



**Faculty
of Physics**

WARSAW UNIVERSITY OF TECHNOLOGY



Programowanie Obiektowe Java

Małgorzata Janik

Zakład Fizyki Jądrowej
malgorzata.janik@pw.edu.pl
<http://java.fizyka.pw.edu.pl/>



WDI – 27-28 marca

<https://www.warszawskiedniinformatyki.pl/>

(wtorek, środa)



Zarejestruj się na największe wydarzenie IT w Polsce dla Studentów i Profesjonalistów. #WDI18

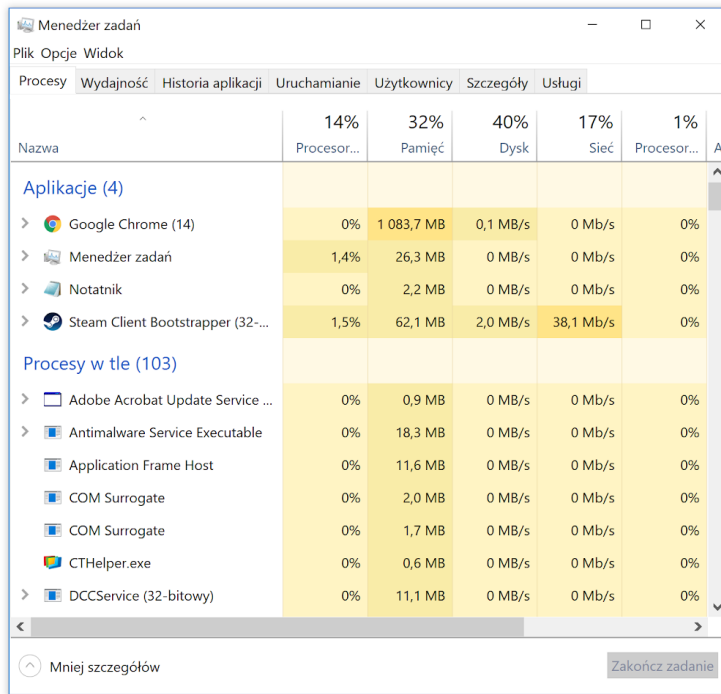
27-28 marca 2018

ZAREJESTRUJ SIĘ

Uczestnictwo jest bezpłatne. Jeśli posiadasz już konto w WDI18: [Zaloguj się](#)

Wprowadzenie do programowania współbieżnego

Procesy

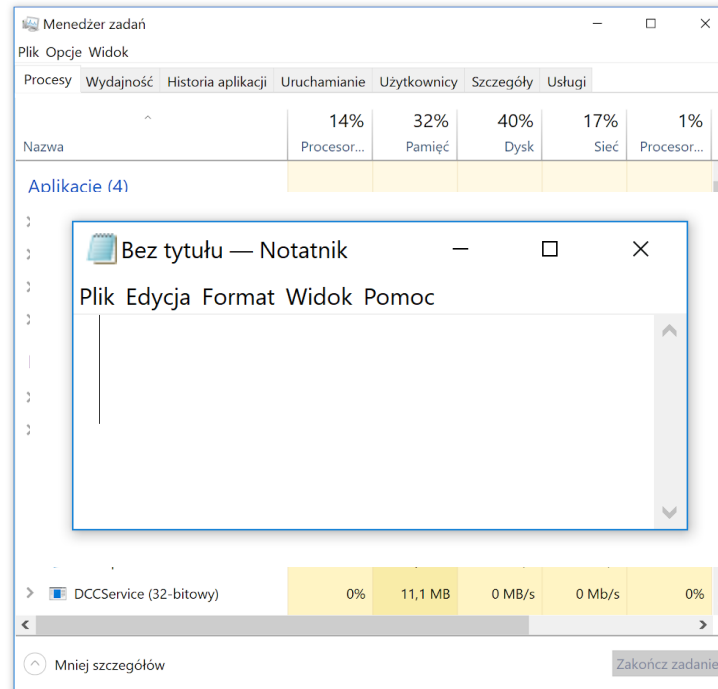


PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5963	wfpw	20	0	43164	3968	3320	R	0,7	0,0	0:00.05	top
7	root	20	0	0	0	0	S	0,3	0,0	0:18.02	rcu_sched
1446	wfpw	20	0	1769080	344744	91980	S	0,3	1,0	2:45.19	cinnamon
2478	wfpw	20	0	917364	120096	64676	S	0,3	0,4	0:03.02	chrome
2646	wfpw	20	0	8135984	610848	72352	S	0,3	1,9	3:31.60	java
4923	root	20	0	0	0	0	S	0,3	0,0	0:00.90	kworker/1:2
5579	wfpw	20	0	1471312	506752	92452	S	0,3	1,5	0:11.30	chrome
5646	wfpw	20	0	532816	35896	28340	S	0,3	0,1	0:00.24	gnome-term+
1	root	20	0	119920	6088	4000	S	0,0	0,0	0:01.43	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0,0	0,0	0:00.03	ksoftirqd/0
5	root	0	-20	0	0	0	S	0,0	0,0	0:00.00	kworker/0:+

- **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.



Procesy



- **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. → Równocześnie działa nam firefox, notatnik i Steam



Wątki

- **Wątek** to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)
 - Każdy proces ma co najmniej jeden wykonujący się wątek.



Wątki

- **Wątek** to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)
 - Każdy proces ma co najmniej jeden wykonujący się wątek.
 - Ciąg instrukcji zapisanych w metodzie main(...) definiuje instrukcje, które będą wykonane w ramach **tzw. głównego wątku**. Uruchomienie programu to de facto uruchomienie głównego wątku.
 - Dotychczasowe programowanie polegało na określaniu **instrukcji, które mają się wykonywać jedna po drugiej**. Pierwszą instrukcją programu jest pierwsza instrukcja metody głównej main(...). Następnie wszystkie instrukcje naszego programu są wykonane w jednej sekwencji.



Wątki

- **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 „równolegle” wiele wątków, które wykonują się w jednej przestrzeni adresowej procesu.



Modyfikatory dostępu

- Czy na jednordzeniowym komputerze program może działać wielowątkowo?

Tak

Nie



Modyfikatory dostępu

- Czy na jednordzeniowym komputerze program może działać wielowątkowo?

Tak !

Nie



Równoległość

- Równoległość działania wątków w systemie operacyjnym osiągana jest przez dwa mechanizmy:
 - **Faktyczną wielowątkowość** (uruchomienie programu na kilku rdzeniach procesora jednocześnie)
 - **Przydzielanie 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 zwykle następuje na zasadzie **wywłaszczania** (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.



Wątki

- **Wątek** to sekwencja działań, która wykonuje się w kontekście danego procesu (programu)
 - Każdy proces ma co najmniej jeden wykonujący się wątek.



Wątki - po co?

- Tworząc aplikację z interfejsem użytkownika łatwo spotkać się z sytuacją, w której pewna czynność
 - na przykład obliczenie wyniku skomplikowanej funkcji, czy pobranie pewnych danych z bazy danych

zabiera dużo czasu, a przez to aplikacja sprawia wrażenie jakby się zawiesiła.

- W przypadku **jednego wątku** – jeśli wchodzimy do funkcji obliczającej skomplikowane równanie, cały **interfejs użytkownika jest zamrażany aż do momentu skończenia obliczeń.**
 - **Rozwiązanie: wiele wątków.**



Wątki - po co?

- Jeśli obliczenia uruchomimy w wątku niezależnym od interfejsu, zarówno interfejs jak i wątek obliczeniowy będą naprzemiennie otrzymywały krótki czas procesora
 - będą sprawiały wrażenie wykonywania się równoległego (a w przypadku procesora wielordzeniowego faktycznie mogą wykonywać się równolegle)

dzięki czemu użytkownik aplikacji będzie miał lepsze odczucia w związku z jej użytkowaniem.

- Wątki pozwalają również na symultaniczne wykonywanie części operacji dzięki czemu czas wykonania pewnych operacji można znacząco skrócić.



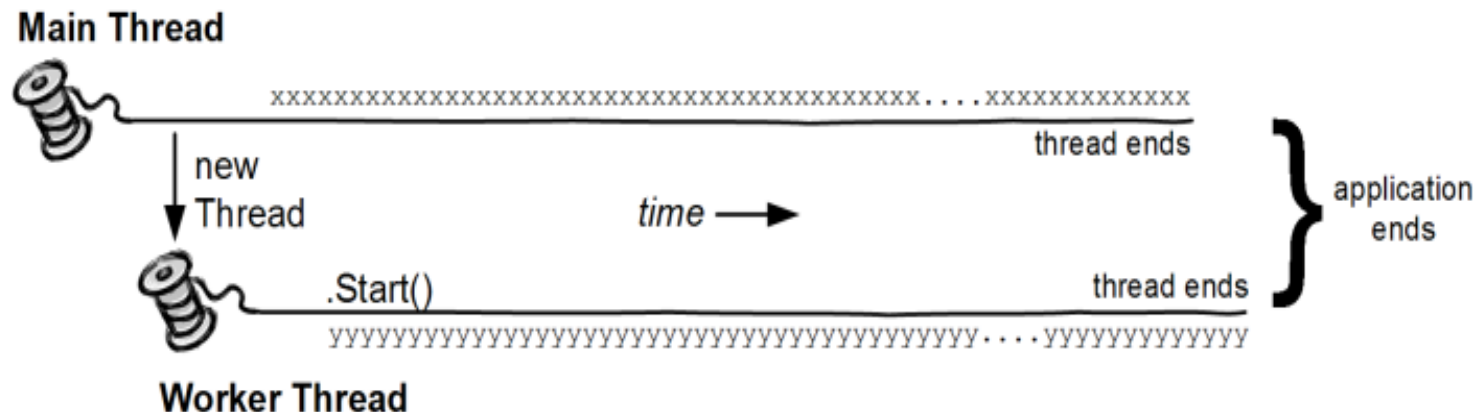
Kiedy wykorzystywać wątki?

- Jeśli chcemy aby jakiś inny kod (ciąg instrukcji) wykonywał się niezależnie od kodu wątku głównego
- W wielu sytuacjach:
 - Wszelkie obliczenia, które mogą zablokować interfejs użytkownika powinny być wykonywane asynchronicznie
 - Animacje, które powinny być przetwarzane niezależnie od interfejsu użytkownika
 - Pobieranie danych z internetu (zamiast przetwarzać strony internetowe jedna po drugiej można połączyć się np. z 10 jednocześnie)
 - W ogólności wszystkie operacje wejścia/wyjścia, zapis i odczyt plików, czy baz danych
 - Złożone obliczenia, które mogą być podzielone na mniejsze podzadania
 - I wiele innych



Tworzenie wątków

- Jeśli chcemy aby jakiś inny kod (ciąg instrukcji) wykonywał się niezależnie od kodu wątku głównego, tj. współbieżnie z nim, to
 - musimy określić jaki to kod,
 - musimy go uruchomić.



Tworzenie wątków

- Jeśli chcemy aby jakiś inny kod (ciąg instrukcji) wykonywał się niezależnie od kodu wątku głównego, tj. współbieżnie z nim, to
 - musimy określić jaki to kod,
 - musimy go uruchomić.
- Kod wątku głównego określamy implementując metodę `main(...)`
- Kod innych wątków określamy implementując dowolną klasę dziedziczącą z klasy `java.lang.Thread` albo klasę która implementuje interfejs `java.lang.Runnable`.
 - Mamy więc dwie możliwości, jednak w obydwu przypadkach implementacja wątku sprowadza się do implementacji metody `run()`.



Tworzenie wątków

Nowy wątek



Klasa
Thread

Interfejs
Runnable

lub interfejs
Callable...

Tworzenie wątków

Nowy wątek



Klasa
Thread

Interfejs
Runnable

lub interfejs
Callable...

Tworzenie i uruchamianie wątków

- Klasa **Thread** to podstawowa klasa pozwalająca na tworzenie, uruchamianie i zarządzanie wątkami.

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

- Oprócz szeregu metod służących do zarządzania wątkami, klasa **Thread** implementuje interfejs **Runnable**, zawierający jedną metodę **run()** – która jest wykonywana w momencie uruchamiania wątku

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

<http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>



Klasa Thread

Definiowanie klasy dziedziczącej po klasie **Thread**:

```
class MyThread extends Thread {  
    . . .  
    public void run() {  
        // kod do wykonania  
    }  
}
```

(1)

Utworzyć klasę dziedziczącą
po Thread i napisać
dla niej metodę run()



Klasa Thread

Definiowanie klasy dziedziczącej po klasie **Thread**:

```
class MyThread extends Thread {  
    . . .  
    public void run() {  
        // kod do wykonania  
    }  
}
```

(1) Utworzyć klasę dziedziczącą po Thread i napisać dla niej metodę run()

Tworzenie i uruchamianie wątku:

```
MyThread t = new MyThread();  
t.start();
```

(2)
Utworzyć obiekt nowej klasy

(3)
Wywołać metodę start.



Klasa Thread

Definiowanie klasy dziedziczącej po klasie **Thread**:

```
class MyThread extends Thread {  
    . . .  
    public void run() {  
        // kod do wykonania  
    }  
}
```

(1) Utworzyć klasę dziedziczącą po Thread i napisać dla niej metodę run()

Tworzenie i uruchamianie wątku:

```
MyThread t = new MyThread();  
t.start();
```

(2) Utworzyć obiekt nowej klasy

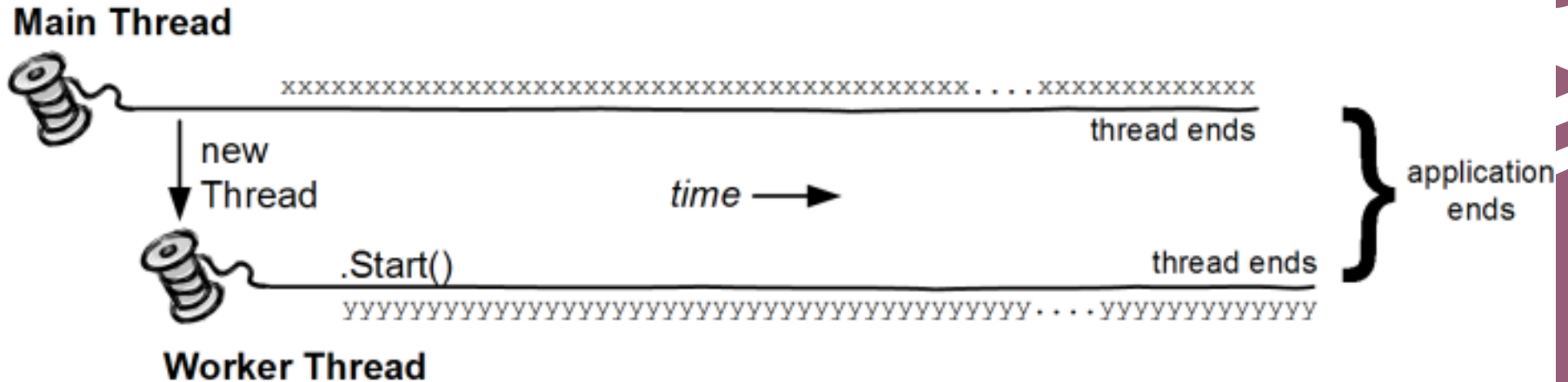
(3) Wywołać metodę start.

- Metoda **run()** nie jest wywoływana jawnie, lecz pośrednio poprzez metodę **start()**.
- Użycie metody **start()** powoduje wykonanie działań zawartych w ciele metody **run()**.
- Jeśli w międzyczasie nie zostanie przerwane zadanie, w ciele którego dany wątek działa, to końcem życia wątku będzie koniec działania metody **run()**.



Przykład

WatkiXxxYyy.java



(1) Utworzyć klasę dziedziczącą po Thread i napisać dla niej metodę run() w której wypiszemy 1000 razy 'y'

(2) Utworzyć obiekt nowej klasy

(3) Wywołać metodę start.

W głównym wątku wypiszemy 1000 razy 'x'



Przykład

WatkiXxxYyy.java

```
class ThreadY extends Thread {  
  
    public void run() {  
        for (int i = 0; i < 1000; i++)  
            System.out.print("y");  
    }  
}
```

(1) Utworzyć klasę dziedziczącą po Thread i napisać dla niej metodę run()

```
public class WatkiXxxYyy {
```

```
    public static void main(String[] args) {
```

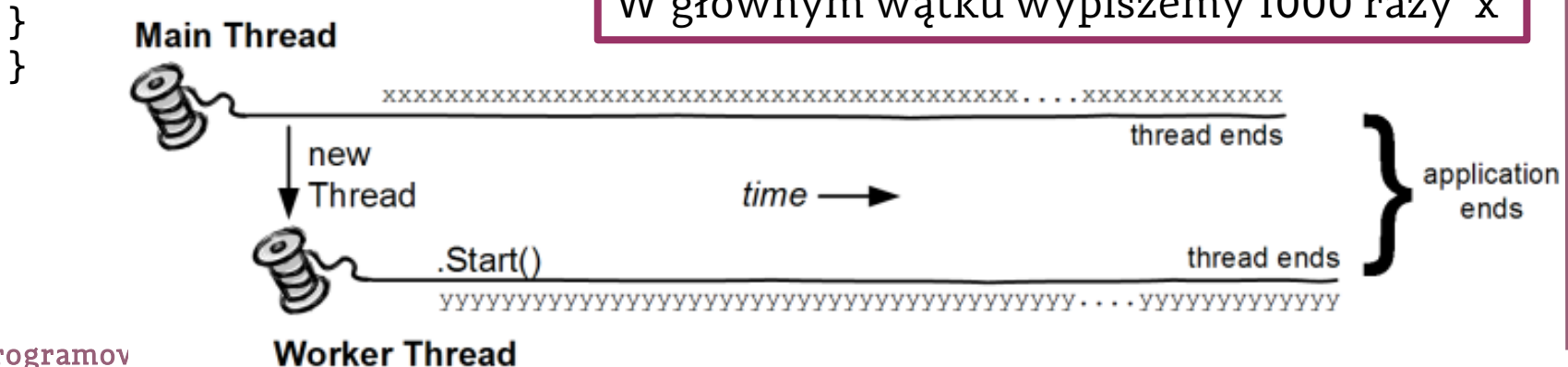
```
        ThreadY threadPrintY = new ThreadY();  
        threadPrintY.start();
```

(2) Utworzyć obiekt nowej klasy

(3) Wywołać metodę start.

```
        for (int i = 0; i < 1000; i++)  
            System.out.print("x");
```

W głównym wątku wypiszemy 1000 razy 'x'



WatkiXxxYyy.java

Wątki na zmianę dostają
swoją część czasu procesora

[illegible]

Główne metody klasy Thread

- Uruchamianie i zatrzymywanie wątków:
 - **start** - uruchomienie wątku,
 - ~~stop~~ - zakończenie wątku (metoda niezalecana),
 - **run** - kod wykonywany w ramach wątku.
- 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.
- Priorytety i szeregowanie wątków:
 - **getPriority** - odczytanie priorytetu wątku,
 - **setPriority** - ustawienie priorytetu wątku,
 - **yield** - wywołanie szeregowania.



Główne metody klasy Thread c.d.

- Synchronizacja wątków:
 - **sleep** - zawieszenie wykonania wątku na dany okres czasu,
 - **join** - czekanie na zakończenie innego wątku,
 - **wait** - czekanie na ryglu (lock),
 - **notify** - odblokowanie wątku zablokowanego na ryglu (lock),
 - **notifyAll** - odblokowanie wszystkich wątków zablokowanych na ryglu (lock)
 - **interrupt** - odblokowanie zawieszonego wątku,
 - **suspend** - zablokowanie wątku,
 - **resume** - odblokowanie wątku zawieszonego przez suspend



ThreadExample.java

```
class MyThread extends Thread{

    final int threadId;

    public MyThread(int threadId) {
        super();
        this.threadId = threadId;
    }

    @Override
    public void run() {
        for(int ii = 0; ii < 10; ii++){
            System.out.println("Thread " + threadId + " "
                + Thread.currentThread().getName() + " prints " +
                ThreadExample.nextNumber());
        }
    }
}
```



ThreadExample.java

```
public class ThreadExample {
    static int currentInt = 0;

    public static void main(String[] args) {
        int nthreads = 5;
        Thread[] threads = new Thread[nthreads];

        for(int ii = 0; ii < nthreads; ii++){
            threads[ii] = new MyThread(ii);
            threads[ii].setName("Watek-" + ii);
        }

        for(int ii = 0; ii < nthreads; ii++){
            threads[ii].start();
        }

        static int nextNumber(){
            currentInt++;    return currentInt;
        }
    }
}
```



ThreadExample.java

```
public class ThreadExample {
    static int currentInt = 0

    public static void main(S
        int nthreads = 5;
        Thread[] threads = new

        for(int ii = 0; ii < n
            threads[ii] = new I
            threads[ii].setName
        }

        for(int ii = 0; ii < n
            threads[ii].start(
        }
    }

    static int nextNumber(){
        currentInt++;
    }
}
```

```
Thread 1 Watek-1 prints 1
Thread 3 Watek-3 prints 4
Thread 2 Watek-2 prints 3
Thread 0 Watek-0 prints 2
Thread 2 Watek-2 prints 8
Thread 4 Watek-4 prints 7
Thread 3 Watek-3 prints 6
Thread 1 Watek-1 prints 5
Thread 3 Watek-3 prints 12
Thread 4 Watek-4 prints 11
Thread 4 Watek-4 prints 15
Thread 2 Watek-2 prints 10
Thread 0 Watek-0 prints 9
Thread 2 Watek-2 prints 17
Thread 4 Watek-4 prints 16
Thread 3 Watek-3 prints 14
Thread 1 Watek-1 prints 13
Thread 3 Watek-3 prints 21
Thread 4 Watek-4 prints 20
Thread 2 Watek-2 prints 19
Thread 0 Watek-0 prints 18
```

Koniec pracy wątku

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

```
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");
    }
}
```

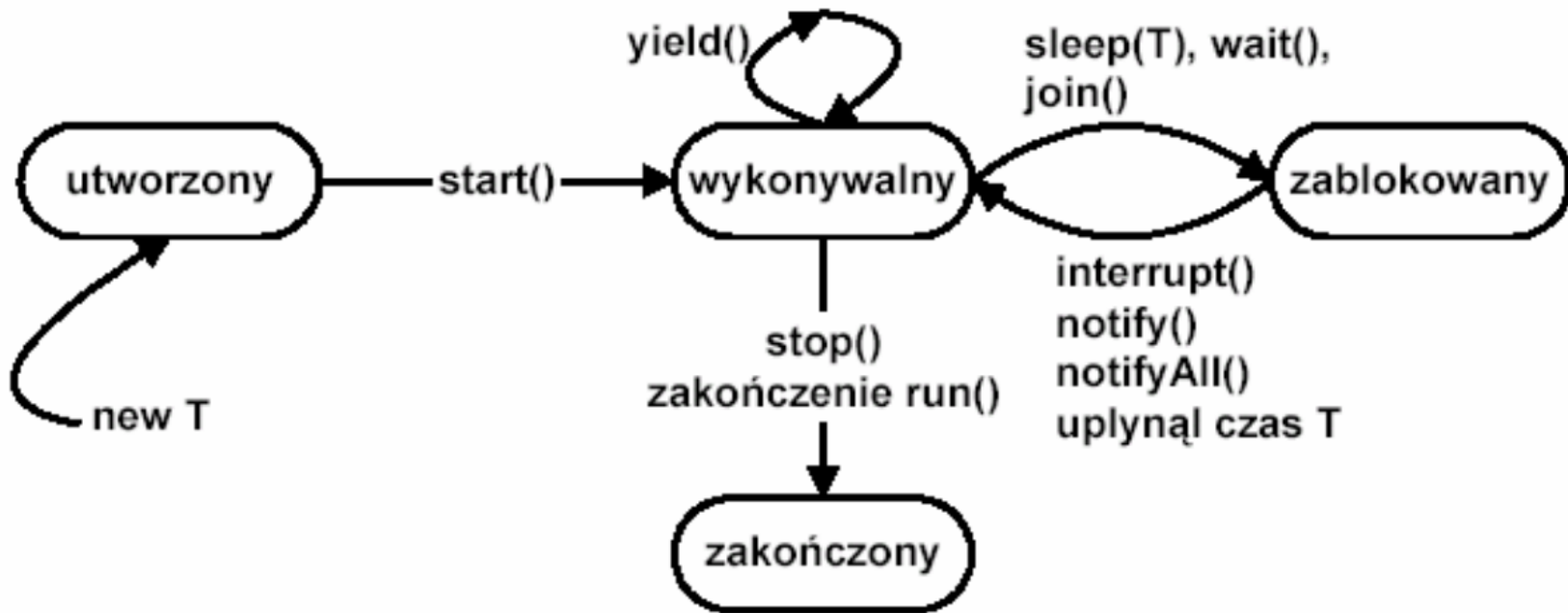
testowanie warunku zakończenia wątku

wymuszenie warunku zakończenia wątku

Źródło / na podstawie: Paweł Rogaliński, Język Java – wątki, Instytut Automatyki i Robotyki Politechnika Wrocławska



Stany wątków



Wyścigi

- Jaką wartość zwróci metoda `balance()`?

```
class Balance {  
    private int number = 0;  
  
    public int balance() {  
        number++;  
        number--;  
        return number;  
    }  
}
```

Jeśli wywołamy ją:

(1) w pojedynczym wątku, np.
w `main(..)`

(2) w dwóch wątkach na raz



Wyścigi

- Jaką wartość zwróci metoda `balance()`?

```
class Balance {  
    private int number = 0;  
  
    public int balance() {  
        number++;  
        number--;  
        return number;  
    }  
}
```

Wydaje się, że jakiegokolwiek wielokrotne wywoływanie metody `balance()` na rzecz dowolnego obiektu klasy `Balance` zawsze zwróci wartość 0.



Wyścigi

- Jaką wartość zwróci metoda `balance()`?

```
class Balance {  
    private int number = 0;  
  
    public int balance() {  
        number++;  
        number--;  
        return number;  
    }  
}
```

W świecie programowania
współbieżnego nie jest to
wcale takie oczywiste!

Więcej: wynik różny od 0
może pojawiać się nader
często!



Wyścigi

- 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



Wyścigi

- 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>	
1	<i> number--;</i>	
0	<i> return number;</i>	
0	<i>}</i>	
0	...	
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

wyłączenie
pierwszego
wątku

zwróci
wartość 0



Wyścigi

- 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>	
1		...
1		<i>balance(){</i>
1		<i>number++;</i>
2		<i>number--;</i>
1		<i>return number;</i>
1		}
1		...
1	<i>number--;</i>	
0	<i>return number;</i>	
0	}	
...		

wyłączenie pierwszego wątku

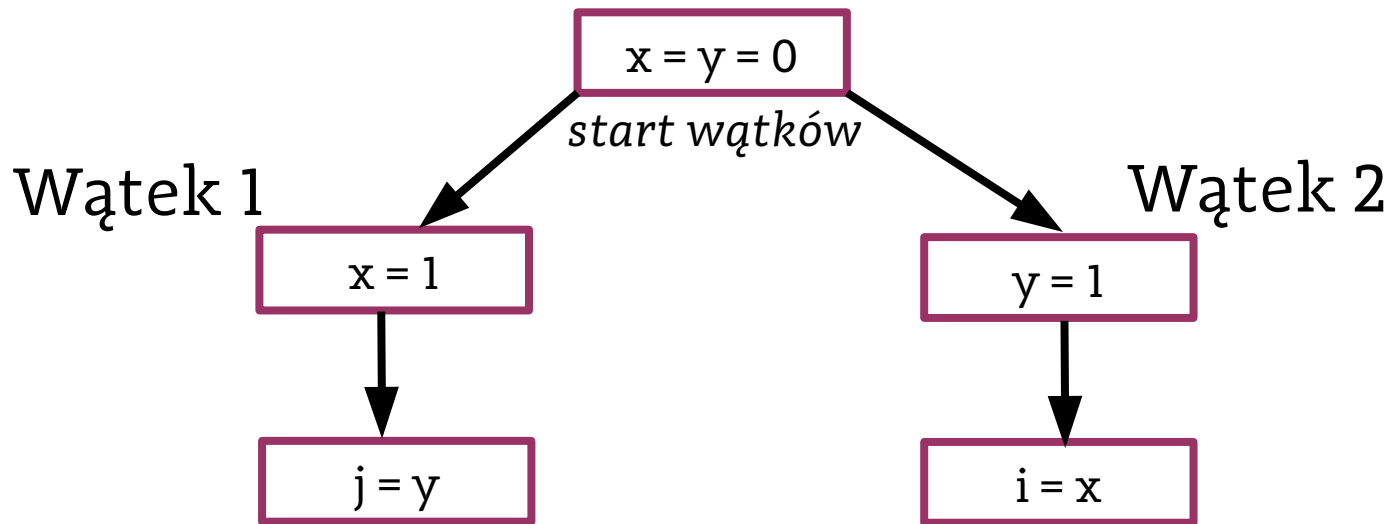
zwróci wartość 1

wyłączenie drugiego wątku

zwróci wartość 0



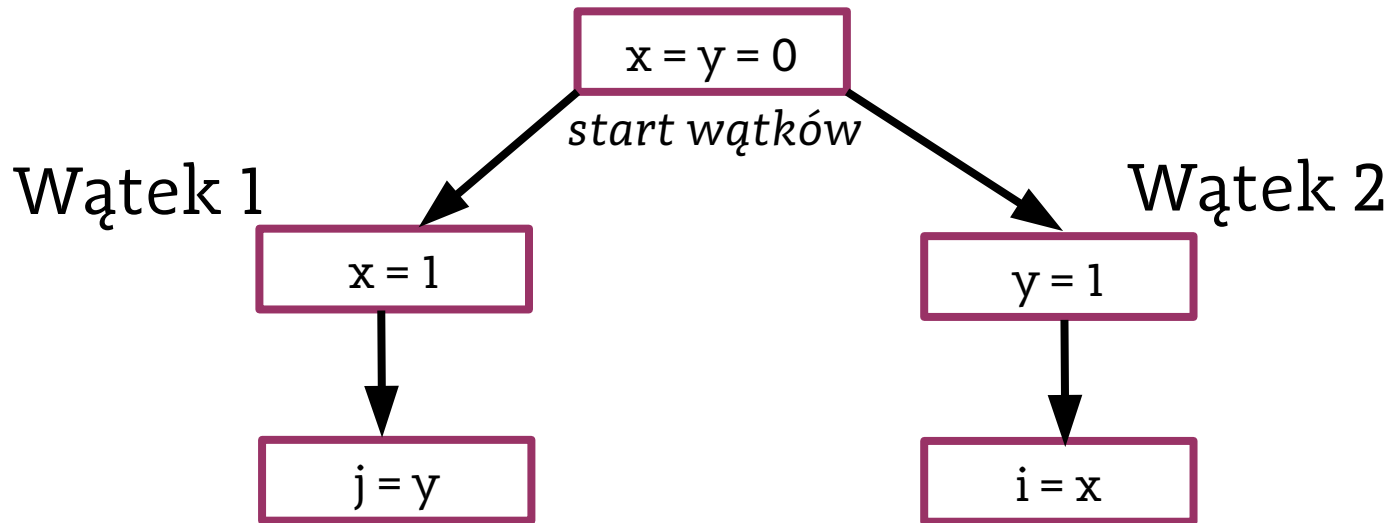
Wyścigi, wersja zaawansowana



Jakie mamy możliwe wartości dla i oraz j ?

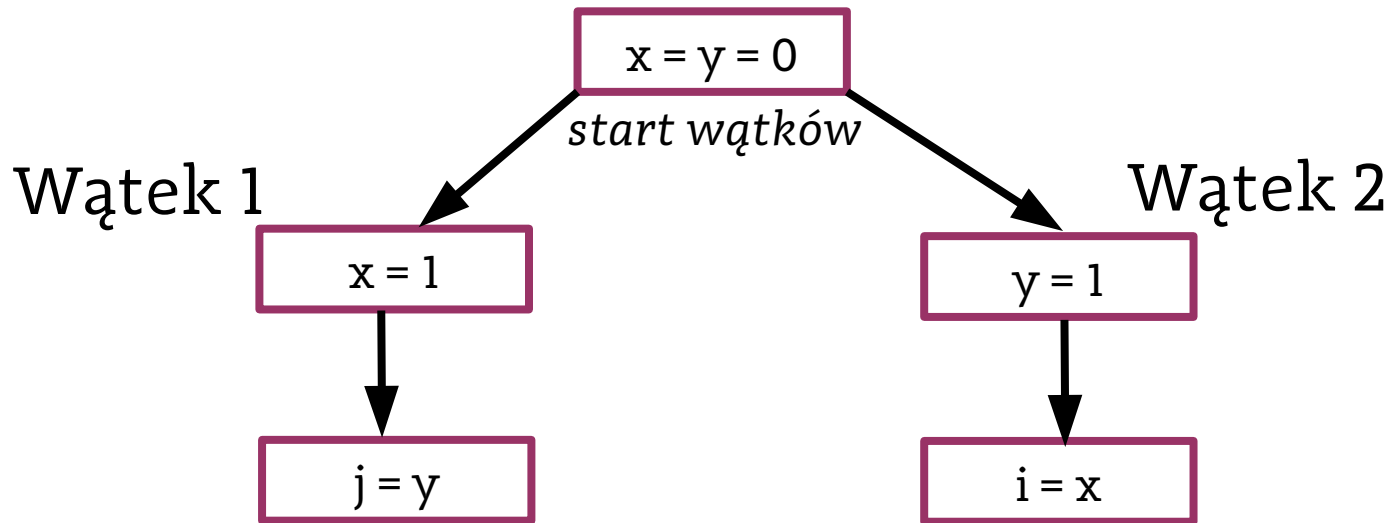


Wyścigi, wersja zaawansowana



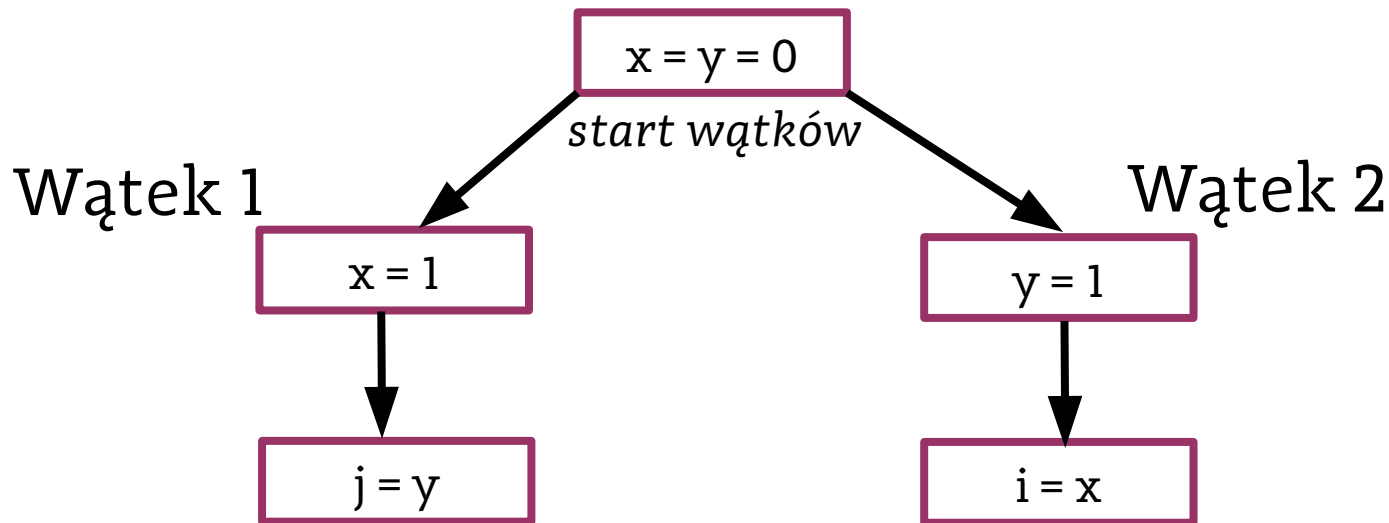
Instrukcja 1	Instrukcja 2	Instrukcja 3	Instrukcja 4	Wynik
x = 1	j = y	y = 1	i = x	j = 0, i = 1
x = 1	y = 1	j = y	i = x	j = 1, i = 1
x = 1	y = 1	i = x	j = y	j = 1, i = 1

Wyścigi, wersja zaawansowana



Instrukcja 1	Instrukcja 2	Instrukcja 3	Instrukcja 4	Wynik
y = 1	i = x	x = 1	j = y	j = 1, i = 0
y = 1	x = 1	i = x	j = y	j = 1, i = 1
y = 1	x = 1	j = y	i = x	j = 1, i = 1

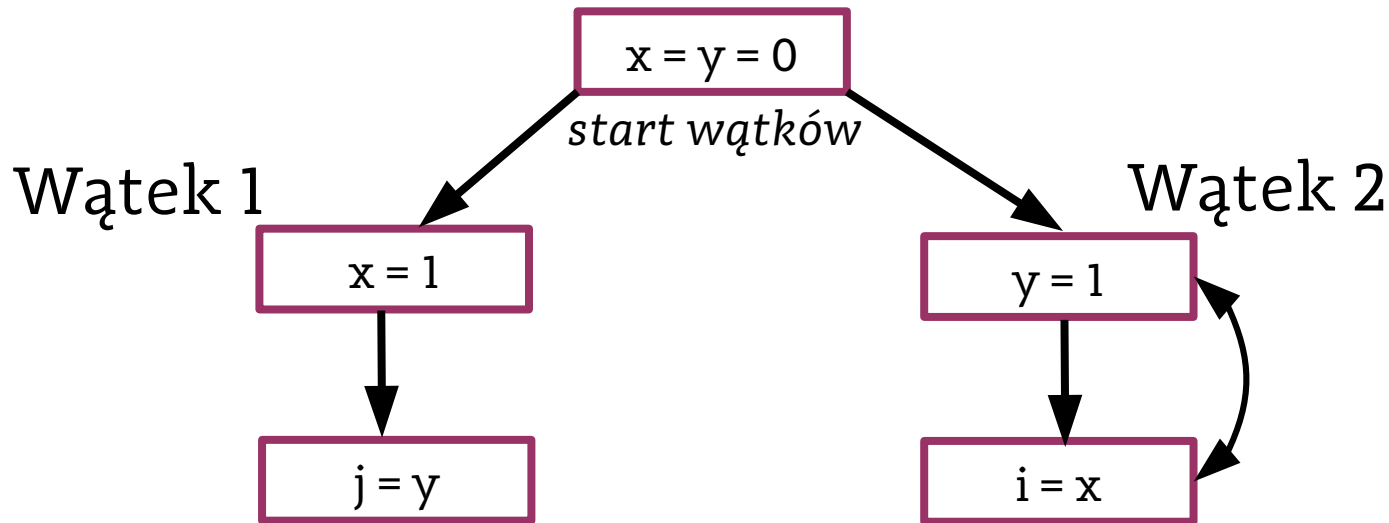
Wyścigi, wersja zaawansowana



Czy wynikiem może być **i=0** i **j=0**?



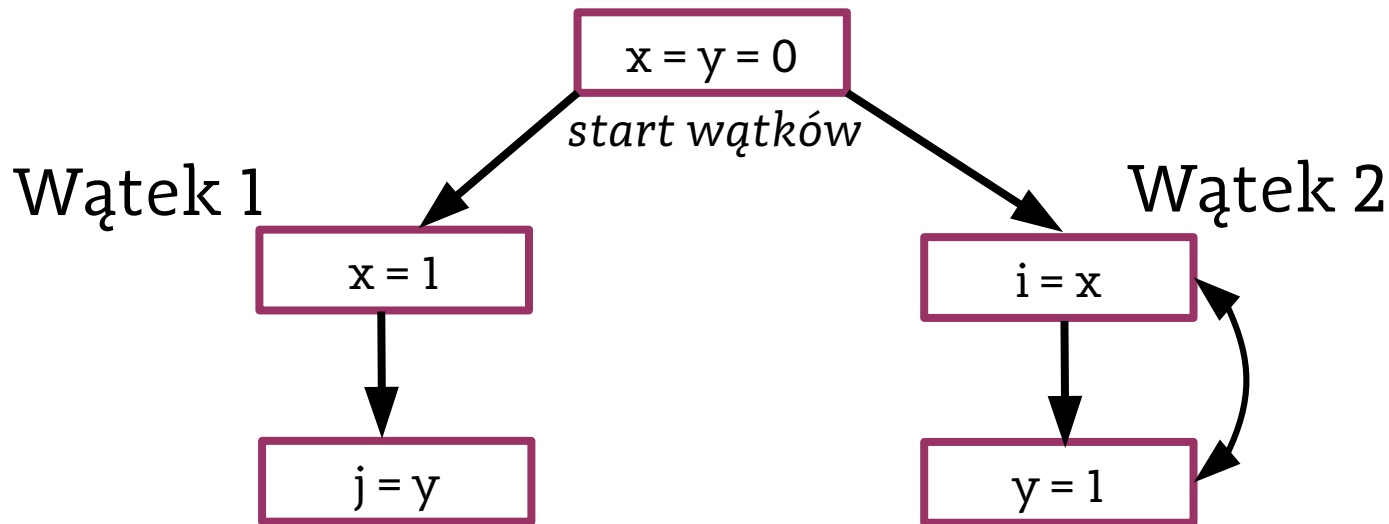
Wyścigi, wersja zaawansowana



Czy wynikiem może być **i=0** i **j=0**?

Jeśli dwie instrukcje bazują na niezależnych zmiennych kompilator może zmieniać ich kolejność!
 Agresywne optymalizacje: dobre dla wydajności, złe dla naszej intuicji.

Wyścigi, wersja zaawansowana



Czy wynikiem może być **i=0** i **j=0**?

Instrukcja 1	Instrukcja 2	Instrukcja 3	Instrukcja 4	Wynik
i = x	x = 1	j = y	y = 1	j = 0, i = 0

Wyścigi: efekt współdzielonej pamięci

- Zawsze musimy się liczyć z tym, że wątki operujące na współdzielonych zmiennych mogą być wywłaszczone 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.

Inny przykład *race conditions*: ThreadHell.java



Synchronizacja wątków

- Komunikacja między wątkami opiera się na wspólnej pamięci. W takim przypadku występuje zjawisko **wyścigów**.
- Wyścigi (ang. **race conditions**) występują, jeśli 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.
- By uniknąć równoczesnego działania wątków na tym samym obiekcie (co w sposób nieprzewidywalny ukształtować może jego stany) stosuje się **synchronizację**.



Synchronizacja wątków

- 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.
- Obiekty, wykluczają równoczesny dostęp do zasobów/wykonywania danego kodu przez kilka wątków nazywają się ogólnie **synchronizatorami** lub **muteksami** (od ang. **mutual-exclusion semaphore**). W Javie tą rolę pełnią **rygle** (ang. **lock**).
- 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.
- Do wersji 1.5 synchronizację wątków można było uzyskać wyłącznie za pomocą słowa kluczowego **synchronized**.



Synchronized

- Każdy egzemplarz klasy Object i jej podklas posiada rygiel (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--;  
}
```



Wyścigi

- Zmiany wartości zmiennej `number` gdy wykonywane są dwa wątki dla których metoda `balance` jest **synchronizowana**:

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

Może się również najpierw wykonać wątek 2, a dopiero potem wątek 1... Ale nigdy sobie nie przerwą.

zwróci
wartość 0

wyłączenie
pierwszego
wątku

zwróci
wartość 0



Synchronized

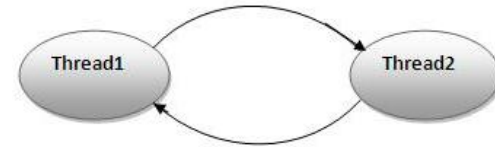
- Kiedy wątek wywołuje na rzecz jakiegoś obiektu metodę synchronizowaną, automatycznie zamykany jest rygiel (lock) (obiekt jest zajmowany przez wątek).
- Inne wątki usiłujące wywołać na rzecz tego obiektu metodę synchronizowaną (niekoniecznie tą samą) lub usiłujące wykonać instrukcję **synchronized**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



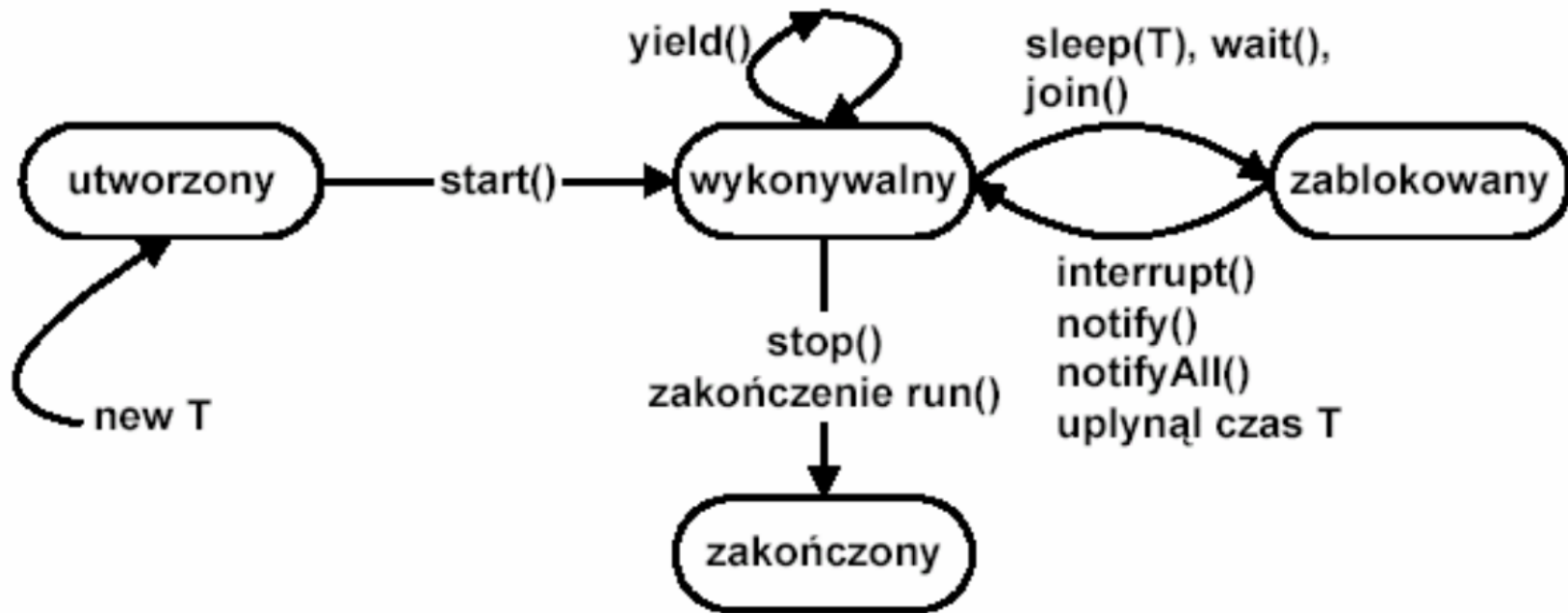
Deadlock

Deadlock.java

- **Deadlock** opisuje sytuację, gdzie dwa (lub więcej) wątki utknęły, bez końca oczekując na siebie wzajemnie.
- Przykład:
 - Wątek 1 potrzebuje dwóch rygli (R1 i R2) żeby wykonać konkretną instrukcję. Najpierw blokuje rygiel R1, następnie R2, a potem wykonuje instrukcje.
 - Wątek 2 również potrzebuje tych samych dwóch rygli (R1 i R2) żeby wykonać własną instrukcję. Najpierw blokuje rygiel R2, następnie R1, a potem wykonuje instrukcje.
 - Program może więc zrobić coś takiego:
 - **Thread 1: Blokuje lock R1...**
 - **Thread 2: Blokuje lock R2...**
 - **Thread 1: Czeką aż lock R2 się zwolni...**
 - **Thread 2: Czeką aż lock R1 się zwolni...**
 - I w ten sposób będą czekać na siebie w nieskończoność.
- Jak zaradzić? Zawsze brać rygle w tej samej kolejności we wszystkich wątkach!



Stany wątków



Stany wątków

- **Przejście od stanu wykonywalny do zablokowany następuje gdy:**
 - wywołano metodę `sleep(...)`
 - wątek chce wejść do zablokowanego rygla (`lock`)
 - wykonana została metoda `wait()`, `join()`, `suspend()`
 - wątek wykonał operację wejścia / wyjścia.
- **Powrót od stanu zablokowany do wykonywany następuje gdy:**
 - Gdy wątek zakończył wykonywanie metody `sleep()`
 - upłynął zadany interwał czasu.
 - rygiel (`lock`) został odblokowany
 - inny wątek wykonał operacja odblokowania zablokowanego wątku
 - wywołał metodę `notify()`, `notifyAll()`, `resume()`, `interrupt()`
 - Jeżeli wątek czekał na zakończenie operacji wejścia / wyjścia
 - operacja ta się zakończyła



Koordinacja wątków

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



join()

- Metoda **join** z klasy Thread powoduje oczekiwanie na zakończenie wątku, na rzecz którego została wywołana.
 - `thread1.join();`
// obecny proces czeka, aż thread1 się skończy
- Oczekiwanie może być przerwane, gdy dany wątek został przerwany przez inny wątek - wystąpi wyjątek `InterruptedException`.

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 watek nie wykona metody `notify()` lub `notifyAll()` odnoszącej się do tego wątku.
- `wait(...)` musi się odbyć w synchronizowanym bloku.
- Wykonanie `wait(...)` powoduje zwolnienie rygla.

```
synchronized(LOCK) {  
    LOCK.wait(); // LOCK został oddany  
}
```



notify()

- `public final void notify();`
- Metoda powoduje odblokowanie jednego z wątków zablokowanych na ryglu 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 rygla. Odblokowany wątek będzie konkutował z innymi o nabycie blokady rygla. Metoda może być wykonana tylko przez wątek, który jest właścicielem danego rygiela.

```
synchronized (LOCK)  
{ LOCK.notify(); }
```



notifyAll()

- `public final void notifyAll()`
- Metoda powoduje odblokowanie wszystkich wątków zablokowanych na ryglu pewnego obiektu poprzez uprzednie wykonanie `wait()`.
- Wątki będą jednak czekały aż wątek bieżący nie zwolni blokady rygla.
- Odblokowane wątki będą konkurowały o nabycie blokady rygla.

```
synchronized (LOCK)  
{ LOCK.notifyAll(); }
```



Wątki

Z punktu widzenia programisty wspólny dostęp wszystkich wątków jednego procesu do kontekstu tego procesu ma zarówno zalety jak i wady.

Zaletą jest możliwość łatwego dostępu do wspólnych danych programu.

Wadą jest brak ochrony danych programu przed równoległymi zmianami, dokonywanymi przez różne wątki, co może prowadzić do niespójności danych, a czego unikanie wiąże się z koniecznością synchronizacji działania wątków.

(więcej szczegółów nt. modelu pamięci w filmiku szkoleniowym Google: „Java Memory Model” :

<http://www.youtube.com/watch?v=WTVooKLLVT8>)



Tworzenie wątków

Nowy wątek



Klasa
Thread

Interfejs
Runnable

lub interfejs
Callable...

Interfejs Runnable

Inny sposób tworzenia wątków polega na implementacji do obiektu interfejsu **Runnable**:

1. Zdefiniować klasę implementującą interfejs Runnable
(np. `class Klasa implements Runnable`).
2. Zdefiniowanie metody `run ()`.
3. Utworzenie obiekt tej klasy
(np. `Klasa k = new Klasa();`)
4. Utworzenie obiektu klasy **Thread**, przekazując w konstruktorze referencję do obiektu utworzonego w kroku 3
(np. `Thread thread = new Thread(k);`)
5. Wywołać na rzecz nowoutworzonego obiektu klasy **Thread** metodę `start (thread.start();`)





(1) Utworzyć klasę implementującą Runnable i napisać dla niej metodę run() w której wypiszemy 1000 razy 'y'

(2a) Utworzyć obiekt nowej klasy. (2b) Utworzyć obiekt Thread przyjmujący klasę Runnable w konstruktorze.

(3) Wywołać metodę start dla Thread.

W głównym wątku wypiszemy 1000 razy 'x'

Przykład

WatkiXxxYyyRunnable.java

```
class RunnableY implements Runnable {
```

```
    public void run() {  
        for (int i = 0; i < 1000; i++)  
            System.out.print("y");  
    }  
}
```

(1) Utworzyć klasę implementującą Runnable i napisać dla niej metodę run() w której wypiszemy 1000 razy 'y'

```
public class WatkiXxxYyyRunnable {
```

```
    public static void main(String[] args) {
```

```
        RunnableY runnablePrintY = new RunnableY();  
        Thread thread = new Thread(runnablePrintY);  
        thread.start();
```

(2a) Utworzyć obiekt nowej klasy.
(2b) Utworzyć obiekt Thread przyjmujący klasę Runnable.

```
        for (int i = 0; i < 1000; i++)  
            System.out.print("x");  
    }  
}
```

(3) Wywołać metodę start dla Thread.

W głównym wątku wypiszemy 1000 razy 'x'



Unnablen
klasy.
d
e.
s.
W

Unnablen
klasy.
d
e.
s.
W

(1) Utworzyć klasę implementującą Runnable i napisać dla niej metodę run() w której wypiszemy 1000 razy 'y'

(2a) Utworzyć obiekt nowej klasy.
(2b) Utworzyć obiekt Thread przyjmujący klasę Runnable.

[illegible]

RunnableExample.java

```
class MyRunnableExample implements Runnable{

    final int threadId;

    public MyRunnableExample(int threadId) {
        super();
        this.threadId = threadId;
    }

    public void run() {
        for(int ii = 0; ii < 10; ii++){
            System.out.println("Thread " +
                Thread.currentThread().getName() + " prints " +
                RunnableExample.nextNumber());
        }
    }
}
```

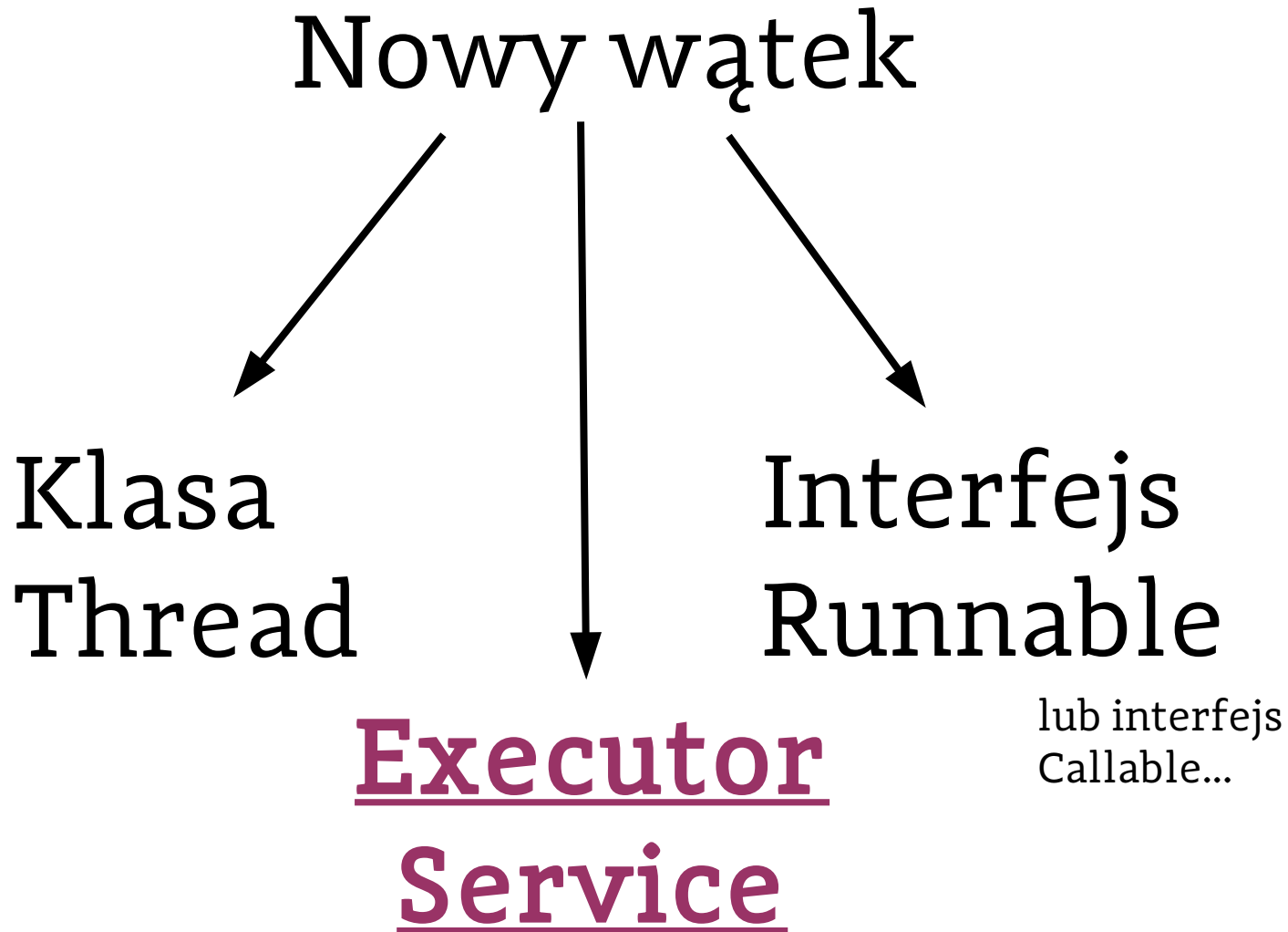


RunnableExample.java

```
public class RunnableExample {  
    static int currentInt = 0;  
  
    public static void main(String[] args) {  
        int nthreads = 5;  
        Thread[] threads = new Thread[nthreads];  
  
        for(int ii = 0; ii < nthreads; ii++){  
            threads[ii] = new Thread(new  
MyRunnableExample(ii)); }  
  
        for(int ii = 0; ii < nthreads; ii++){  
            threads[ii].start(); }  
  
    }  
  
    static int nextNumber(){  
        currentInt++;  
        return currentInt;  
    }  
}
```



Tworzenie wątków



„Wykonawcy” – Executors

- Można zauważyć, że
 - kod wątku zapisywany jest w metodzie `run()`
→ tam określamy wykonywane zadanie,
 - a klasa `Thread` tak naprawdę nic nie robi.
- Dotychczas trzeba było samemu uruchomić każdy pojedynczy wątek i dalej martwić się o jego los.
- Wolelibyśmy rozumować raczej w kategoriach zadań do wykonania, a nie technicznych szczegółów sposobu ich implementacji. Sposób: *ExecutorService*.
- Od Java 1.5 zaleca się uruchamiać wątki przy pomocy tzw. klas „wykonawców” (executors).
 - Pozwala łatwo tworzyć pule wątków i zarządzać nimi



„Wykonawcy” – Executors

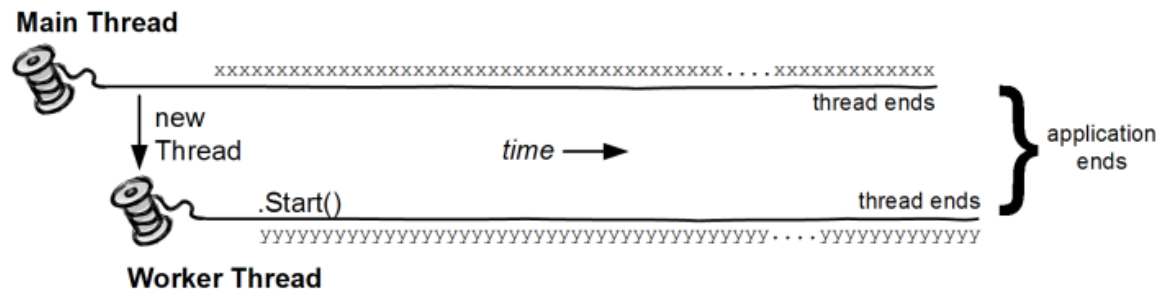
- Samodzielne zarządzanie wątkami może być czasem kłopotliwe, dlatego zaleca się uruchamiać wątki przy pomocy tzw. klas „wykonawców” (executors).
- Pozwalają one na odseparowanie zadań do wykonania od mechanizmów tworzenia i uruchamiania wątków.

```
ExecutorService exec =  
    Executors.newFixedThreadPool(2);  
        //Executors.newSingleThreadExecutor();  
  
exec.execute(b1);  
exec.execute(b2);  
// b1, b2 – obiekty implementujące Runnable  
  
exec.shutdown();
```



SingleThreadExecutor

Executors.newSingleThreadExecutor() - Wykonawca uruchamiający podane mu zadania w jednym wątku
= identycznie jak w przypadku z Runnable, tylko tworzymy obiekt ExecutorService przymujący obiekt z Runnable zamiast obiektu Thread.



Po co więc tworzyć wykonawcę zamiast wątku?

Jeśli tylko chcemy wystartować nowy wątek i o nim zapomnieć, to żadna różnica.

ALE

Wykonawcy mają dużo dodatkowych metod, które pozwalają odtworzyć wątki jeśli w przypadku wystąpienia wyjątku, lepiej sobie radzą z recyklingiem wątków, jeśli kiedyś byśmy potrzebowali coś rozbudować, byłoby wygodniej, itd... → to lepsza klasa.



SingleThreadExecutor

Executors.newSingleThreadExecutor() - Wykonawca uruchamiający podane mu zadania w jednym wątku
= identycznie jak w przypadku z Runnable, tylko tworzymy obiekt ExecutorService przymującą obiekt z Runnable zamiast obiektu Thread.

Tworzymy ExecutorService,
A następnie wywołujemy execute,
podając konkretny obiekt Runnable.

```
ExecutorService exec =  
Executors.newSingleThreadExecutor();  
  
exec.execute(b2);  
// b1 – obiekt implementujący Runnable  
  
exec.shutdown();
```

Z javadoc:

"An unused ExecutorService should be shut down to allow reclamation of its resources."



Pule wątków

Executors.newFixedThreadPool(int n) - Wykonawca, prowadzący pulę wątków o zadanych maksymalnych rozmiarach.

Pule wątków pozwalają na ponowne użycie wolnych wątków, a także na ew. limitowanie maksymalnej liczby wątków w puli.

```
ExecutorService exec =  
Executors.newFixedThreadPool(2);  
exec.execute(b1);  
exec.execute(b2);  
exec.execute(b3);  
exec.execute(b4);  
// b1, b2, b3, b4 – obiekty implementujące Runnable  
  
exec.shutdown();
```

Executor wykona cztery wątki,
ale nie więcej niż 2 na raz.



JButtonRunnable.java

```
public class JButtonRunnable extends JButton implements Runnable {
    (...)
    String[] tekst = {"To", "jest", "animowany", "przycisk"};
    (...)
    public void run() {
        int i = 0;
        while(czynny){

            if (i < tekst.length-1 ) i++; else i = 0;

            setText(tekst[i]);
            try {
                Thread.sleep(pauza);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
}
```



Klasa implementująca Runnable
Tak samo jak wcześniej



JButtonRunnable.java – metoda main()

```
JFrame f = new JFrame();  
f.setLayout(new GridLayout(2,1));  
f.setSize(200, 200);  
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
JButtonRunnable b1 = new JButtonRunnable();  
f.add(b1);
```

```
String[] innyTekst = {"inny", "tekst", "do", "anim",  
"przycisku"};
```

// wykorzystanie drugiego konstruktora pozwalajacego zmienic tekst i szybkość:

```
JButtonRunnable b2 = new JButtonRunnable(innyTekst, 1600);  
f.add(b2);
```

```
ExecutorService exec = Executors.newFixedThreadPool(2);
```

```
exec.execute(b1);  
exec.execute(b2);  
exec.shutdown();  
f.setVisible(true);
```

Różnica: zamiast tworzyć obiekt
Klasy Thread, tworzymy
obiekt ExecutorService



ScheduledExecutorService

ScheduledExecutorService - wykonawca zarządzający tworzeniem i wykonaniem wątków w określonym czasie lub z określoną periodycznością:

schedule(Runnable command, long delay, TimeUnit unit)

Jednokrotnie uruchomienie zadania po upływie czasu „delay”

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

Periodyczne wykonywanie zadania (z możliwym opóźnieniem startu), czas liczony między rozpoczęciem kolejnych iteracji

scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)

Periodyczne wykonywanie zadania (z możliwym opóźnieniem startu) - czas liczony od zakończenia jednej iteracji do rozpoczęcia następnej



ScheduledExecutorExample.java

```
import static java.util.concurrent.TimeUnit.*;
(...)

final ScheduledExecutorService scheduler =
    Executors.newScheduledThreadPool(2);

scheduler.scheduleAtFixedRate(r, 0, 1, SECONDS);

scheduler.scheduleWithFixedDelay( (new Runnable() {
    public void run() {
        System.out.println("Po 5 sekundach - potem co 3 sekundy");
    }
}), 5, 3, SECONDS);

import static java.util.concurrent.TimeUnit.*;
MILLISECONDS, NANOSECONDS...
```



ScheduledExecutorExample.java



```
scheduler.schedule(new Runnable() {  
    @Override  
    public void run() { System.out.println("Koniec  
                                     programu po 15 sekundach");  
        scheduler.shutdownNow();  
        System.exit(0);}  
}, 15, SECONDS);
```

JButtonScheduled.java

```
public class JButtonScheduled extends JButton
implements Runnable {

    String[] tekst = {"To", "jest", "przycisk",
        "animowany", "przez",
        ScheduledExecutorService"};
    int i = 0;

    public JButtonScheduled() {
        super();
    }

    public void run() {
        if (i < tekst.length-1 ) i++; else i = 0;
        setText(tekst[i]);
    }
}
```



JButtonScheduled.java

```
final ScheduledExecutorService scheduler =  
    Executors.newScheduledThreadPool(3);  
  
// zadanie powtarzane cyklicznie - czas liczony od  
uruchomienia poprzedniego wykonania takie zadanie moze byc  
przerwane jedynie przez anulowanie - patrz nizej  
final ScheduledFuture<?> sc1 =  
    scheduler.scheduleAtFixedRate(b1, 1000, 50, MILLISECONDS);  
  
// zadanie powtarzane - czas liczony od zakonczenia  
poprzedniego wykonania  
scheduler.scheduleWithFixedDelay(b2, 2, 1, SECONDS);  
  
// jednokrotne wywołanie metody run z zadany opoznieniem  
scheduler.schedule(b3, 5, SECONDS);  
  
//Anulowanie pierwszego watku po 15 sekundach  
scheduler.schedule (new Runnable() {  
    public void run() { sc1.cancel(true);}  
}, 15, SECONDS);
```



Więcej informacji

- Polsko-Japońska Wyższa Szkoła Technik Komputerowych
 - <http://edu.pjwstk.edu.pl/wyklady/zap/scb/W8/W8.htm>
 - <http://edu.pjwstk.edu.pl/wyklady/zap/scb/W9/W9.htm>
- Paweł Rogaliński, Język Java – wątki, Instytut Automatyki i Robotyki Politechnika Wrocławska
 - http://pawel.rogalinski.staff.iiar.pwr.wroc.pl/dydaktyka/INE2018L_JP3_Java/Java%20-%20watki.pdf
- Nauka Javy
 - <http://naukajavy.pl/kurs-jezyka-java/115-programowanie-wspolbiezne>
 - <http://naukajavy.pl/kurs-jezyka-java/116-definiowanie-watkow>
- Java Start
 - <https://javastart.pl/static/zaawansowane-programowanie/watki-wprowadzenie-i-przyklad/>
- Piotr Tokarski, Wątki w Javie,
 - <http://math.uni.lodz.pl/~kowalcr/PodstawyJava/WatkiWJavie.pdf>



Współbieżność w Swingu

- Uważne użycie współbieżności jest niezwykle ważne przy programowaniu interfejsu w Swingu.
- Dobrze użyty program używa współbieżności żeby stworzyć GUI, które nigdy się nie “zawiesza” - program zawsze będzie odpowiadać na interakcje z użytkownikiem, niezależnie od wykonywanych w tle obliczeń.
- Żeby dobrze używać współbieżności w Swingu, musimy zrozumieć w jaki sposób Swing radzi sobie z wątkami.
- W Swingu mamy poniższe rodzaje wątków:
 - Początkowe wątki, od nich rozpoczyna się wykonywanie kodu aplikacji.
 - **Event Dispatch Thread**, wątek zarządzający wszystkimi zdarzeniami. Cały kod który jest związany z GUI powinien się znaleźć w tym wątku.
 - Wątki typu “Worker” - wątki tła, w nich należy wykonywać wszystkie zadania intensywnie obliczeniowo lub czasowo.
- Nie musimy sami tworzyć każdego z tych wątków – dostarcza je Swing. Ale musimy potrafić je wykorzystać.



Początkowe wątki

- W prostych aplikacjach początkowym i jedynym wątkiem jest **metoda main** głównej klasy programu.
- W programach bazujących na Swingu początkowe wątki nie robią wiele. Od nich zaczyna się aplikacja i ich głównym zadaniem jest stworzenie obiektu typu `Runnable` który zainicjalizuje GUI i rozpocznie wykonywanie **event dispatch thread**.
- Tworzenie GUI należy wykonać poprzez wywołanie jednej z metod:
 - `javax.swing.SwingUtilities.invokeLater`
 - `javax.swing.SwingUtilities.invokeAndWait`.
 - → Metody te przyjmują jeden argument: obiekt `Runnable` który definiuje nowe zadanie. Różnica między nimi jest zasugerowana przez ich nazwę:
 - `InvokeLater` tworzy nowy wątek i sam kończy działanie
 - `InvokeAndWait` tworzy nowy wątek i czeka, aż ten się skończy, dopiero wtedy sam kończy działanie.

```
public static void main(String[] args) {  
    SwingUtilities.invokeLater(  
        new Runnable(){  
            public void run() {  
                NewGUI();  
            }  
        }  
    );  
}
```



Event Dispatch Thread

- W Java wątek odpowiadający za obsługę GUI nosi nazwę **Event Dispatch Thread** (EDT).
- Zajmują się on
 - obsługą kolejki zdarzeń i informowaniem o nich obiektów nasłuchujących (czyli Listenerów),
 - zarządza rozłożeniem komponentów, ich wyświetleniem, zmianą właściwości komponentów (np. dezaktywacja przycisku)
 - obsługą zadań.
- Zadaniami tymi powinny być tylko i wyłącznie krótkotrwałe procesy.



Event Dispatch Thread

- Domyślnie w EDT dzieją się:
 - Wszystkie eventy (jeśli programujecie Listenera, to wewnątrz actionPerformed jesteście w EDT)
 - Wywołania metod repaint(), revalidate(), invalidate()
 - Jeśli nie jesteście pewien, czy jesteście w EDT, użyj metody:

```
System.out.println("EDT: "+SwingUtilities.isEventDispatchThread());
```



Podstawowa zasada

- Rule of thumb
 - Wszystko, co dotyka w jakikolwiek sposób UI (Swinga) powinno się odbywać w EDT.

Wątki w Swingu

Wszystko = więc również tworzenie ramek.

Zaleca się, żeby każda aplikacja tworzyła i uruchamiała GUI poprzez metodę „invokeLater”, np..

```
public static void main(String[] args) {  
    SwingUtilities.invokeLater(new Runnable(){  
        public void run() {  
            JFrame f = new JFrame();  
            f.setSize(200,200);  
            f.setVisible(true);  
        }  
    });  
}
```

//lub w niemal równoważny sposób:

```
public static void main(String[] args) {  
    EventQueue.invokeLater(  
        new Runnable(){  
            public void run() {  
                JFrame f = new JFrame();  
                f.setSize(200,200);  
                f.setVisible(true);  
            }  
        }  
    });  
}
```


Ale wcześniej też działało...

- Bez względu na to: GUI należy inicjalizować w EDT.
- Dla prostych programów nadal będzie działać, jeśli tego nie zrobimy.
 - We wczesnych wersjach Javy w ogóle nie było takiego wymogu.
 - Dlatego ludzie często tego nie robią. Często brakuje również tej części kodu w tutorialach.
- Wcześniejsze przykłady na wykładach / laboratoriach nie inicjalizowały GUI w EDT, ze względu na:
 - Upraszczenie przykładów
 - Unikanie niezrozumiałego kodu na początku zajęć.
- Skomplikowany kod może wymagać tworzenia GUI w EDT. Należy zrobić to tak, jak pokazano na poprzednim slajdzie.
- Dodatkowy efekt **setVisible(true)**
 - Nawet jeśli jawnie nie zainicjalizujecie GUI w EDT, to dodatkowym efektem wywołania `setVisible(true)` jest wystartowanie wątku EDT, który przejmuje wykonywanie i monitorowanie interfejsu użytkownika.

• Więcej info: <https://www.lepoint.net/JavaBasics/gui/gui-commentary/guicom-main-thread.html>

• https://www.reddit.com/r/learnprogramming/comments/29ik8n/java_can_someone_explain/



Klasa javax.swing.Timer

- **Timer** pozwala uruchomić jedno lub kilka zdarzeń akcji (`ActionEvent`) z zadany opóźnieniem lub interwałem czasowym.
- Zdarzenia będą wykonywane w ustalonych interwałach w wątku EDT

```
timer = new Timer(speed, this);
timer.setInitialDelay(pause);
timer.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // kod wykonywany cyklicznie
    }
});
timer.start();
```



Klasa javax.swing.Timer

Przykład: NoClick.java

```
Timer timer = new Timer(2500, null);

timer.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {
        autoPress.doClick();
    }
});

timer.start();
```



Klasa `java.util.Timer`

- Działanie zbliżone do `scheduledExecutor` omawianego wcześniej...
- Przykład: `UtilTimer.java`

```
Timer timer = new Timer(true);
timer.scheduleAtFixedRate(new TimerTask() {
    // po owinięciu w w invokeLater:
    public void run() {
        autoPress.setBackground(new Color(rand.nextInt()));
    }
}, 250, 250 );
```



Blokowanie GUI

Umieszczanie zbyt długich zadań w **Event Dispatch Thread (EDT)** może skutecznie zablokować GUI.

przykład: **LongTaskInEDT.java**



SwingWorker

<http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html>

Klasa `SwingWorker` została zaprojektowana do szybkiego tworzenia wątków, które pracują równolegle do EDT.

```
public abstract class SwingWorker<T,V> extends  
Object implements RunnableFuture
```



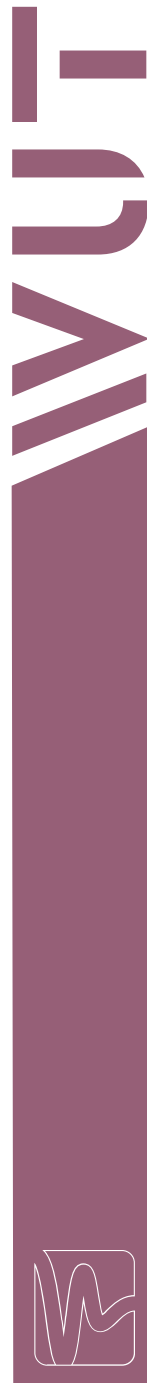
SwingWorker

<http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html>

Klasa `SwingWorker` została zaprojektowana do szybkiego tworzenia wątków, które pracują równolegle do EDT.

```
public abstract class SwingWorker<T,V> extends  
Object implements RunnableFuture
```

- `SwingWorker` to klasa abstrakcyjna, czyli możemy utworzyć obiekt tylko i wyłącznie klasy po niej dziedziczącej.
- Implementowany interfejs `RunnableFuture` jest połączeniem interfejsów `Runnable` i `Future` (z niego właśnie pochodzi metoda `get()` zwracająca rezultat obliczeń).



SwingWorker

<http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html>

Klasa `SwingWorker` została zaprojektowana do szybkiego tworzenia wątków, które pracują równolegle do EDT.

```
public abstract class SwingWorker<T, V> extends  
Object implements RunnableFuture
```

- `SwingWorker` to klasa abstrakcyjna, czyli możemy utworzyć obiekt tylko i wyłącznie klasy po niej dziedziczącej.
- Implementowany interfejs `RunnableFuture` jest połączeniem interfejsów `Runnable` i `Future` (z niego właśnie pochodzi metoda `get()` zwracająca rezultat obliczeń).
- **Parametr T** – określa typ zwracany przez metody `doInBackground()` i `get()`.
Jest to rezultat naszego zadania np. liczba / napis / obrazek.
- **Parametr V** – określa typ danych pośrednich, które może produkować zadanie, np. linie tekstu z wczytywanego pliku albo status postępu symulacji.
 - Dane te można uzyskać za pomocą metody `process(List dane)`,
 - A dane pochodzą z metody `publish(V... dane)` – która powinna być wywoływana w implementacji metody `doInBackground()`.



SwingWorker

<http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html>

Klasa `SwingWorker` została zaprojektowana do szybkiego tworzenia wątków, które pracują równolegle do EDT.

public abstract class SwingWorker<T, V> extends Object implements RunnableFuture

- `SwingWorker` to klasa abstrakcyjna, czyli możemy utworzyć obiekt tylko i wyłącznie klasy po niej dziedziczącej.
- Implementowany interfejs `RunnableFuture` jest połączeniem interfejsów `Runnable` i `Future` (z niego właśnie pochodzi metoda `get()` zwracająca rezultat obliczeń).
- **Parametr T** – określa typ zwracany przez metody `doInBackground()` i `get()`.
Jest to rezultat naszego zadania np. liczba / napis / obrazek.
- **Parametr V** – określa typ danych pośrednich, które może produkować zadanie, np. linie tekstu z wczytywanego pliku albo status postępu symulacji.
 - Dane te można uzyskać za pomocą metody `process(List dane)`,
 - A dane pochodzą z metody `publish(V... dane)` – która powinna być wywoływana w implementacji metody `doInBackground()`.
- → Aby metody nic nie zwracały, w deklaracji klasy należy podstawić parametr `Void` (przez duże „V”).



SwingWorker – wybrane metody

abstract protected doInBackground() - najważniejsza metoda.

W niej powinniśmy napisać kod zadania do wykonania.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonywanego zadania, czyli pobrać rezultat działania `doInBackground()` przy pomocy metody `get()`



SwingWorker – wybrane metody

abstract protected doInBackground() - najważniejsza metoda.

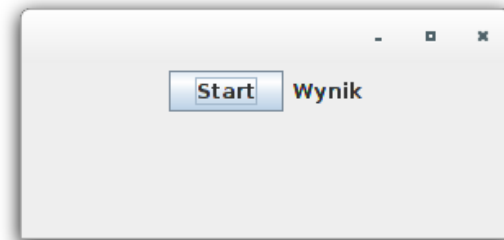
W niej powinniśmy napisać kod zadania do wykonania.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonywanego zadania, czyli pobrać rezultat działania `doInBackground()` przy pomocy metody `get()`

Przykład:

Chcemy napisać wątek, który

- 1) odpala się po naciśnięciu przycisku Start
- 2) następnie w tle (niezależnie od GUI)
 - Przez 5 sekund udaje, że coś bardzo intensywnie liczy (`Thread.sleep(5000);`)
 - Zwraca wynik obliczeń w postaci String (jest to napis “Wynik”)
- 3) Wynik działania wątku powinien zostać wyświetlony w głównym okienku po upływie określonego czasu



SwingWorkerDemo.java

```
buttonStart.addActionListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e) {
```

1) odpala się po naciśnięciu przycisku Start

```
});
```

SwingWorkerDemo.java

```
buttonStart.addActionListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e) {  
  
        SwingWorker<String, Void> worker = new  
            SwingWorker<String, Void>(){  
  
            protected String doInBackground() throws Exception {  
  
                Thread.sleep(5000);  
                return "Wynik";  
            }  
        }  
    }  
};
```

2) następnie w tle (niezależnie od GUI)

- Przez 5 sekund udaje, że coś bardzo intensywnie liczy (Thread.sleep(5000);)
- Zwraca wynik obliczeń w postaci String (jest to napis “Wynik”)

```
};  
worker.execute();  
} });
```



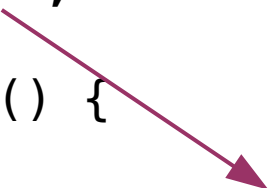
SwingWorkerDemo.java

```
buttonStart.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {

        SwingWorker<String, Void> worker = new
            SwingWorker<String, Void>(){

                protected String doInBackground() throws Exception {

                    Thread.sleep(5000);
                    return "Wynik";
                }
                protected void done() {
                    try {
                        etykieta.setText(get());
                    } catch (Exception ex) {ex.printStackTrace(); }
                }
            };
        worker.execute();
    }
});
```



3) Wynik działania wątku powinien zostać wyświetlony w okienku po upływie określonego czasu



SwingWorker – wybrane metody

abstract protected doInBackground() - najważniejsza metoda.

W niej powinniśmy napisać kod zadania do wykonania.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonywanego zadania, czyli pobrać rezultat działania doInBackground() przy pomocy metody get()



SwingWorker – wybrane metody

abstract protected doInBackground() - najważniejsza metoda.

W niej powinniśmy napisać kod zadania do wykonania.

- Jeśli przesłoniemy także metodę *process(List dane)*, możemy przy pomocy *publish(V... dane)* wywoływanej w *doInBackground()* przekazywać do *process(List dane)* częściowe wyniki działania zadania.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonywanego zadania, czyli pobrać rezultat działania *doInBackground()* przy pomocy metody *get()*

protected void process(List dane) - dzięki tej metodzie możemy operować na pośrednich danych zwróconych przez *publish(V... dane)*. W tej metodzie możemy bezpiecznie operować na komponentach graficznych ponieważ działa ona asynchronicznie w EDT.

protected void publish(V... chunks) - tworzona automatycznie, przesyła częściowe dane metodzie *protected process(List dane)*, nie ma potrzeby jej przesłaniania, ale można to zrobić w przypadku gdy np. chcemy wykonać jakieś dodatkową operację na dostarczanych danych.



SwingWorker – wybrane metody

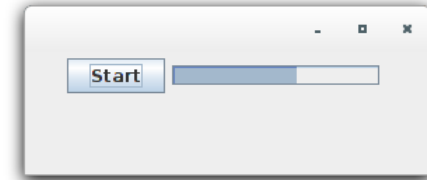
abstract protected doInBackground() - najważniejsza metoda.

W niej powinniśmy napisać kod zadania do wykonania.

- Jeśli przesłoniemy także metodę *process(List dane)*, możemy przy pomocy *publish(V... dane)* wywoływanej w *doInBackground()* przekazywać do *process(List dane)* częściowe wyniki działania zadania.

protected void done() - wywoływana po zakończeniu zadania, wykonywana w EDT, można w niej przeprowadzić „sprzątanie” i zaprezentować w GUI główny rezultat wykonanego zadania, czyli pobrać rezultat działania.

Przykład:



Chcemy napisać wątek, który

1) odpala się po naciśnięciu przycisku Start

2) następnie w tle (niezależnie od GUI)

- Przez 5 sekund udaje, że coś bardzo intensywnie liczy, **ale co 1 sekundę wyświetla status na pasku postępu.**

- Zwraca wynik obliczeń w postaci String (jest to napis “Wynik”)

3) Wynik działania wątku powinien zostać wyświetlony w okienku po upływie określonego czasu

SwingWorkerProgressDemo.java

```
JProgressBar progressBar = new JProgressBar(0,5);  
progressBar.setValue([int]);
```

```
buttonStart.addActionListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e) {
```

```
        SwingWorker<String, Integer> worker = new  
            SwingWorker<String, Integer>(){
```

```
                protected String doInBackground() throws Exception {  
                    //co sekunde ma wyswietlac postep
```

```
                        return "Wynik";  
                }
```

```
                protected void done() {  
                    try {
```

```
                        etykieta.setText(get());  
                    } catch (Exception ex) {ex.printStackTrace(); }  
                }  
            };  
        worker.execute();  
    } });
```

Co 1 sekundę wyświetla
status na pasku postępu → postęp
będzie wyrażany przez "Integer"



SwingWorkerProgressDemo.java

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {

        SwingWorker<String, Integer> worker = new
            SwingWorker<String, Integer>(){

                protected String doInBackground() throws Exception {
                    for(int i=1;i<=5;i++){
                        Thread.sleep(1000);
                        publish(i);
                    }
                    return "Wynik";
                }

                protected void done() {
                    try {
                        etykieta.setText(get());
                    } catch (Exception ex) {ex.printStackTrace(); }
                }
            };
        worker.execute();
    } });
```

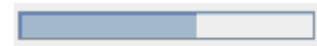
Co 1 sekundę wyświetla
status na pasku postępu → postęp
będzie wyrażany przez "Integer"



SwingWorkerProgressDemo.java

```
SwingWorker<String, Integer> worker = new  
    SwingWorker<String, Integer>(){
```

```
protected void process(List<Integer> dane) {  
    for(Integer progress : dane)  
        progressBar.setValue(progress);  
}
```



```
protected String doInBackground() throws Exception {  
    for(int i=1;i<=5;i++){  
        Thread.sleep(1000);  
        publish(i);  
    }  
    return "Wynik";  
}
```

Co 1 sekundę wyświetla
status na pasku postępu → postęp
będzie wyrażany przez "Integer"

```
protected void done() {  
    try {  
        etykieta.setText(get());  
    } catch (Exception ex) {ex.printStackTrace(); }  
}  
};
```



SwingWorkerProgressDemo.java

```
SwingWorker<String, Integer> worker = new
    SwingWorker<String, Integer>(){

    protected void process(List<Integer> dane) {
        for(Integer progress : dane)
            progressBar.setValue(progress);
    }

    protected String doInBackground() throws Exception {
        for(int i=1;i<=5;i++){
            Thread.sleep(1000);
            publish(i);
        }
        return "Wynik";
    }

    protected void done() {
        try {
            etykieta.setText(get());
        } catch (Exception ex) {ex.printStackTrace(); }
    }
};
```

Kompilator pozwoliłby zmienić `progressBar`
w `doInBackground` ale to by było

ZŁE.

ProgressBar jest częścią GUI.
Nie zmieniamy GUI w wątkach tła!!!



SwingWorkerCopyFiles.java

- **OnCopyActionListener**
 - interfejs wyboru plików i uruchamiania SwingWorkera:
`copySwingWorker.execute();`
- **protected Void doInBackground()**
 - Worker, pracujący w tle, w którym zachodzi kopiowanie...
- **Wykorzystanie metod `setProgress()`, `getProgress()` i interfejsu „zmiany stanu”:**
 - `setProgress(progress);`
 - `addChangeListener (...`
`progressBar.setValue(getProgress()));`



Więcej o wątkach w Swing

- https://www.javamex.com/tutorials/threads/swing_ui.shtml
- <https://www.math.uni-hamburg.de/doc/java/tutorial/uiswing/misc/threads.html>
- http://www.comp.nus.edu.sg/~cs3283/ftp/Java/swingConnect/archive/tech_topics_arch/threads/threads.html



Podsumowanie - wątki

- Każdy program ma przynajmniej jeden wątek
- Wątki są po to, by symulować równoległe wykonywanie czynności
- Nikt nie może przewidzieć jaka będzie kolejność przydzielania procesora poszczególnym wątkom



Podsumowanie – wątki w Swing

- **Tworzenie i wszelkie zmiany GUI należy wykonywać w wątku EDT.**

```
SwingUtilities.invokeLater(  
    new Runnable(){  
        public void run() {  
            CosZGUI();  
        }  
    });
```

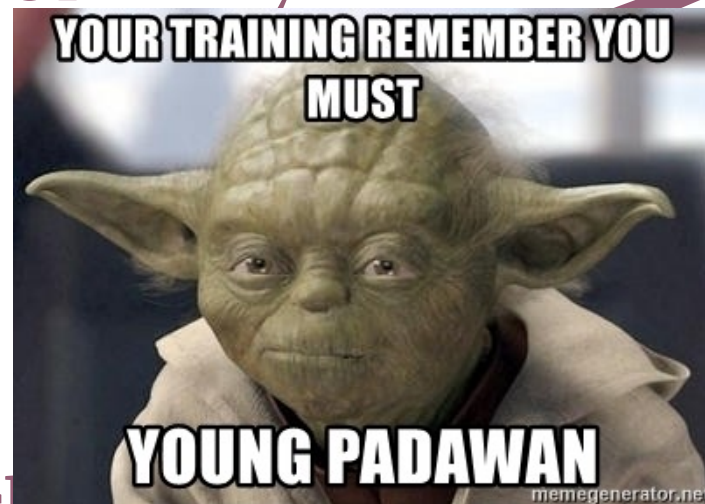
- Długie obliczenia, czytanie długich plików czy zapytania do bazy danych należy wykonywać **poza EDT**.
 - Inaczej może nas trafić efekt “zawieszenia” programu.
 - Pomaga nam w tym klasa `SwingWorker`.



Podsumowanie – wątki w Swing

- Tworzenie i wszelkie zmiany GUI należy wykonywać w wątku EDT.

```
SwingUtilities.invokeLater(  
    new Runnable(){  
        public void run() {  
            CosZGUI();  
        }  
    });
```



- Długie obliczenia, czytanie długich plików czy zapytania do bazy danych należy wykonywać **poza EDT**.
 - Inaczej może nas trafić efekt “zawieszenia” programu.
 - Pomaga nam w tym klasa `SwingWorker`.



Projekty

WYMOGI FORMALNE

- Jeśli projekt zawiera poważne błędy związane z wielowątkowością kodu ocena może zostać znacząco obniżona, może nawet doprowadzić do niezaliczenia projektu
- W szczególności, jeśli program korzysta z wątków niepoprawnie i tą niepoprawność widać bez zaglądania do kodu programu.
- Na przykład:
 - Długotrwałej symulacji nie da się przerwać bez zamykania programu.
 - Podczas wykonywania obliczeń program się "zacina".



Proste Animacje



Animacje

Ogólny schemat tworzenia animacji:

- Wykorzystać jedną z omawianych wcześniej metod tworzenia wątków do periodycznego przerysowywania wybranego komponentu (np. dziedziczącego z JPanel...) z każdorazową zmianą rysowanej „sceny”
- Przerysowywanie komponentu zaleca się umieszczać w EDT
 - Java domyślnie tak robi, jeśli wywołacie metodę `repaint()`;



Przykładowe proste animacje

- **ProstaAnimacja.java** – animowany JPanel
- **PlikiGraficzne0.java, PlikiGraficzne.java** – animacja z kilku plików jpg (periodyczne setIcon() w JLabel)
- **PlikiGraficzneTimer.java** – j.w. z wykorzystaniem klasy Timer (swing) i możliwością sterowania wątkiem
- **Prostokat.java, Rysowanie7.java** – animacja obiektu Prostokat, ExecutorService



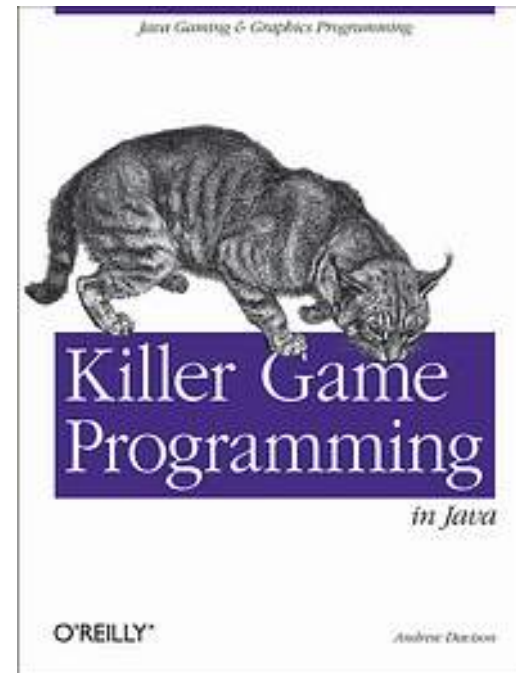
Animacje – eliminowanie migotania

- Jeśli animacje „migają” warto zastosować tzw. „podwójne buforowanie”, które polega na utworzeniu obiektu pośredniego (np. klasy `BufferedImage`) na którym następuje rysowanie sceny, a odświeżanie polega tylko na podmienieniu obrazka wyświetlanego w metodzie `paintComponent` (np. `g.drawImage(pilkaImg, 0, 0, this);`) – przykład w **Rysowanie8.java** i **RysowanieUproszczone.java**
- Tworzenie złożonej sceny może trwać dłużej (np. wczytanie pliku tła, obliczanie współrzędnych animowanych obiektów) – zdecydowanie powinno być poza EDT, natomiast odświeżanie obrazka w EDT



Polecana dalsza lektura dla zaawansowanych

- <http://fivedots.coe.psu.ac.th/~ad/jg/index.html>
- internetowa wersja obszernej książki
„Killer Game Programming in Java”
 - Animations
 - Java 3D



Ankieta

