

Programowanie równoległe i rozproszone

Wykład 2

**Programowanie współbieżne w Javie:
wątki, sekcje krytyczne,
interakcja pomiędzy wątkami**

Dla przypomnienia -> Proces a Wątek

Proces

- wykonywalny program,
- własna przestrzeń adresowa,
- droga komunikacja,

Wątki

- wątek to część programu,
- wspólna przestrzeń adresowa,
- tania komunikacja,

Wątki w Javie

Podczas uruchamiania programu w Javie powoływany jest automatycznie do życia tzw. wątek główny. Od niego pochodzą wszystkie wątki potomne i on też zazwyczaj kończy działanie całego programu.

Każdy wątek ma początek, sekwencje instrukcji i koniec.

Wątek nie jest niezależnym programem, jest wykonywany jako część programu.

Przemysław Stpiczyński, Marcin Brzuszek
„Programowanie współbieżne i rozproszone w języku Java”

https://www.researchgate.net/publication/309274480_Programowanie_wspolbiezne_i_rozproszone_w_jezyku_Java

W Javie wątki są obiektami zdefiniowanymi za pomocą specjalnego rodzaju klas.

Program wielowątkowy definiujemy, na dwa sposoby:

- jako podklasę klasy **Thread**,
- implementując interfejs **Runnable**,

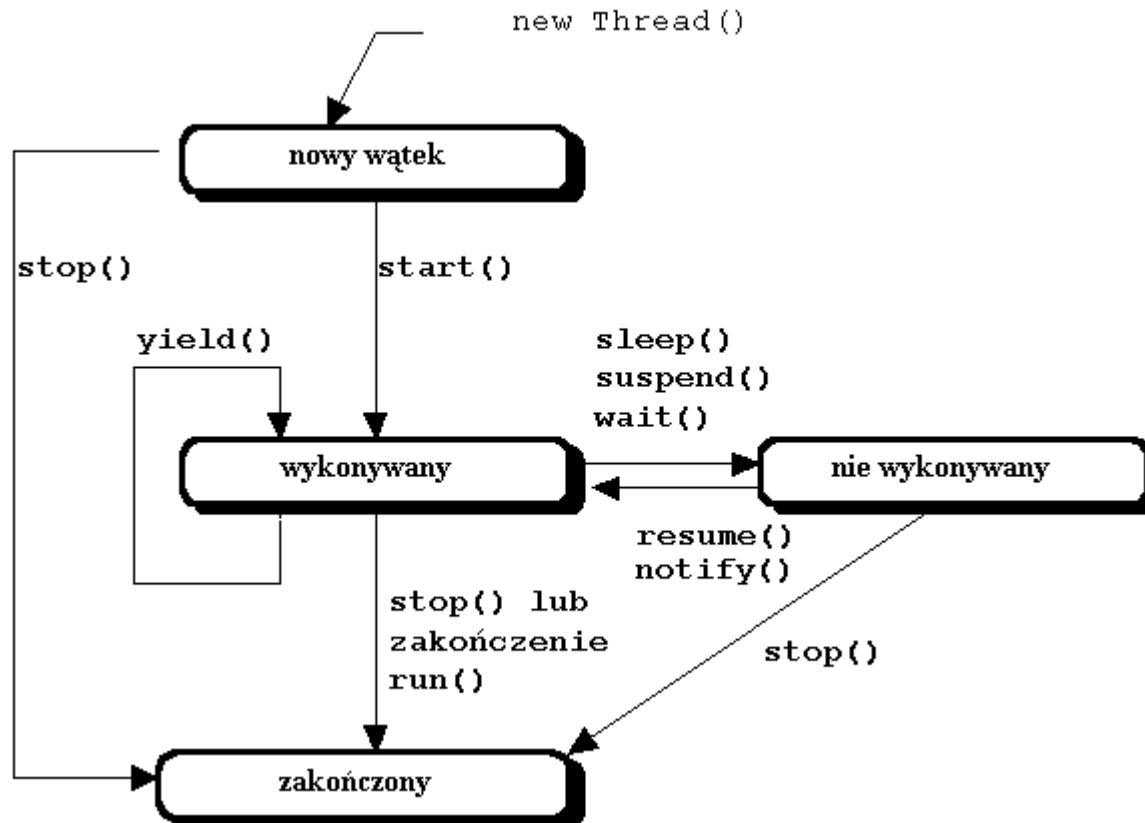
Tworzenie wątków

Polega na zdefiniowaniu klasy rozszerzającej klasę **Thread**, a następnie utworzeniu obiektu tej klasy. Nowa klasa musi nieodzownie zawierać metodę przesłaniającą metodę **run()**, stanowiąca początek kodu wątku.

Stany wątku

W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
- wykonywany,
- nie wykonywany,
- zakończony,



W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
- wykonywany,
- nie wykonywany,
- zakończony,

Do tworzenia nowego wątku służy instrukcja

```
Thread mojWatek = new MojaKlasaWatku();
```

- po wykonaniu tej instrukcji mamy pusty obiekt Thread,
- w momencie utworzenia wątku nie posiada on jeszcze żadnych zasobów komputera,
- po utworzeniu wątku, możemy go uruchomić metodą **start**, lub zatrzymać metodą **stop**,

W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
- **wykonywany**,
- nie wykonywany,
- zakończony,

Do uruchomienia wątku służy instrukcja

```
Thread mojWatek = new MojaKlasaWatku();  
mojWatek.start();
```

- metoda **start()** przedziela zasoby komputera niezbędne do wykonania wątku, uruchamia wątek oraz wywołuje metodę **run()**,

W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
 - wykonywany,
 - **nie wykonywany**,
 - zakończony,
- wątek przechodzi do stanu "**nie wykonywany**" gdy zachodzi jedno z poniższych zdarzeń:
- wywołano metodę **sleep ()**,
 - wywołano metodę **suspend ()**,
 - wątek wywołuje swoją metodę **wait ()**,
 - wątek jest zablokowany przy operacji wejścia / wyjścia (ang. I/O).

Przykład uśpienia wątku na 1000 milisekund

```
try
{ Thread.sleep(1000); }
catch (InterruptedException e)
{ }
```


W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
- wykonywany,
- nie wykonywany,
- zakończony,

Warunki, jakie muszą być spełnione, aby nastąpił powrót do stanu

"wykonywany":

- jeśli wątek uśpiono (metoda `sleep()`), musi upłynąć określona liczba milisekund,
- jeśli wątek zawieszono (metoda `suspend()`), inny wątek musi wywołać metodę `resume()` wątku, powodującą jego odwieszenie,
- jeśli wątek czeka na np. ustawienie jakiejś zmiennej, to obiekt, do którego należy ta zmienna, musi ją odstąpić a następnie wywołać metodę `notify()` lub `notifyAll()`,
- jeśli wątek jest zablokowany przy operacjach wejścia / wyjścia, wtedy operacje te muszą być zakończone.

W czasie swego istnienia wątek może znajdować się w jednym z kilku stanów:

- nowy,
- wykonywany,
- nie wykonywany,
- zakończony,

- wątek może zakończyć działanie z dwu powodów: albo naturalnie zakończy swe działanie albo zostanie zabity
- wątek naturalnie kończy swoje działanie wtedy, gdy jego metoda run kończy się normalnie
- możemy także zabić wątek w każdym momencie poprzez wywołanie jego metody stop

```
Thread mojWatek = new MojaKlasaWatku();  
mojWatek.start();  
try  
{  
    Thread.sleep(10000);  
} catch (InterruptedException e){}  
mojWatek.stop();
```

- metoda stop oznacza nagłe zakończenie wykonania metody run wątku, w takim przypadku np. wykonywane przez jego obliczenia mogą być stracone

Przykład programu wielowątkowego realizowanego jako podklasa klasy Thread

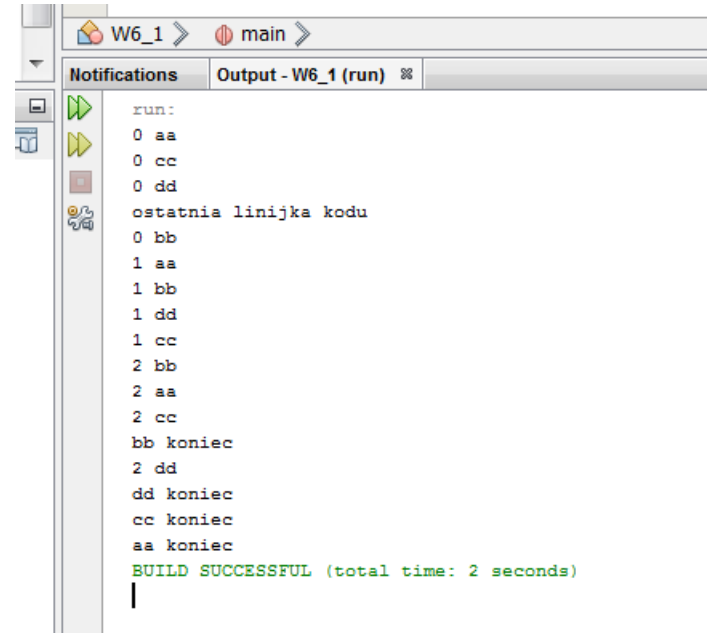
```
class Watek extends Thread
{
    String wyraz;

    public Watek(String str, int numer)
    {
        wyraz = str;
    }

    public void run()
    {
        for (int k = 0; k < 3; k++)
        {
            System.out.println(k + " " + wyraz);
            try
            {
                sleep( (int)(Math.random() * 1000) );
            }
            catch ( InterruptedException e )
            {
                e.printStackTrace();
            }
            System.out.println(wyraz + " koniec" );
        }
    }
}
```

- klasa **Watek** implementuje działanie pojedynczego wątku,
- klasa dziedziczy z klasy bazowej **Thread**,
- wszystkie zadania, jakie ma wykonywać wątek umieszczone są w metodzie **run** wątku,
- po utworzeniu i inicjalizacji wątku, środowisko przetwarzania wywołuje metodę **run**,
- wątek ma za zadanie, pobrać jako argument w konstruktorze nazwę i wypisać ją 3 razy, w międzyczasie jest usypiany na pewien czas za pomocą **sleep((int)(Math.random() * 1000));**

```
class W6_1
{
    public static void main (String[] args) throws Exception
    {
        new Watek("aa",1).start();
        new Watek("bb",2).start();
        new Watek("cc",3).start();
        new Watek("dd",4).start();
        System.out.println("ostatnia linijka kodu");
    }
}
```



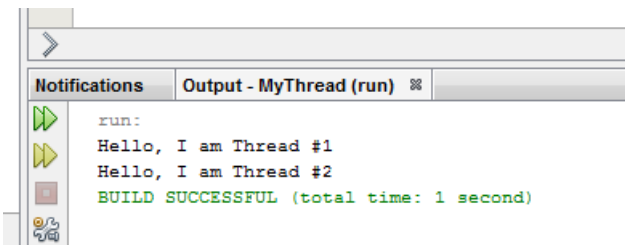
```
run:
0 aa
0 cc
0 dd
ostatnia linijka kodu
0 bb
1 aa
1 bb
1 dd
1 cc
2 bb
2 aa
2 cc
bb koniec
2 dd
dd koniec
cc koniec
aa koniec
BUILD SUCCESSFUL (total time: 2 seconds)
```

- klasa **W6_1** definiuje całą aplikację,
- w metodzie **main()** tworzone są cztery wątki o nazwach: aa, bb, cc, dd,
- wszystkie wątki zaraz po ich utworzeniu są uruchamiane dzięki użyciu metody **start()**,

Kolejny prosty przykład tworzący 2 wątki

```
class MyThread extends Thread {  
    public MyThread (String s) {  
        super(s);  
    }  
  
    public void run() {  
        System.out.println("Hello, I am " + getName());  
    }  
}
```

```
public class TestThread {  
    public static void main (String arg[]) {  
        MyThread t1, t2;  
        t1 = new MyThread ("Thread #1");  
        t2 = new MyThread ("Thread #2");  
  
        t2.start();  
        t1.start();  
    }  
}
```

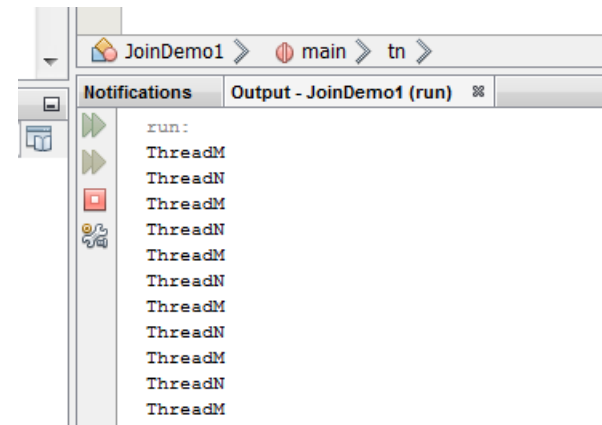


Kolejny przykład także tworzący 2 wątki: **TreadN** i **ThreadM**

```
class ThreadN extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                Thread.sleep(1000);
                System.out.println("ThreadN");
            }
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

class ThreadM extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                Thread.sleep(1000);
                System.out.println("ThreadM");
            }
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}

class JoinDemo1 {
    public static void main(String args[]) {
        ThreadM tm = new ThreadM();
        tm.start();
        ThreadN tn = new ThreadN();
        tn.start();
        try {
            tm.join();
            tn.join();
            System.out.println("Both threads havfinished");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Mechanizm synchronizacji

- kolejnej wersji programu wprowadzimy synchronizację sekcji krytycznej, tak aby mogła być ona wykonana bez przerywania od początku do końca przez wykonujący ją wątek – czyli dostęp do sekcji krytycznej jest blokowany w danym momencie dla innych wątków,
- synchronizacja jest realizowana przy pomocy słowa kluczowego

`'synchronized' '(' objectidentifier ')' '{' // Critical code section '}' ,`

- z punktu widzenia kodu źródłowego, wątek próbuje wejść do kodu sekcji krytycznej -> w tym momencie system operacyjny (a dokładnie JVM – maszyna wirtualna Javy) sprawdza, czy inny wątek aktualnie nie korzysta już z kodu sekcji krytycznej,
- jeżeli kod sekcji krytycznej jest wykonywany przez inny wątek, to oczywiście żaden inny wątek nie będzie miał do niego dostępu w tym momencie,
- na następnej stronie przedstawiono zmodyfikowany kod programu,

Synchronizacja wątków

Przykład – klasyczny problem Producent-Konsument

Założenia:

- wykonywane są dwa niezależne wątki: Producenta oraz Konsumenta,
- wątki te współdzielą dane i stan każdego z nich zależy od stanu drugiego wątku,
- Producent generuje ciąg danych, które są wykorzystywane (konsumowane) przez Konsumenta,
- ciąg tych danych stanowi wspólny zasób, wątki muszą być zatem synchronizowane.

Synchronizacja wątków

Przykład – klasyczny problem Producent-Konsument

Założenia – ciąg dalszy:

- wątek Producenta generuje liczby od 0 do 9, które są następnie składowane w obiekcie typu CubbyHole (Pudełko),
- Producent, po włożeniu do pudełka liczby i wypisaniu jej na ekranie, zostaje uśpiony na losowo wybrany czas, następnie generuje kolejną liczbę,
- Konsument podczas swego działania konsumuje wszystkie liczby złożone w pudełku, wyprodukowane przez Producenta, tak szybko, jak staną się one dostępne,
- Producent i Konsument w tym przykładzie współdzielą dane przez wspólny obiekt typu CubbyHole,
- Konsument ma prawo pobrać każdą wyprodukowaną liczbę tylko raz,
- synchronizacja między tymi dwoma wątkami występuje w metodach get() i put() obiektu CubbyHole,

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);

            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

- wątek Producenta generuje liczby od 0 do 9, które są następnie składowane w obiekcie typu CubbyHole (Pudełko),
Producent, po włożeniu do pudełka liczby i wypisaniu jej na ekranie, zostaje uśpiony na losowo wybrany czas, następnie generuje kolejną liczbę,

```
public class Consumer extends Thread {  
    private CubbyHole cubbyhole;  
    private int number;  
  
    public Consumer(CubbyHole c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
  
    public void run() {  
        int value = 0;  
        for (int i = 0; i < 10; i++) {  
            value = cubbyhole.get();  
            System.out.println("Consumer #" + this.number  
                               + " got: " + value);  
        }  
    }  
}
```

- Konsument podczas swego działania konsumuje wszystkie liczby złożone w pudełku, wyprodukowane przez Producenta, tak szybko, jak staną się one dostępne,

```
public class CubbyHole {  
    private int contents;  
    private boolean available = false;  
  
    public synchronized int get() {  
        while (available == false) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        available = false;  
        notifyAll();  
        return contents;  
    }  
  
    public synchronized void put(int value) {  
        while (available == true) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        contents = value;  
        available = true;  
        notifyAll();  
    }  
}
```

- Producent i konsument w tym przykładzie współdzielą dane przez wspólny obiekt typu CubbyHole,
- Konsument ma prawo pobrać każdą wyprodukowaną liczbę tylko raz,
- synchronizacja między tymi dwoma wątkami występuje w metodach **get()** i **put ()** obiektu **CubbyHole**

```
public class W6_2 {  
    public static void main(String[] args) {  
        CubbyHole c = new CubbyHole();  
        Producer p1 = new Producer(c, 1);  
        Consumer c1 = new Consumer(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```

Klasa główna aplikacji W6_2:

- tworzy obiekty typu CubbyHole, Producent i Konsument,
- następnie uruchamia wątki Producenta i Konsumenta (współdzielące obiekt typu Pudełko)

```
run:  
Producer #1 put: 0  
Consumer #1 got: 0  
Producer #1 put: 1  
Consumer #1 got: 1  
Producer #1 put: 2  
Consumer #1 got: 2  
Producer #1 put: 3  
Consumer #1 got: 3  
Consumer #1 got: 4  
Producer #1 put: 4  
Consumer #1 got: 5  
Producer #1 put: 5  
Producer #1 put: 6  
Consumer #1 got: 6  
Producer #1 put: 7  
Consumer #1 got: 7  
Producer #1 put: 8  
Consumer #1 got: 8  
Consumer #1 got: 9  
Producer #1 put: 9  
BUILD SUCCESSFUL (total time: 2 seconds)
```

- w programie występują 2 mechanizmy synchronizacji wątków Producenta i Konsumenta:
 - **monitor**,
 - dwie metody: **wait** oraz **notifyAll**,
- klasa **CubbyHole**, jest współdzielona pomiędzy dwa wątki, dostęp do niej jest synchronizowany za pomocą zmiennych warunkowych,
- Java pozwala synchronizować wątki pracujące ze zmiennymi warunkowymi dzięki użyciu monitorów,
- gdy wątek zajmie monitor dla jakiejś danej, inne wątki do czasu zwolnienia monitora zostają zablokowane i nie mogą odczytywać lub modyfikować danych,
- segment kodu w programie, w którym następuje dostęp do tej samej danej z różnych wątków nazywany jest sekcją krytyczną (ang. **critical section**),
- w Javie sekcję krytyczną oznaczmy przy użyciu słowa kluczowego **synchronized**.

W Javie każdy obiekt, który ma metody synchroniczne posiada swój monitor

- W klasie **CubbyHole** mamy 2 metody synchroniczne:
 - metodę **put()**, używaną do zmiany wartości w obiekcie typu Pudelko,
 - metodę **get()**, która jest używana do pobrania liczby przechowywanej w obiekcie Pudelko.
- oznacza to, że system skojarzy z każdym obiektem typu **CubbyHole** unikalny monitor.

W przedstawionym poniżej kodzie klasy **CubbyHole** elementy wyróżnione pogrubioną czcionką służą do synchronizacji wątków:

```
public class CubbyHole {  
    private int contents;  
    private boolean available = false;  
  
    public synchronized int get() {  
        while (available == false) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        available = false;  
        notifyAll();  
        return contents;  
    }  
  
    public synchronized void put(int value) {  
        while (available == true) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        contents = value;  
        available = true;  
        notifyAll();  
    }  
}
```

- klasa posiada dwa pola danych:
- **contents** - określa bieżącą zawartość pudełka – wyprodukowaną liczbę,
- **available** - które określa, czy zawartość pudełka może być pobrana,

Po usunięciu z klasy CubbyHole składników synchronizacyjnych pojawia się następujący kod

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

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

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

```
public class CubbyHole {
    private int contents;

    public int get() {
        return contents;
    }

    public void put(int value) {
        contents = value;
    }
}
```

Wynik programu bez synchronizacji

```
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
BUILD SUCCESSFUL (total time: 2 seconds)
```

Metody **notifyAll** i **wait**

- metody **wait** i **notifyAll** mogą być wywoływane tylko przez wątki, które założyły blokadę,
- metoda **notifyAll** powiadamia wszystkie wątki oczekujące na monitor zajęty przez bieżący wątek o zwolnieniu tego monitora i budzi te wątki,
- w prezentowanym przykładzie, metody **wait** i **notifyAll** służą do koordynacji wkładania i wyjmowania liczb z CubbyHole,

Główne metody klasy Thread w Javie

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.

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 zawieszonego wątku,

suspend - zablokowanie wątku,

resume - odblokowanie wątku zawieszonego przez suspend,

setDaemon - ustanowienie wątku demonem,

isDaemon - testowanie czy wątek jest demonem.

Priorytet wątku

- priorytet wątku informuje program szeregujący wątki Javy (ang. Java thread scheduler), kiedy nasz wątek powinien być wykonywany w odniesieniu do innych wątków,
- gdy nowy wątek jest tworzony, dziedziczy priorytet z wątku, który go utworzył,
- priorytety wątku mogą być modyfikowane w każdej chwili po utworzeniu wątku poprzez użycie metod **setPriority**,
- priorytet wątku jest liczbą typu integer o wartości większej lub równej **MIN_PRIORITY** i nie większej niż **MAX_PRIORITY** - są to stałe zdefiniowane w klasie Thread o wartości odpowiednio 1 i 10,
- im większa wartość liczby określającej priorytet, tym priorytet wątku jest większy,

Grupowanie wątków

- każdy wątek Javy jest członkiem grupy wątków (ang. thread group),
- grupowanie wątków w jednym obiekcie pozwala na jednoczesne manipulowanie wszystkimi zgrupowanymi wątkami,
- grupowanie wątków w Javie zaimplementowano w klasie *java.lang.ThreadGroup*,

przykład zadeklarowania i stworzenia nowej grupy wątków, do której przypisany zostaje nowo stworzony wątek:

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");  
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Aby dowiedzieć się, do jakiej grupy wątek należy, wystarczy wywołać metodę *getThreadGroup*:

```
theGroup = myThread.getThreadGroup();
```

```
class CalcPI1
{
    public static void main (String [] args)
    {
        MyThread mt = new MyThread ();
        mt.start ();
        try
        {
            Thread.sleep (10); // Sleep for 10 milliseconds
        }
        catch (InterruptedException e)
        {
        }
        System.out.println ("pi = " + mt.pi);
    }
}
class MyThread extends Thread
{
    boolean negative = true;
    double pi; // Initializes to 0.0, by default
    public void run ()
    {
        for (int i = 3; i < 100000; i += 2)
        {
            if (negative)
                pi -= (1.0 / i);
            else
                pi += (1.0 / i);
            negative = !negative;
        }
        pi += 1.0;
        pi *= 4.0;
        System.out.println ("Finished calculating PI");
    }
}
```

- aplikacja generuje wartość liczby PI na podstawie algorytmu matematycznego,
- liczba PI jest generowana w odrębnym wątku,
- metoda **sleep()** zatrzymuje działanie wątku na n milisekund,

- wynik aplikacji to:

pi = -0.2146197014017295
Finished calculating PI

- błędny wynik wynika z faktu, wcześniejszego wypisania wyniku za pomocą **System.out.println ("pi = " + mt.pi);** przed zakończeniem działania wątku

```

class CalcPI2
{
    public static void main (String [] args)
    {
        MyThread mt = new MyThread ();
        mt.start ();
        while (mt.isAlive ())
            try
            {
                Thread.sleep (10); // Sleep for 10 milliseconds
            }
            catch (InterruptedException e)
            {
            }
        System.out.println ("pi = " + mt.pi);
    }
}

class MyThread extends Thread
{
    boolean negative = true;
    double pi; // Initializes to 0.0, by default
    public void run ()
    {
        for (int i = 3; i < 100000; i += 2)
        {
            if (negative)
                pi -= (1.0 / i);
            else
                pi += (1.0 / i);
            negative = !negative;
        }
        pi += 1.0;
        pi *= 4.0;
        System.out.println ("Finished calculating PI");
    }
}

```

- poprawiona wersja poprzedniego programu, poprzez dodanie do kodu metody **isAlive ()**
- w takim przypadku głównego wątek programu w którym istnieje instrukcja wypisująca wynik PI **System.out.println ("pi = " + mt.pi)** czeka na zakończenie wątku MyThread, w którym liczymy wartość liczby PI


```
class CalcPI3
{
    public static void main (String [] args)
    {
        MyThread mt = new MyThread ();
        mt.start ();
        try
        {
            mt.join ();
        }
        catch (InterruptedException e)
        {
        }
        System.out.println ("pi = " + mt.pi);
    }
}

class MyThread extends Thread
{
    boolean negative = true;
    double pi; // Initializes to 0.0, by default
    public void run ()
    {
        for (int i = 3; i < 100000; i += 2)
        {
            if (negative)
                pi -= (1.0 / i);
            else
                pi += (1.0 / i);
            negative = !negative;
        }
        pi += 1.0;
        pi *= 4.0;
        System.out.println ("Finished calculating PI");
    }
}
```

- w poprzednim kodzie w pętli while w połączeniu z metodą **isAlive()** sprawdzaliśmy wielokrotnie czy wątek generujący liczbą PI już jest nadal „żywy”,
- aktualna wersja kodu zastępuje tę konstrukcję metodą **join()**,
- metoda **join()** pozwala zawiesić wątek w oczekiwaniu na zakończenie innego,
- nie mamy w tym przypadku zatem wielokrotnego sprawdzania, czy wątek klasy **MyThread** już się zakończył

Demony

- każdy wątek Javy może zostać demonem (ang. daemon thread),
- wątek będący demonem zajmuje się obsługiwaniem innych wątków uruchomionych w tym samym procesie, co wątek demona,
- metoda run wątku demona jest przeważnie nieskończoną pętlą, w której demon czeka na zgłoszenia zapotrzebowania na usługi dostarczane przez ten wątek,
- aby wątek został demonem używany metody **setDaemon** z argumentem równym true.
- w celu sprawdzenia, czy wątek jest demonem używana jest metoda **isDaemon**,

```
class UserDaemonThreadDemo
{
    public static void main (String [] args)
    {
        if (args.length == 0)
            new MyThread ().start ();
        else
        {
            MyThread mt = new MyThread ();
            mt.setDaemon (true);
            mt.start ();
        }
        try
        {
            Thread.sleep (100);
        }
        catch (InterruptedException e)
        {
        }
    }
}
class MyThread extends Thread
{
    public void run ()
    {
        System.out.println ("Daemon is " + isDaemon ());
        while (true);
    }
}
```

- przykład kodu, w którym uruchomiliśmy wątek jako Demon,
- każdy wątek Javy może zostać demonem (ang. daemon thread),
- wątek będący demonem zajmuje się obsługiwaniem innych wątków uruchomionych w tym samym procesie, co wątek demona,
- metoda run wątku demona jest przeważnie nieskończoną pętlą, w której demon czeka na zgłoszenia zapotrzebowania na usługi dostarczane przez ten wątek,

```
class NeedForSynchronizationDemo
{
    public static void main (String [] args)
    {
        FinTrans ft = new FinTrans ();
        TransThread tt1 = new TransThread (ft, "Deposit Thread");
        TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
        tt1.start ();
        tt2.start ();
    }
}

class FinTrans
{
    public static String transName;
    public static double amount;
}

class TransThread extends Thread
{
    private FinTrans ft;
    TransThread (FinTrans ft, String name)
    {
        super (name); // Save thread's name
        this.ft = ft; // Save reference to financial transaction object
    }
}
```

- program używający parę wątków do symulacji wpłat/wypłat z konta bankowego

Withdrawal 250.0
Withdrawal 2000.0
Deposit 2000.0
Deposit 2000.0
Deposit 250.0

```
public void run ()
{
    for (int i = 0; i < 100; i++)
    {
        if (getName ().equals ("Deposit Thread"))
        {
            // Start of deposit thread's critical code section
            ft.transName = "Deposit";
            try
            {
                Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 2000.0;
            System.out.println (ft.transName + " " + ft.amount);
            // End of deposit thread's critical code section
        }
        else
        {
            // Start of withdrawal thread's critical code section
            ft.transName = "Withdrawal";
            try
            {
                Thread.sleep ((int) (Math.random () * 1000));
            }
            catch (InterruptedException e)
            {
            }
            ft.amount = 250.0;
            System.out.println (ft.transName + " " + ft.amount);
            // End of withdrawal thread's critical code section
        }
    }
}
```

Opis programu:

- operacje wpłaty i wypłaty środków pieniężnych realizowane są przez wątek **TransThread** w zależności od nazwy wywołania wątku

```
TransThread tt1 = new TransThread (ft, "Deposit Thread"); //wątek wpłat  
TransThread tt2 = new TransThread (ft, "Withdrawal Thread"); //wątek wypłat
```

- wątek posiada 2 sekcje krytyczne, jedną dla operacji wpłaty, drugą dla operacji wypłaty środków pieniężnych,
- w trakcie jego działania spodziewamy się dwóch wartości:
2000.0 dla depozytu oraz 250.0 po wycofaniu się środków
- niestety brak synchronizacji powoduje, że wyniki są odmienne, co przedstawiono na poprzednim slajdzie,
- wynika to z faktu, że czas dostępu wątku do procesora może być krótszy niż wymagany do pełnego wykonania sekcji krytycznej i w międzyczasie konkurujący o zasób procesora wątek może zacząć wykonywać swój kod zmieniając wartość zmiennej amount,

Budowa aplikacji serwera - protokół TCP (klasa `ServerSocket`)

Zadaniem programu serwera jest realizacja usługi na rzecz klienta. Program serwera rozpoczyna pracę w systemie komputerowym po czym zasypia i oczekuje na zgłoszenie klienta zamawiającego jego usługę.

W języku Java funkcję serwera połączeń z potwierdzeniem (TCP) realizują przede wszystkim obiekty klasy `ServerSocket`.

Obiekt klasy `ServerSocket` nasłuchuje na przydzielonym porcie do momentu nadejścia zgłoszenia po czym budzi się i rozpoczyna realizację usługi.

Aplikacja Javy (aplikacja wielowątkowa) może utworzyć kilka obiektów klasy `ServerSocket` a tym samym postawić kilka serwerów czuwających na różnych portach (na jednym hoście, na każdym porcie może działać jeden i tylko jeden serwer!).

Serwery możemy podzielić na dwie podstawowe grupy

1. serwery iteracyjne - obsługujące w danej chwili tylko jednego klienta. Kolejne zgłoszenia klientów oczekują w kolejce na połączenie z serwerem. Takie rozwiązania są najprostsze w realizacji lecz dla wielu zastosowań niewystarczające. Serwery iteracyjne buduje się dla prostych usług, w których czas obsługi usługi jest krótki i znany z góry. Jest to rozwiązanie stosowane również w specyficznych rozwiązaniach, w których serwer komunikuje się jedynie z jednym stałym klientem.

2. serwery współbieżne - realizowane w sytuacji gdy czas obsługi usługi może być różny w zależności od żądań klienta a także gdy serwer ma obsłużyć wyjątkowo dużo zgłoszeń od klientów. Serwer współbieżny (w Javie obiekt `ServerSocket`), po ustanowieniu połączenia, tworzy nowy proces (wątek) do wykonania zamówień klienta po czym ponownie zasypia w oczekiwaniu na kolejne zgłoszenie. Przykładami takich serwerów mogą być serwery usług ftp, http itd.

Konstruktor klasy `ServerSocket`

- `public ServerSocket(int port)`

konstruktor gniazda serwera z parametrem *port* - numer portu wiązanego z serwerem typu *int*;

Przykładowy fragment poprawnie utworzonego gniazda serwera z obsługą wyjątków może wyglądać następująco:

```
try {  
    ServerSocket server = new ServerSocket (80);  
    ...  
}  
catch (IOException e) {  
    System.out.println("Bład utworzenia gniazda");  
}
```


Przykład aplikacji serwera - serwer iteracyjny

Aplikacja realizuje funkcję serwera iteracyjnego - przyjmuje zgłoszenie klienta, realizuje usługę. Kolejny klient żądający połączenia oczekuje w kolejce na realizację połączenia.

Realizacja usługi zrealizowana została w nieskończonej pętli *for*:

```
while (true) {  
    socket = server.accept(); // uśpienie serwera  
    ... // tutaj obsługa usługi, pobranie strumieni z gniazda  
    ... // odebranie i wysłanie odpowiedzi  
}
```

Serwer przechodzi w stan uśpienia (za pomocą metody *accept*) po czym budzi się i realizuje usługę. Po zakończeniu ponownie wchodzi w stan uśpienia. Jeżeli kolejny klient zgłosił wcześniej żądanie (czeka w kolejce) serwer ponownie budzi się i realizuje usługę.

Realizacja usługi w tym przykładzie polega na odebraniu komunikatu (zapytania) od klienta i wysłania odpowiedzi (zasada działania usługi http). Serwer można przetestować przy użyciu dowolnej przeglądarki internetowej podając adres *localhost* (dla serwera uruchamianego lokalnie).

Dla uproszczenia kodu źródłowego serwer odbiera jedynie pierwszą linię żądania klienta.

Serwer wysyła do klienta odpowiedź wygenerowaną przez metodę `getAnswer`. Metoda ta zawiera dwie zmienne lokalne typu `String`: `document`, `header`.

Odpowiedź serwera składana jest z tych dwóch zmiennych oddzielonych pustą linią (pusta linia jest separatorem oddzielającym nagłówek odpowiedzi od dokumentu - jest podstawowa informacja dla przeglądarki):

```
return header + "\n\n" + document;
```

Niektóre fragmenty odpowiedzi serwera są (muszą być) generowane dynamicznie. W nagłówku odpowiedzi parametr *Content-Length* jest długością dokumentu w bajtach i jest wyznaczany metodą *length* klasy *String*.

Dokument zwraca bieżącą datę i godzinę (obiekt klasy *Date*), nazwę i adres hosta.

Server iteracyjny

```
import java.net.*;
import java.io.*;
import java.util.*;

public class jHTTPServer {
    private int port = 80;
    String getAnswer() {
        InetAddress adres;
        String name = "";
        String ip = "";
        try {
            adres = InetAddress.getLocalHost();
            name = adres.getHostName();

            ip = adres.getHostAddress();
        }
        catch (UnknownHostException e) { System.err.println(e); }
        String document = "<html>\r\n" +
            "<body><br>\r\n" +
            "<h2><font color=red>jHTTPServer demo document\r\n" +
            "</font></h2>\r\n" +
            "<h3>Server iteracyjny</h3><hr>\r\n" +
            "Data: <b>" + new Date() + "</b><br>\r\n" +
            "Nazwa hosta: <b>" + name + "</b><br>\r\n" +
            "IP hosta: <b>" + ip + "</b><br>\r\n" +
            "<hr>\r\n" +
            "</body>\r\n" +
            "</html>\r\n";
        String header = "HTTP/1.1 200 OK\r\n" +
            "Server: jHTTPServer ver 1.1\r\n" +
            "Last-Modified: Fri, 28 Jul 2000 07:58:55 GMT\r\n" +
            "Content-Length: " + document.length() + "\r\n" +
            "Connection: close\r\n" +
            "Content-Type: text/html";
        return header + "\r\n\r\n" + document;
    }
}
```

```
public jHTTPServer(int port){
    this.port = port;
    Socket socket = null;
    try {
        ServerSocket server = new ServerSocket(port);
        while (true) {
            socket = server.accept();
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            System.out.println("----- Pierwsza linia zapytania -----");
            System.out.println(in.readLine());
            System.out.println("----- Wysylam odpowiedz -----");
            System.out.println(getAnswer());
            System.out.println("----- Koniec odpowiedzi -----");
            out.println(getAnswer());
            out.flush();
            if (socket != null) socket.close();
        }
    } catch (IOException e) {
        System.out.println("Blad otwarcia");
    }
}

public static void main(String[] args) {
    new jHTTPServer(80);
}
```

Przykład aplikacji serwera - serwer współbieżny

Serwer *jHTTPServer* został rozbudowany (*jHTTPApp*) o możliwość przetwarzania współbieżnego (wielowątkowego).

Działanie serwera od strony klienta wygląda identycznie jak w serwerze iteracyjnym (dane zwracane do przeglądarki są podobne).

Uśpiony serwer oczekujący na zgłoszenie budzi się po czym tworzy nowy wątek aplikacji i przekazuje wątkowi realizację usługi.

W aplikacji *jHTTPApp* obiekt-serwer klasy *ServerSocket* czuwa na porcie 80 (metoda *accept*).

Po zgłoszeniu klienta obiekt *socket* klasy *Socket* przekazywany jest jako parametr konstruktora klasy *jHTTPSMulti*. Klasa ta dziedziczy po klasie *Thread* (ang. *wątek*) przeznaczonej do pracy jako dodatkowy wątek aplikacji.

```
ServerSocket server = new ServerSocket(80);  
try {  
    while (true) {  
        Socket socket = server.accept(); // uśpienie serwera  
        new jHTTPSMulti(socket); // nowy obiekt-wątek aplikacji  
    } // while  
} // try  
finally { server.close();}
```

Obsługa klienta realizowana jest już tylko w obiekcie klasy *jHTTPSMulti* natomiast serwer ponownie wchodzi w stan uśpienia (nieskończona pętla *while*). Realizację usługi wykonuje metoda *run* klasy *jHTTPSMulti* wywołana pośrednio z poziomu konstruktora metodą *start*. Po wymianie danych z klientem działanie metody *run* kończy się a tym samym kończy się działanie obiektu klasy *jHTTPSMulti* (obiekt przestaje istnieć).

```
import java.net.*;
import java.io.*;
import java.util.*;

class jHTTPSMulti extends Thread {
    private Socket socket = null;
    String getAnswer() {

        InetAddress adres;
        String name = "";
        String ip = "";
        try {
            adres = InetAddress.getLocalHost();
            name = adres.getHostName();
            ip = adres.getHostAddress();
        }
        catch (UnknownHostException ex) { System.err.println(ex); }
        String document = "<html>\r\n" +
            "<body><br>\r\n" +
            "<h2><font color=red>jHTTPApp demo document\r\n" +
            "</font></h2>\r\n" +
            "<h3>Serwer na watkach</h3><hr>\r\n" +
            "Data: <b>" + new Date() + "</b><br>\r\n" +
            "Nazwa hosta: <b>" + name + "</b><br>\r\n" +
            "IP hosta: <b>" + ip + "</b><br>\r\n" +
            "<hr>\r\n" +
            "</body>\r\n" +
            "</html>\r\n";
        String header = "HTTP/1.1 200 OK\r\n" +
            "Server: jHTTPServer ver 1.1\r\n" +
            "Last-Modified: Fri, 28 Jul 2000 07:58:55 GMT\r\n" +
            "Content-Length: " + document.length() + "\r\n" +
            "Connection: close\r\n" +
            "Content-Type: text/html; charset=iso-8859-2";
        return header + "\r\n\r\n" + document;
    }
    public jHTTPSMulti(Socket socket){
        System.out.println("Nowy obiekt jHTTPSMulti...");
        this.socket = socket;
        start();
    }
}
```

```
public void run() {
    try {
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        System.out.println("----- Pierwsza linia zapytania -----");
        System.out.println(in.readLine());
        System.out.println("----- Wysylam odpowiedz -----");
        System.out.println(getAnswer());
        System.out.println("----- Koniec odpowiedzi -----");
        out.println(getAnswer());
        out.flush();
    } catch (IOException e) {
        System.out.println("Blad IO danych!");
    }
    finally {
        try {
            if (socket != null) socket.close();
        } catch (IOException e) {
            System.out.println("Blad zamknienia gniazda!");
        }
    } // finally
}
}
```

```
public class jHTTPApp {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(80);
        try {
            while (true) {
                Socket socket = server.accept();
                new jHTTPSMulti(socket);
            } // while
        } // try
        finally { server.close(); }
    } // main
}
```