

WSPÓŁBIEŻNOŚĆ

ZAGADNIENIA:

- procesy,
- wątki,
- synchronizacja,
- synchronizacja w Java 5.0
- blokady, egzekutory, zbiory wątków

MATERIAŁY:

<http://docs.oracle.com/javase/tutorial/essential/concurrency/>



PROCESY I WĄTKI

Program w Javie uruchamiany jest w ramach **pojedynczego** procesu JVM. Z tego powodu implementacja współbieżności koncentruje się głównie na obsłudze wątków. Niemniej Java udostępnia także mechanizmy umożliwiające uruchamianie procesów systemu operacyjnego.

Proces można stworzyć korzystając z metody **Runtime.exec()** lub klasy **ProcessBuilder** z pakietu **java.lang**.

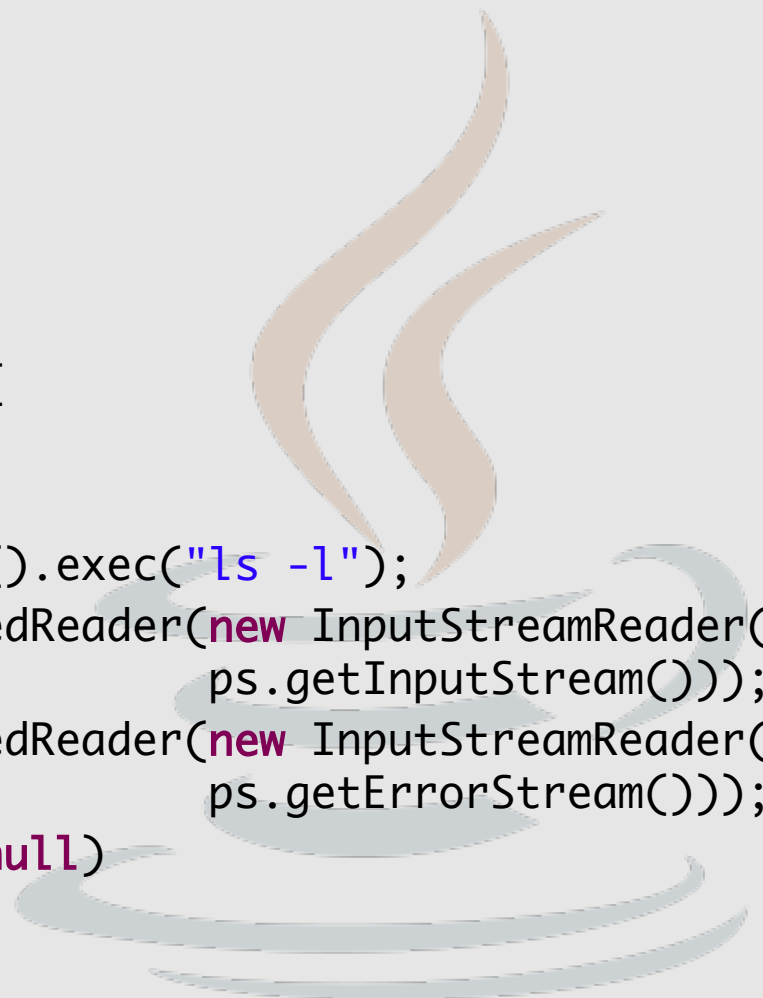
PROCESSY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ProcessExample {

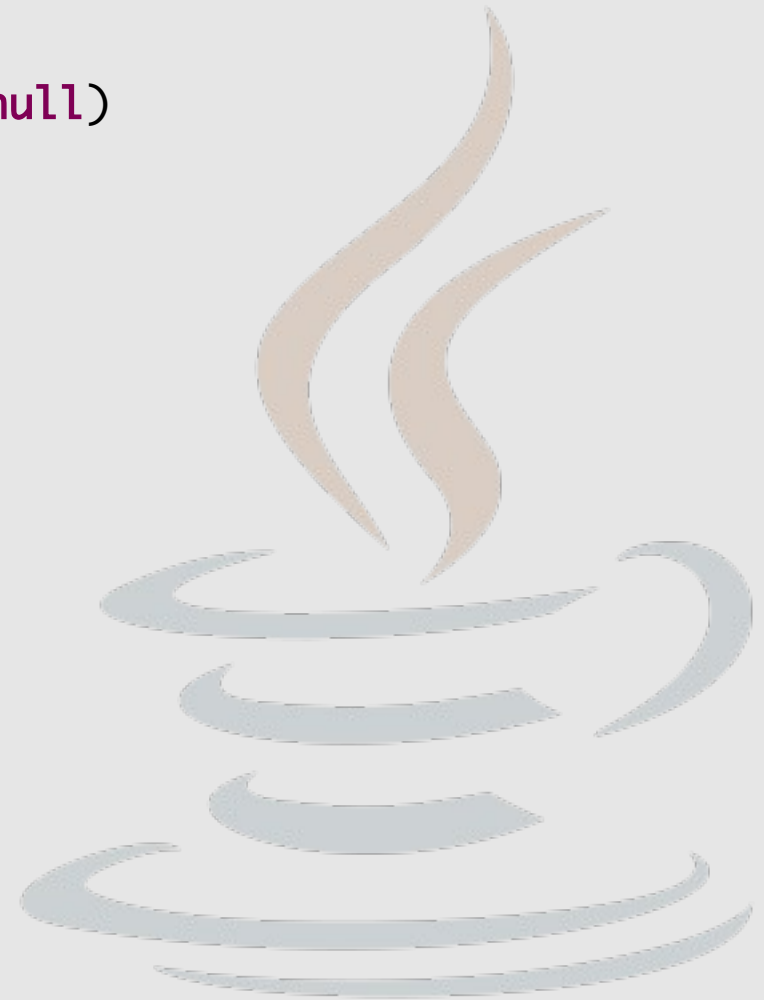
    public static void main(String[] args) {
        try {
            String s;
            Process ps = Runtime.getRuntime().exec("ls -l");
            BufferedReader bri = new BufferedReader(new InputStreamReader(
                ps.getInputStream()));
            BufferedReader bre = new BufferedReader(new InputStreamReader(
                ps.getErrorStream()));

            while ((s = bri.readLine()) != null)
                System.out.println(s);
            bri.close();
        }
    }
}
```



PROCESY

```
        while ((s = bre.readLine()) != null)
            System.out.println(s);
        bre.close();
        ps.waitFor();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Gotowe.");
}
```



PROCESY

Klasa **ProcessBuilder** pozwala precyzyjniej określić środowisko, w którym działa proces:

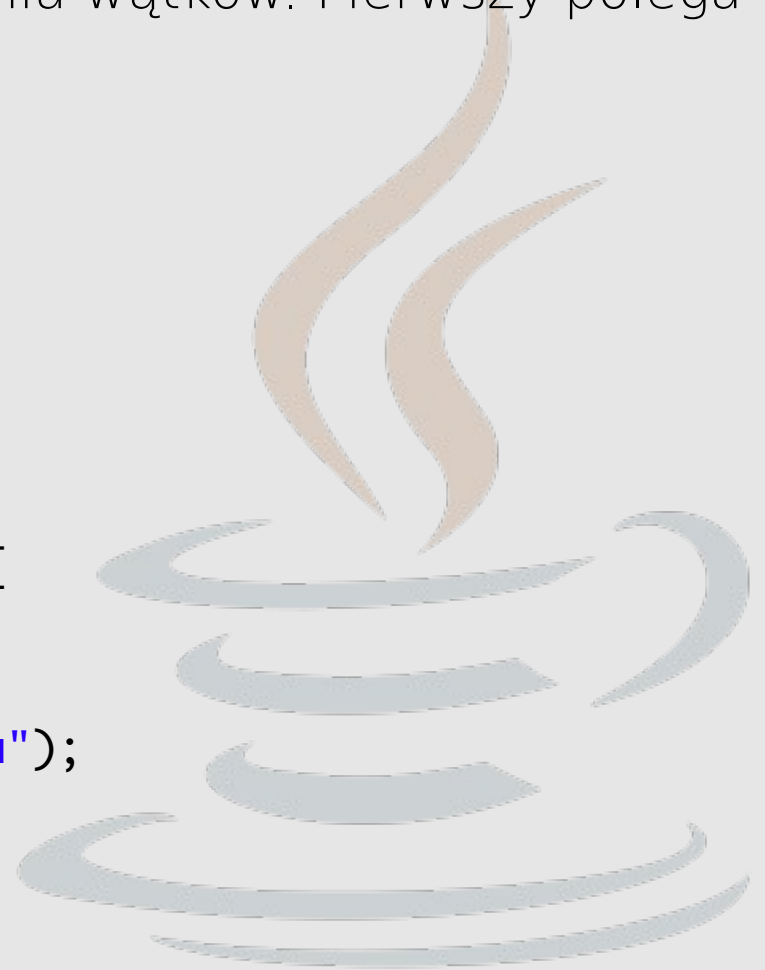
```
ProcessBuilder builder = new ProcessBuilder("ls", "-l");
builder.directory(new File("."));
builder.redirectErrorStream(true);
builder.redirectOutput(Redirect.INHERIT);
Process ps;
try {
    ps = builder.start();
    ps.waitFor();
} catch (IOException | InterruptedException e) {
    e.printStackTrace();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

Warto zajrzeć na: <http://www.rgagnon.com/javadetails/java-0014.html>

WĄTKI

Istnieją dwa podstawowe sposoby tworzenia wątków. Pierwszy polega na rozszerzeniu klasy `java.lang.Thread`:

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



WĄTKI

Drugi opiera się na skonstruowaniu wątku w oparciu o klasę implementującą interfejs `java.lang.Runnable`:

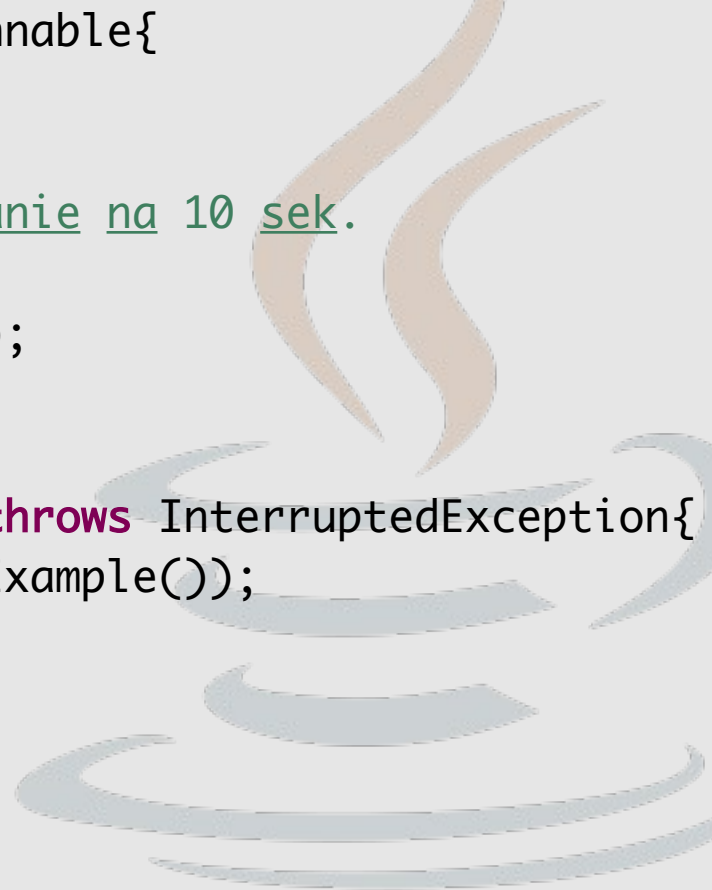
```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Witam z watku");  
    }  
  
    public static void main(String args[]) {  
        Thread t = new Thread(new HelloRunnable());  
        t.start();  
        System.out.println("Witam z programu");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

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

KLASA THREAD

Wątki mogą być wstrzymywane oraz wzbudzane/przerywane:

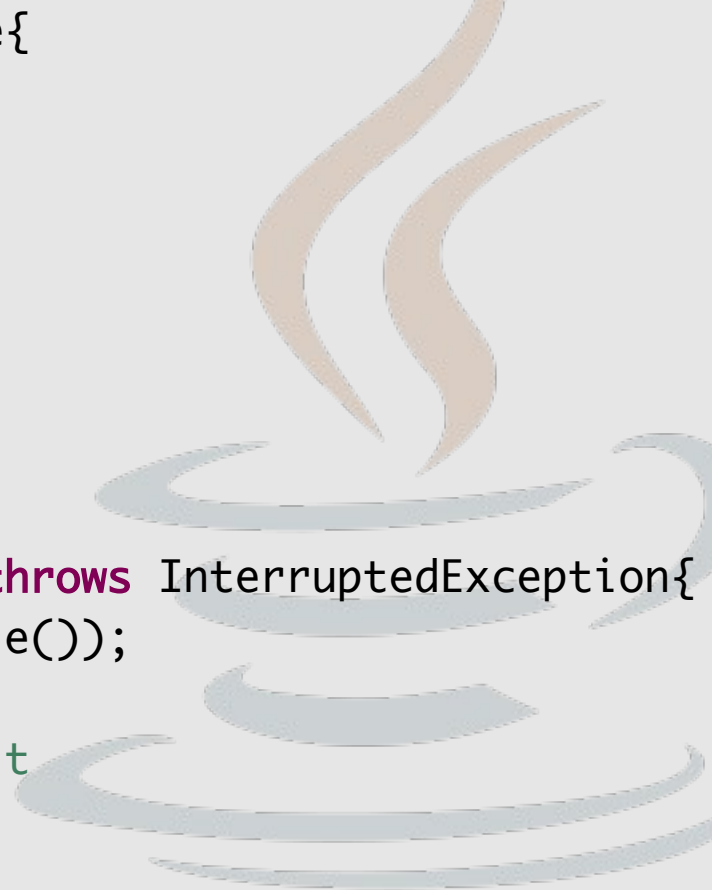
```
public class InterruptExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(10000); // wstrzymanie na 10 sek.
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
    }
    public static void main(String[] args)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



KLASA THREAD

Można również poczekać na zakończenie wskazanego wątku.

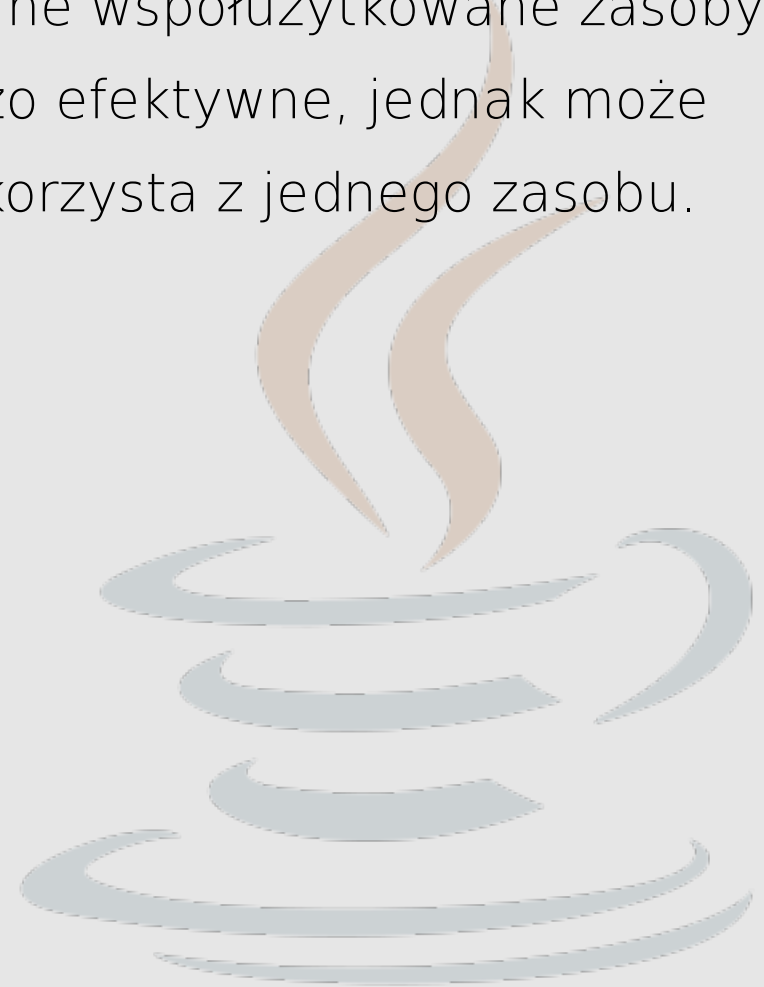
```
public class JoinExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("watek");
    }
    public static void main(String[] args) throws InterruptedException{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



SYNCHRONIZACJA

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

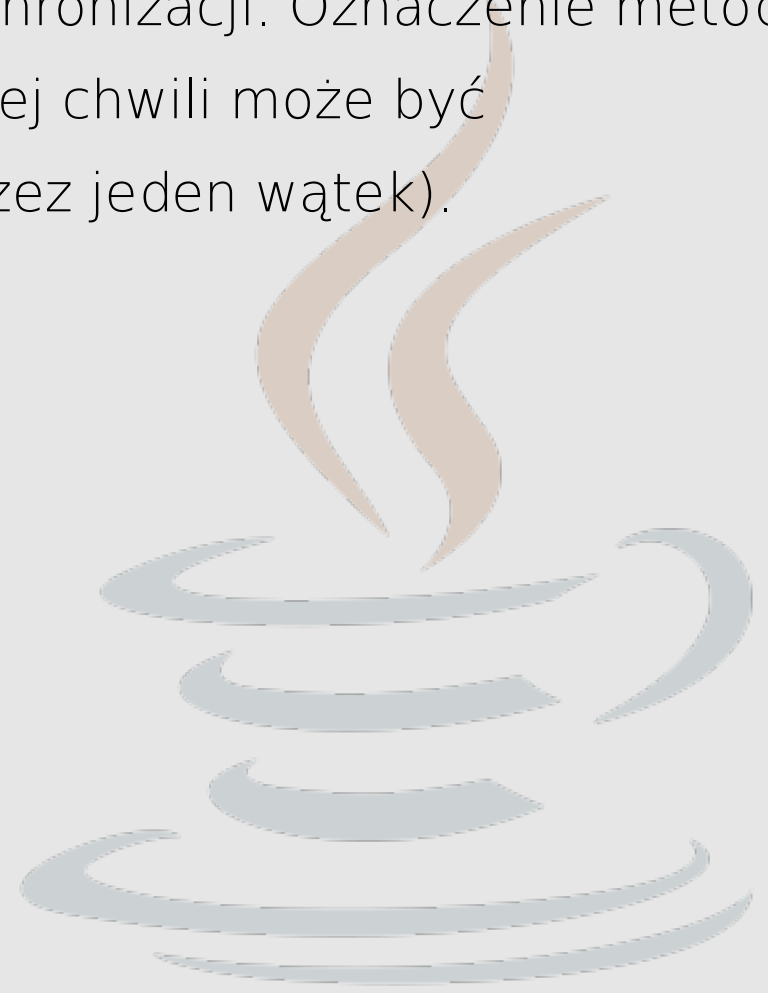
```
class Counter {  
    private int c = 0;  
    public void increment() {  
        c++;  
    }  
    public void decrement() {  
        c--;  
    }  
    public int value() {  
        return c;  
    }  
}
```



SYNCHRONIZACJA METOD

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

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



BLOKADA WEWNĘTRZNA

Każdy obiekt posiada powiązaną z nim blokadę wewnętrzną (intrinsic lock). Jeśli wątek chce uzyskać wyłączny dostęp do obiektu (lub jego atrybutów) może skorzystać z tej blokady.

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

Wątek musi jednoznacznie wskazywać obiekt, z którym jest związana używana przez niego blokada. Taki typ kłódkę nazywamy synchronizacją na poziomie instrukcji (synchronized statements)

BLOKADA DROBNOZIARNISTA

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

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



OPERACJE ATOMOWE

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

UWAGA:

```
private volatile long c1 = 0;
```

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

więcej:

http://www.javamex.com/tutorials/synchronization_volatile.shtml

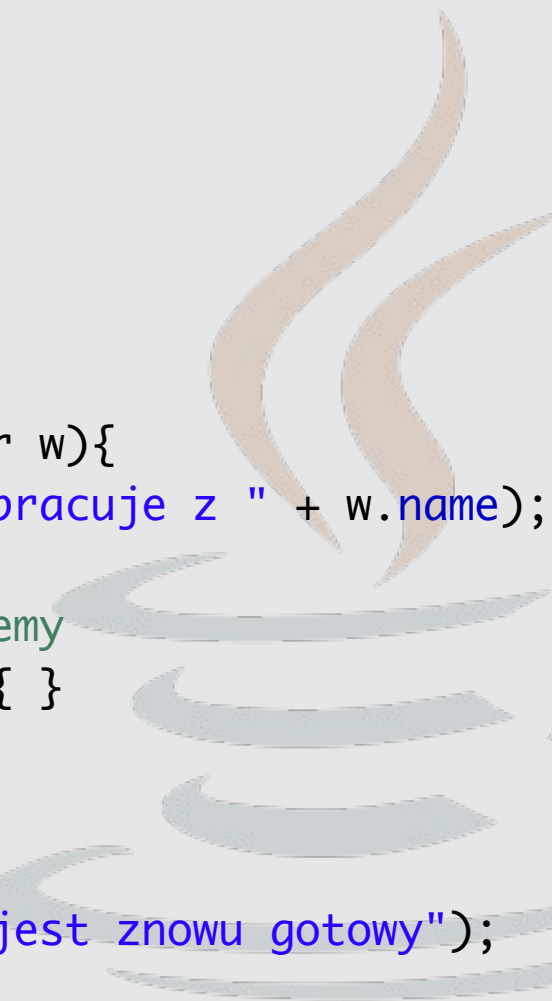
TYPOWE PROBLEMY

W programach wielowątkowych występuje kilka rodzajów problemów, które mogą powodować niewłaściwe działanie programów:

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

ZAKLESZCZENIE

```
public class Deadlock {  
  
    static class Worker {  
        public String name;  
        public Worker(String name) {  
            this.name = name;  
        }  
        public synchronized void doWork(Worker w){  
            System.out.println(this.name + " pracuje z " + w.name);  
            try {  
                Thread.sleep(1000); // pracujemy  
            } catch (InterruptedException e) { }  
            w.release();  
        }  
        public synchronized void release() {  
            System.out.println(this.name + " jest znowu gotowy");  
        }  
    }  
}
```



ZAKLESZCZENIE

```
public static void main(String[] args) {  
    final Worker w1 = new Worker("w1");  
    final Worker w2 = new Worker("w2");  
  
    new Thread(new Runnable() {  
        public void run() { w1.doWork(w2); }  
    }).start();  
  
    new Thread(new Runnable() {  
        public void run() { w2.doWork(w1); }  
    }).start();  
}  
}
```

Obaj workerzy zaczną ze sobą pracować. Blokada nastąpi na metodzie **release()** w obu obiektach.

WAIT / NOTIFY

Często wątek musi poczekać, aż inny wątek wykona określoną część swoich zadań, np. jeden wątek oblicza wyniki, a drugi je sukcesywnie wypisuje na ekranie.

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

SYNCHRONIZACJA W JRE 5.0

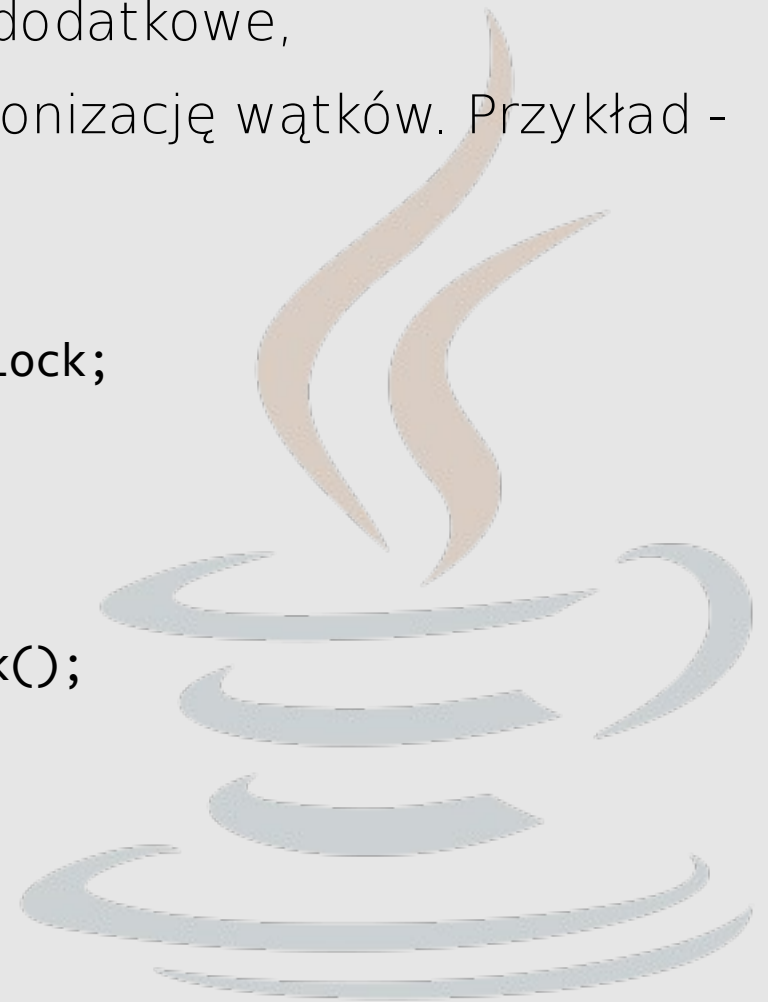
Począwszy od wersji 5.0 Java udostępnia dodatkowe, wysokopoziomowe API ułatwiające synchronizację wątków. Przykład - blokady:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockObjects {

    static class Worker {
        public Lock lock = new ReentrantLock();
        public String name;

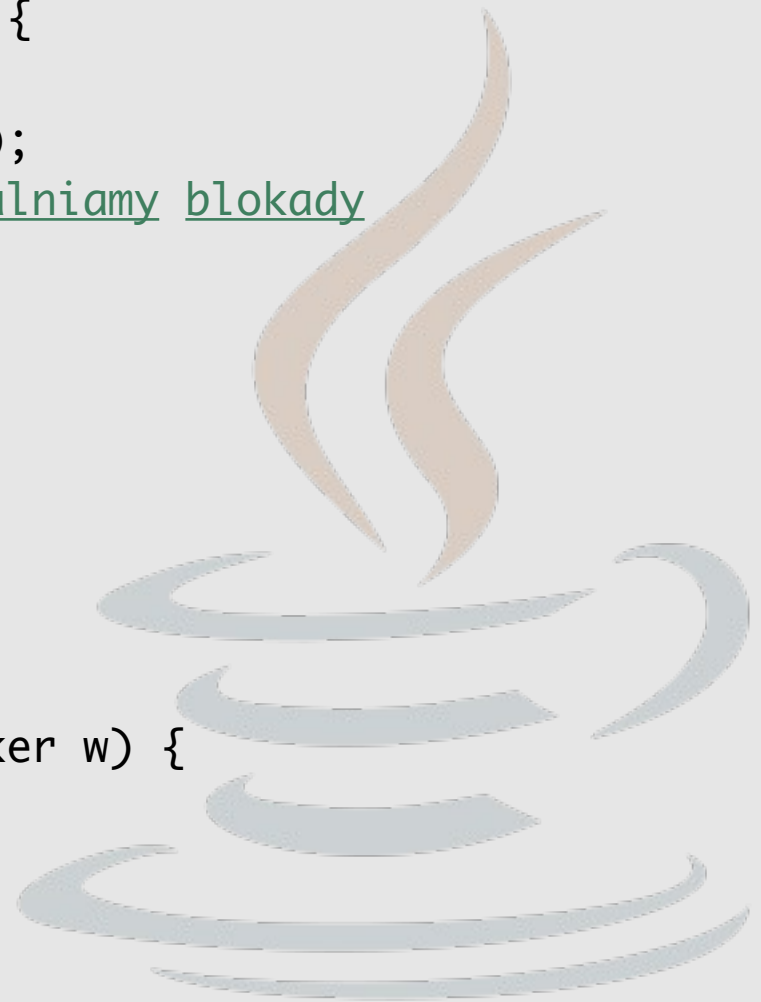
        public Worker(String name) {
            this.name = name;
        }
    }
}
```



BLOKADY

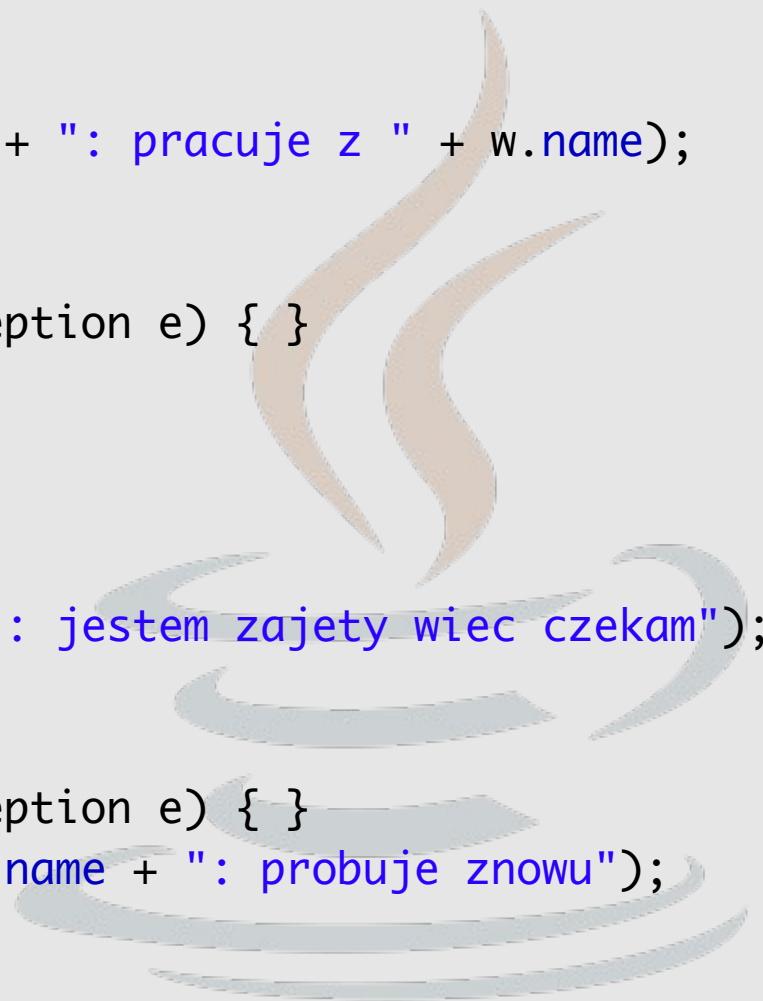
```
public boolean tryWorking(Worker w) {  
    boolean myLock = lock.tryLock();  
    boolean wLock = w.lock.tryLock();  
    if (!(myLock && wLock)) { // zwalniamy blokady  
        if (myLock)  
            lock.unlock();  
        if (wLock)  
            w.lock.unlock();  
    }  
    return myLock && wLock;  
}
```

```
public synchronized void doWork(Worker w) {  
    boolean done = false;
```



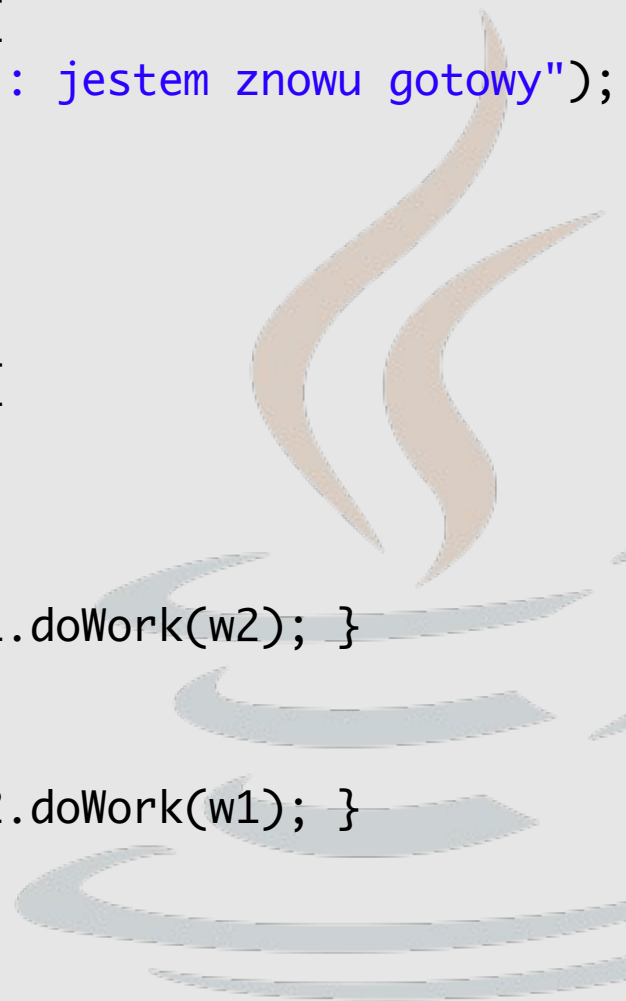
BLOKADY

```
while (!done) {  
    if (tryWorking(w)) {  
        System.out.println(name + ": pracuje z " + w.name);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) { }  
        w.release();  
        this.lock.unlock();  
        done = true;  
    } else {  
        System.out.println(name+": jestem zajety wiec czekam");  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
        System.out.println(this.name + ": probuje znowu");  
    }  
}
```



BLOKADY

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



EXECUTORS

Egzekutory oddzielają wątek od zarządzania nim

```
Executor exec = ...;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());
```

Executor jest rozszerzany przez ExecutorService, który dodatkowo umożliwia uruchamianie wątków Callable, które mogą zwracać wynik działania.

Kolejnym rodzajem egzekutorów są tzw. Thread Pools – zbiory, wspólnie zarządzanych wątków. W JRE 7.0 wprowadzono ForkJoinPool, przeznaczony do wieloprocessorowych obliczeń równoległych.

EXECUTORS

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Fibonacci extends RecursiveTask<Integer>{

    final int n;

    public Fibonacci(int n){
        this.n = n;
    }

    protected Integer compute() {
        if (n>2){
            doRecursion();
        }else{
            return 1;
        }
    }
}
```

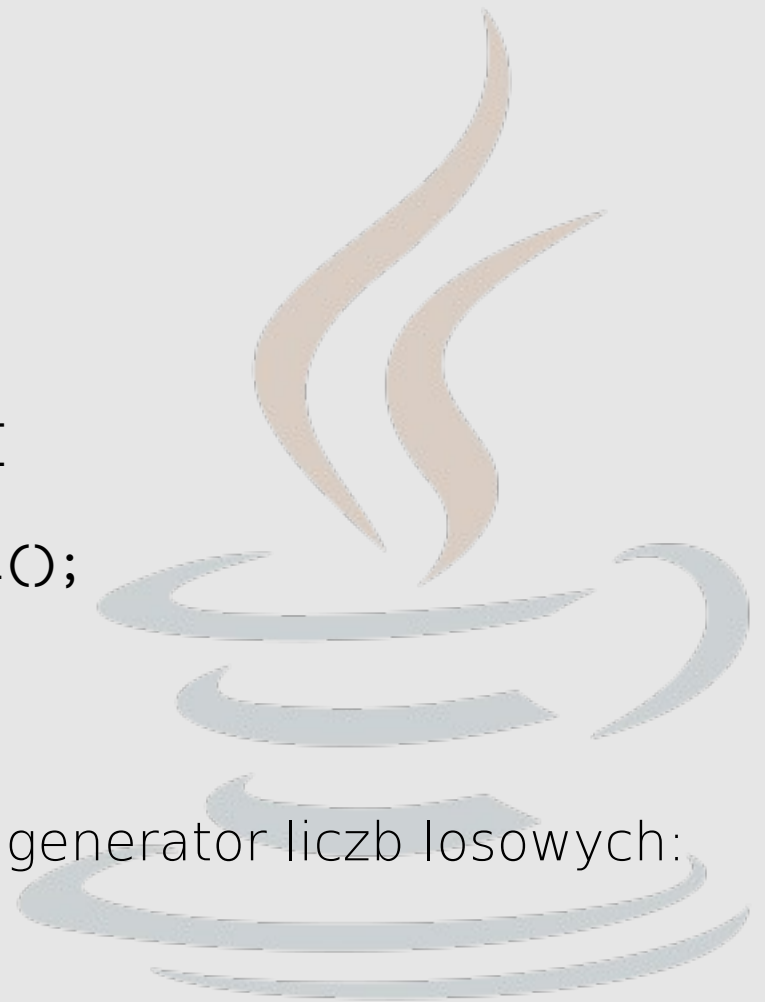


EXECUTORS

```
public doRecursion(){
    Fibonacci f1 = new Fibonacci(n-1);
    Fibonacci f2 = new Fibonacci(n-2);
    f2.fork();
    return f1.compute()+f2.join();
}

public static void main(String[] args) {
    Fibonacci f = new Fibonacci(35);
    ForkJoinPool pool = new ForkJoinPool();
    System.out.println(pool.invoke(f));
}
}
```

Ponadto wprowadzono także współbieżny generator liczb losowych:
`ThreadLocalRandom`,



DZIĘKUJĘ ZA UWAGĘ