producent umieści w buforze liczbę.

Klasa Producent

```
public class Producent extends Thread
{
    private Bufor buf;
    private int number;

    public Producent(Bufor c, int number)
    {
        buf = c;
        this.number = number;
    }

    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            buf.put(i);
            System.out.println("Producent #" +
                this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Klasa Konsument

```
public class Konsument extends Thread
{
    private Bufor buf;
    private int number;

    public Konsument(Bufor c, int number)
    {
        buf = c;
        this.number = number;
    }

    public void run()
    {
        int value = 0;
        for (int i = 0; i < 10; i++)
        {
            value = buf.get();
            System.out.println("Konsument #" +
                this.number + " got: " + value);
        }
    }
}
```

Klasa *ProducentKonsumentTest*

```
public class ProducentConsumentTest
{
    public static void main(String[] args)
    {
        Bufor c = new Bufor();

        Producent p1 = new Producent(c, 1);
        Konsument c1 = new Konsument(c, 1);

        p1.start();
        c1.start();
    }
}
```

Klasa *Bufor* – metoda *get()*

```
public class Bufor
{
    private int contents;
    private boolean available = false;

    public synchronized int get()
    {
        while (available == false)
        {
            try { wait();
                } catch (InterruptedException e) { }
        }
        available = false;

        notifyAll();

        return contents;
    }
}
```

Klasa *Bufor* – metoda *put()*

```
public synchronized void put(int value)
{
    while (available == true)
    {
        try { wait();
            } catch (InterruptedException e) { }
    }

    contents = value;
    available = true;

    notifyAll();
}
}
```