# Java.
## Ćwiczenia zaawansowane

Autor: Marcin Lis
ISBN: 83-7197-947-9
Format: B5, stron: 142


Ćwiczenia zaawansowane są kolejnym etapem na drodze doskonalenia informatycznych umiejętności. Czytelnicy, którzy poznali poprzednią książką Marcina Lisa „Java. Ćwiczenia praktyczne", z całą pewnością nie będą zawiedzeni.

Z niniejszej publikacji dowiemy się, jak pisać programy wielowątkowe i jak w Javie obsługiwać bazy danych. Napiszemy własny, gotowy do praktycznego użycia czat oraz aplikację do wysyłania SMS-ów. Nauczymy się też tworzyć aplikacje sieciowe z interfejsem graficznym!

Wybrane zagadnienia:

- Wątki i programowanie współbieżne w Javie
- Synchronizacja wątków
- Programowanie sieciowe
- Czym są gniazda?
- Serwer wielowątkowy
- Łączenie z bazą danych
- Dodawanie rekordów
- Modyfikacja rekordów w bazie
- Obsługa transakcji i wartości null
- Aplikacja z interfejsem graficznym

Książka otwiera nową serię wydawniczą, której głównym zadaniem będzie poszerzenie uzyskanych wiadomości.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

# Spis treści

Rozdział 1.

# Wątki i programowanie współbieżne w Javie

## Klasa Thread

Wątki w Javie reprezentowane są przez klasę `Thread`, znajdującą się w pakiecie `java.lang`. Program korzystający z więcej niż jednego wątku możemy utworzyć na dwa sposoby. Albo wyprowadzimy własną, nową klasę z klasy `Thread`, albo też nasza klasa będzie musiała implementować interfejs `Runnable`. Zajmijmy się na początku metodą pierwszą. Utworzymy dwie klasy: klasę główną np. `Main` i klasę rozszerzającą klasę `Thread` np. `MyThread`. W klasie `MyThread` należy zdefiniować metodę `run()`, od której rozpocznie się działanie wątku; w klasie `Main` trzeba utworzyć obiekt klasy `MyThread` i wywołać jego metodę `start()`. Najlepiej wykonać od razu odpowiedni przykład.

### Ćwiczenie 1.1.

Napisz kod klasy `MyThread` dziedziczącej z klasy `Thread`.

```
public
class MyThread extends Thread
{
      public MyThread()
      {
               super();
      }
      public void run()
      {
               System.out.println("Thread: MyThread");
      }
}
```

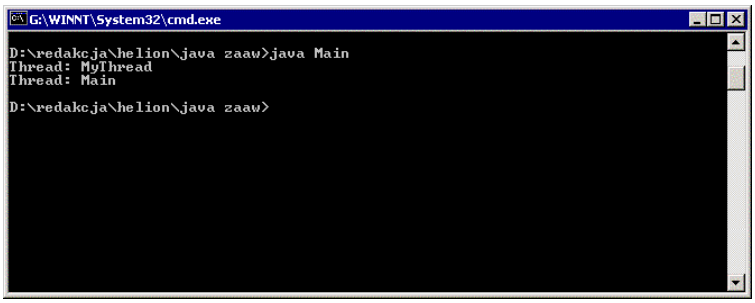
### Ćwiczenie 1.2.

Napisz kod klasy `Main` tworzący wątek `MyThread`.

```
public class Main
{
      public static void main(String args[])
      {
               MyThread myThread = new MyThread();
               myThread.start();
               System.out.println("Thread: Main");
      }
}
```

Jeśli spojrzymy teraz na rysunek 1.1, przekonamy się, że oba wątki faktycznie zostały wykonane. Podobny efekt możemy osiągnąć również w nieco inny sposób. Nie trzeba tworzyć oddzielnie klasy uruchomieniowej dla wątku (w naszym przypadku była to klasa `Main`). Wystarczy w klasie wyprowadzonej z `Thread` zdefiniować metodę `main` i tam utworzyć obiekty wątków.

**Rysunek 1.1.** *Wyraźnie widać, że oba wątki zostały wykonane*



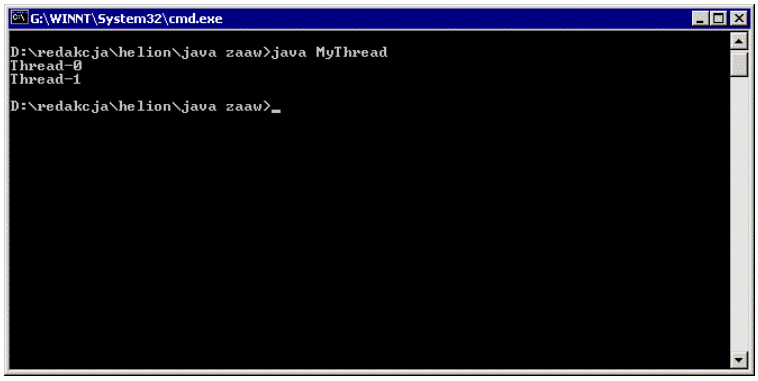

### Ćwiczenie 1.3.

Napisz kod klasy `MyThread` wyprowadzonej z klasy `Thread`, uruchamiającej dwa przykładowe wątki.

```
public
class MyThread extends Thread
{
      public MyThread()
      {
               super();
      }
      public void run()
      {
               System.out.println(getName());
      }
      public static void main(String args[])
      {
               new MyThread().start();
               new MyThread().start();
      }
}
```

Efekt działania kodu z ćwiczenia 1.3 widoczny jest na rysunku 1.2. Na ekranie wyświetlone są nazwy wątków nadane im przez system. Wykorzystaliśmy w tym celu metodę `getName()` klasy `Thread`. Warto zauważyć, że w tej chwili mamy inną sytuację niż w poprzednich przykładach. W ćwiczeniach 1.1 i 1.2 występowały dwa wątki, wątek główny i wątek klasy `MyThread`. Teraz mamy trzy wątki — wątek główny, którego wykonanie rozpoczyna się od metody `main()`, oraz dwa wątki tworzone przez nas w tej metodzie, których wykonywanie rozpoczyna się od metody `run()`.

**Rysunek 1.2.** *Na ekranie wyświetlone zostały systemowe nazwy wątków*




Gdybyśmy chcieli samodzielnie nadać nazwy poszczególnym wątkom, możemy nazwy te przekazać jako parametr w konstruktorze klasy `MyThread`, a następnie skorzystać z metody `setName()`.

### Ćwiczenie 1.4.

Zmodyfikuj kod z ćwiczenia 1.3 w taki sposób, aby istniała możliwość nadania własnych nazw wątkom.

```
public
class MyThread extends Thread
{
      public MyThread(String name)
      {
               super();
               setName(name);
      }
      public void run()
      {
               System.out.println(getName());
      }
      public static void main(String args[])
      {
               new MyThread("Wątek - 1").start();
               new MyThread("Wątek - 2").start();
      }
}
```

Spróbujmy jednak przekonać się, że rzeczywiście nasze wątki wykonują się niezależnie od siebie. Wystarczy, jeśli dopiszemy pętlę wyświetlającą kilkukrotnie nazwę każdego wątku.

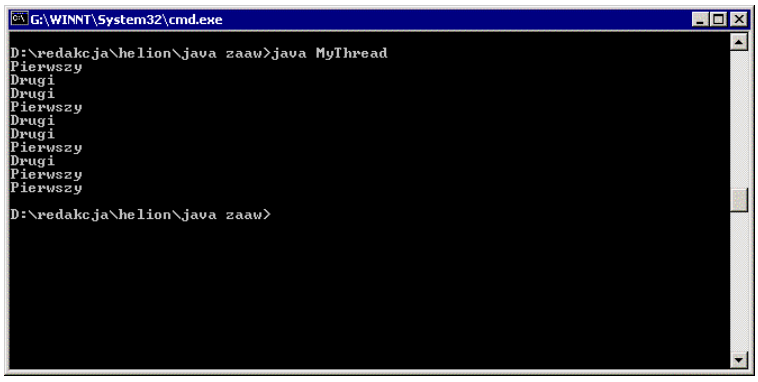
### Ćwiczenie 1.5.

Napisz przykładowy kod ilustrujący niezależne działanie wątków.

```
public
class MyThread extends Thread
{
      int delay;
      public MyThread(String name, int delay)
      {
               super();
               setName(name);
               this.delay = delay;
      }
      public void run()
      {
               for(int i = 0; i < 5; i++){
                        System.out.println(getName());
                        try{
                                 sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
               }
      }
      public static void main(String args[])
      {
               new MyThread("Pierwszy", 2).start();
               new MyThread("Drugi", 1).start();
      }
}
```

Dodatkowo „wyposażyliśmy" nasze wątki w metodę `sleep()`, która „usypia" je na zadaną ilość milisekund. Dzięki temu możemy spowodować, że każdy z nich wypisuje dane na ekran z inną prędkością. Efekt różnych prędkości działania widać wyraźnie na rysunku 1.3.

**Rysunek 1.3.** *Widać wyraźnie, że wątki wykonują się niezależnie od siebie*



# Interfejs Runnable

Wyprowadzanie własnej klasy z klasy `Thread` jest wygodne, ale nie zawsze możliwe. Z sytuacją taką będziemy mieli do czynienia, gdy nasza klasa już dziedziczy z innej, a musimy uzupełnić ją o możliwość działania wielowątkowego. Na szczęście istnieje interfejs `Runnable`, który pomoże nam w rozwiązaniu tego problemu.

W interfejsie `Runnable` zdefiniowana jest jedna metoda: `run()`, od której, podobnie jak w przypadku klasy `Thread`, rozpoczyna się wykonywanie kodu wątku. W celu uruchomienia nowego wątku tworzymy nowy obiekt naszej klasy, a następnie używamy go jako parametru konstruktora klasy `Thread`. Schematycznie wygląda to następująco (zakładając, że `MyClass` implementuje `Runnable`):

```
MyClass myClassThread new MyClass();
new Thread(myClassThread);
```

Dostępne konstruktory klasy `Thread` przedstawione są w tabeli 1.1.

**Tabela 1.1.** *Konstruktory klasy Thread*

| Konstruktor | Opis |
|---|---|
| `Thread()` | Konstruktor bezparametrowy. Tworzy nowy obiekt klasy `Thread` |
| `Thread(Runnable target)` | Tworzy nowy obiekt klasy `Thread` związany z obiektem docelowym `target` |
| `Thread(Runnable target, String name)` | Tworzy nowy obiekt klasy `Thread` związany z obiektem docelowym `target`, o nazwie `name` |
| `Thread(String name)` | Tworzy nowy obiekt klasy `Thread` o nazwie `name` |
| `Thread(ThreadGroup group, Runnable target)` | Tworzy nowy obiekt klasy `Thread` związany z obiektem docelowym `target`, przypisany do grupy `group` |
| `Thread(ThreadGroup group, Runnable target, String name)` | Tworzy nowy obiekt klasy `Thread` o nazwie `name`, związany z obiektem docelowym `target`, przypisany do grupy `group` |
| `Thread(ThreadGroup group, String name)` | Tworzy nowy obiekt klasy `Thread` o nazwie `name`, przypisany do grupy `group` |

### Ćwiczenie 1.6.

Napisz przykładowy kod ilustrujący niezależne działanie wątków. Skorzystaj z interfejsu `Runnable`.

```
public
class Main implements Runnable
{
      int delay;
      String word;
      public Main(String word, int delay)
      {
               this.delay = delay;
               this.word = word;
      }
```

```
        public void run()
        {
                for(int i = 0; i < 5; i++){
                        System.out.println(word + " " + i);
                        try{
                                Thread.sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                }
        }
        public static void main(String args[])
        {
                Runnable thread1 = new Main("thread1", 3);
                Runnable thread2 = new Main("thread2", 1);
                new Thread(thread1).start();
                new Thread(thread2).start();
        }
}
```

Dobrym przykładem wykorzystania interfejsu `Runnable` jest klasa posiadająca interfejs graficzny. Nie możemy w takim przypadku dziedziczyć bezpośrednio z klasy `Thread`, gdyż np. utworzenie okna wymaga dziedziczenia z klasy `Frame`, a wielodziedziczenia w Javie nie ma. Stosujemy więc interfejs `Runnable`, który rozwiązuje nasz problem. Postarajmy się zatem napisać prostą aplikację z interfejsem graficznym, która będzie wykonywała przykładowe obliczenia w osobnym wątku.

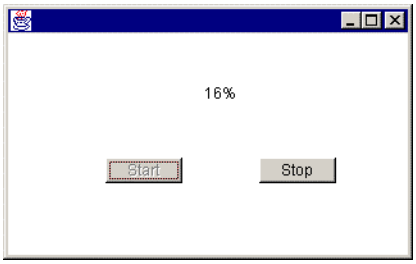
### Ćwiczenie 1.7.

Napisz aplikację z interfejsem graficznym, wykonującą w osobnym wątku przykładowe obliczenia. Stan obliczeń powinien być sygnalizowany użytkownikowi.

```
import java.awt.*;
import java.awt.event.*;

public
class Main extends Frame implements Runnable, ActionListener, WindowListener
{
      private String whichThread;
      protected static Button bStart;
      protected static Button bStop;
      protected static Label lProgress;
      protected static boolean stopped;
      public Main(String whichThread)
      {
                super();
                this.whichThread = whichThread;
                if(!"main".equals(whichThread)){
                        return;
                }
                addWindowListener(this);
                setLayout(null);
                setSize(320, 200);

                bStart = new Button("Start");
                bStart.setBounds(80, 120, 60, 20);
```

```
        bStart.addActionListener(this);
        add(bStart);

        bStop = new Button("Stop");
        bStop.setBounds(200, 120, 60, 20);
        bStop.addActionListener(this);
        add(bStop);

        lProgress = new Label("0%");
        lProgress.setBounds(155, 60, 40, 20);
        add(lProgress);

        bStart.setEnabled(true);
        bStop.setEnabled(false);

        setVisible(true);
}
public void run()
{
        stopped = false;
        for(int i = 0; i < 100; i++){
                try{
                        Thread.sleep(250);
                }
                catch(InterruptedException e){
                }
                if(stopped){
                        break;
                }
                lProgress.setText(Integer.toString(i + 1) + "%");
        }
        bStart.setEnabled(true);
        bStop.setEnabled(false);
}
public static void main(String args[])
{
        new Main("main");
}
public void actionPerformed(ActionEvent evt){
        String tmp = evt.getActionCommand();
        if (tmp.equals("Start")){
                bStart.setEnabled(false);
                Main main = new Main("compute");
                new Thread(main).start();
                bStop.setEnabled(true);
        }
        else if (tmp.equals("Stop")){
                stopped = true;
        }
}
public void windowDeiconified(WindowEvent evt)
{
}
public void windowClosed(WindowEvent evt)
{
}
public void windowDeactivated(WindowEvent evt)
{
```

```
        }
        public void windowClosing(WindowEvent evt)
        {
                System.exit(0);
        }
        public void windowActivated(WindowEvent evt)
        {
        }
        public void windowIconified(WindowEvent evt)
        {
        }
        public void windowOpened(WindowEvent evt)
        {
        }
}
```




**Rysunek 1.4.** *Po kliknięciu przycisku Start rozpoczęły się obliczenia*

W metodzie `run()` napisaliśmy zwykłą pętlę `for` symulującą wykonywanie jakichś obliczeń. Co 250 milisekund uaktualnia ona tekst etykiety `lProgress`. Wykonywanie tej operacji rozpoczyna się po naciśnięciu przycisku *Start*. Działanie pętli możemy przerwać poprzez wciśnięcie przycisku *Stop*, następuje wtedy przypisanie zmiennej `stopped` wartości `true`. Wartość tej zmiennej sprawdzana jest cyklicznie, zatem po takim przypisaniu nastąpi przerwanie działania.

Musimy tutaj zdawać sobie jednak sprawę, że mamy dwa obiekty klasy `Main`. Jeden z nich tworzony jest w metodzie `main()`, drugi po naciśnięciu przycisku *Start*. Zatem jeśli mają one ze sobą współpracować na takiej zasadzie jak przedstawiona powyżej, zmienne `bStart`, `bStop`, `lProgress` i `stopped` muszą być zadeklarowane jako statyczne. Inaczej każdy wątek będzie operował na własnej, lokalnej kopii tych zmiennych i całość oczywiście nie będzie działać. Nic nie stoi jednak na przeszkodzie, aby aplikację tę skonstruować w taki sposób, aby wątek był tworzony znanym nam już sposobem, przez oddzielną klasę, pochodną od `Thread`.

## Ćwiczenie 1.8.

Napisz kod klasy `MyThread` symulującej wykonywanie obliczeń i współpracującej z klasą `Main` realizującą interfejs graficzny.

```
public
class MyThread extends Thread
{
      int delay;
      Main parent;
      boolean stopped;
      public MyThread(Main parent, int delay)
```

```
        {
                super();
                this.delay = delay;
                this.parent = parent;
        }
        public void run()
        {
                stopped = false;
                for(int i = 0; i < 100; i++){
                        try{
                                sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        if(stopped){
                                break;
                        }
                        parent.lProgress.setText(Integer.toString(i + 1) + "%");
                }
        }
}
```

Metoda `run()` wygląda tu bardzo podobnie, jak w poprzednim ćwiczeniu. Różnice są takie, że opóźnienie jest teraz sparametryzowane, możemy je regulować wartością zmiennej `delay` (jest ona przekazywana w konstruktorze klasy) oraz że wprowadziliśmy zmienną `parent`, która jest referencją do obiektu klasy `Main` i umożliwia nam komunikację z nim. Dzięki temu możemy modyfikować tekst pojawiający się na ekranie. Musimy w tej chwili tylko przystosować klasę `Main` do współpracy z naszym nowym wątkiem.

## Ćwiczenie 1.9.

Napisz kod klasy `Main` wykorzystującej przygotowaną w ćwiczeniu 1.8 klasę wątku.

```
import java.awt.*;
import java.awt.event.*;

public
class Main extends Frame implements ActionListener, WindowListener
{
      protected Button bStart;
      protected Button bStop;
      protected Label lProgress;
      protected MyThread thread;
      public Main()
      {
                super();

                addWindowListener(this);
                setLayout(null);
                setSize(320, 200);

                bStart = new Button("Start");
                bStart.setBounds(80, 120, 60, 20);
                bStart.addActionListener(this);
                add(bStart);
```

```
                bStop = new Button("Stop");
                bStop.setBounds(200, 120, 60, 20);
                bStop.addActionListener(this);
                add(bStop);

                lProgress = new Label("0%");
                lProgress.setBounds(155, 60, 40, 20);
                add(lProgress);

                bStart.setEnabled(true);
                bStop.setEnabled(false);

                setVisible(true);
        }
        public static void main(String args[])
        {
                new Main();
        }
        public void actionPerformed(ActionEvent evt){
                String tmp = evt.getActionCommand();
                if (tmp.equals("Start")){
                        bStart.setEnabled(false);
                        thread = new MyThread(this, 250);
                        thread.start();
                        bStop.setEnabled(true);
                }
                else if (tmp.equals("Stop")){
                        thread.stopped = true;
                        bStart.setEnabled(true);
                        bStop.setEnabled(false);
                }
        }
        public void windowDeiconified(WindowEvent evt)
        {
        }
        public void windowClosed(WindowEvent evt)
        {
        }
        public void windowDeactivated(WindowEvent evt)
        {
        }
        public void windowClosing(WindowEvent evt)
        {
                System.exit(0);
        }
        public void windowActivated(WindowEvent evt)
        {
        }
        public void windowIconified(WindowEvent evt)
        {
        }
        public void windowOpened(WindowEvent evt)
        {
        }
}
```

# Synchronizacja wątków

Rozważmy w tej chwili następującą sytuację: mamy zmienną typu całkowitego i dwa wątki modyfikujące jej wartość. Załóżmy, że będzie to dodawanie w pętli w każdym przebiegu wartości *1*, a sama pętla będzie miała tych przebiegów *10*. Jaka będzie ostateczna wartość naszej zmiennej? Jeśli pierwszy wątek *10* razy zwiększył wartość o *1* i drugi wątek zrobił to samo, to w sumie powinno dać *20*. Napiszmy taki program.

### Ćwiczenie 1.10.

Napisz program, w którym dwa wątki będą niezależnie od siebie modyfikowały wartość jednej zmiennej typu `int`.

```
public
class MyThread extends Thread
{
      private int whichThread;
      private int delay;
      private static int account = 0;
      public MyThread(int whichThread, int delay)
      {
               super();
               this.delay = delay;
               this.whichThread = whichThread;
      }
      public void run()
      {
               switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
               }
      }
      public static void main(String args[])
      {
               new MyThread(1, 1).start();
               new MyThread(2, 2).start();
               new MyThread(3, 0).start();
      }
      public void thread1()
      {
               for(int i = 0; i < 10; i++){
                        try{
                                 sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        account++;
               }
      }
      public void thread2()
      {
               for(int i = 0; i < 10; i++){
                        try{
                                 sleep(delay);
```

```
                        }
                        catch(InterruptedException e){
                        }
                        account++;
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account);
        }
}
```

Wątki `thread1` i `thread2` zajmują się zwiększaniem wartości zmiennej `account`. Jedynym zadaniem wątku `thread3` jest odczekanie 1 000 milisekund i wyświetlenie wartości zmiennej `account`. Uruchomienie powyższego kodu wykaże, że faktycznie otrzymamy wartość *20*, tak jak przewidzieliśmy wcześniej. Czy zatem wszystko jest w porządku? Jak najbardziej. Co się jednak stanie, jeśli instrukcja modyfikująca `account` będzie w postaci `account = account + 1`? Napiszmy taki program.

### Ćwiczenie 1.11.

Zmodyfikuj kod z ćwiczenia 1.10 w taki sposób, aby modyfikacja zmiennej `account` była w postaci: `account = account + 1`.

```
public
class MyThread extends Thread
{
        private int whichThread;
        private int delay;
        private static int account = 0;
        public MyThread(int whichThread, int delay)
        {
                super();
                this.delay = delay;
                this.whichThread = whichThread;
        }
        public void run()
        {
                switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
                }
        }
        public static void main(String args[])
        {
                new MyThread(1, 1).start();
                new MyThread(2, 2).start();
                new MyThread(3, 0).start();
        }
```

```
        public void thread1()
        {
                for(int i = 0; i < 10; i++){
                        try{
                                sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        account = account + 1;
                }
        }
        public void thread2()
        {
                for(int i = 0; i < 10; i++){
                        try{
                                sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        account = acocunt + 1;
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account);
        }
}
```

Modyfikacje nie były duże, a po uruchomieniu ujrzymy prawdopodobnie również wynik *20*. Czy zatem wszystko znowu jest w porządku? Otóż absolutnie nie! Wszystko zależy teraz od kompilatora. Jeśli jest on „inteligentny", prawdopodobnie potraktuje instrukcję `account = account + 1` jako `account++`. W takim wypadku faktycznie program będzie prawidłowy, gdyż `account++` jest instrukcją atomową, tzn. nie może być ona przerwana przez inny wątek. Niestety nie należy przyjmować takiego założenia, natomiast trzeba traktować taki kod jako złożenie następujących operacji:

- pobranie wartości `account`,
- dodanie do tej wartości *1*,
- zapisanie otrzymanej wartości do zmiennej `account`.

Skoro tak, operacje te mogą zostać przerwane przez inny wątek. Co się wtedy stanie? Otóż otrzymany wynik na pewno nie będzie prawidłowy. Żeby się o tym przekonać, zasymulujemy przerywanie tych operacji. Zrobimy to w sposób następujący:

- wartość zmiennej `account` będziemy modyfikować w dwóch krokach,
- pomiędzy poszczególnymi operacjami dodamy instrukcję `sleep()`, usypiającą dany wątek.

Kod w każdym wątku powinien zatem wyglądać następująco:

```
int temp;
for(int i = 0; i < 10; i++){
      temp = account;
      try{
               sleep(delay);
      }
      catch(InterruptedException e){
      }
      account = temp + 1;
      System.out.println(getName() + " " + account);
}
```

## Ćwiczenie 1.12.

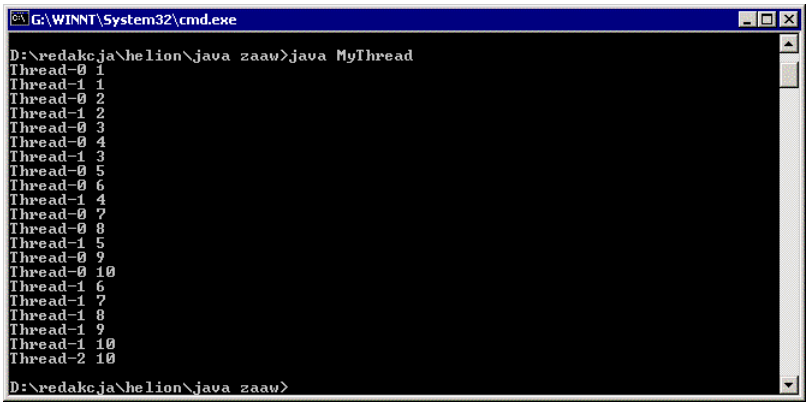
Napisz program wymuszający wzajemne przerywanie pracy wątków przy modyfikacji wspólnej zmiennej typu `int`.

```
public
class MyThread extends Thread
{
      private int whichThread;
      private int delay;
      private static int account = 0;
      public MyThread(int whichThread, int delay)
      {
               super();
               this.delay = delay;
               this.whichThread = whichThread;
      }
      public void run()
      {
               switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
               }
      }
      public static void main(String args[])
      {
               new MyThread(1, 1).start();
               new MyThread(2, 2).start();
               new MyThread(3, 0).start();
      }
      public void thread1()
      {
               int temp;
               for(int i = 0; i < 10; i++){
                        temp = account;
                        try{
                                 sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        account = temp + 1;
                        System.out.println(getName() + " " + account);
               }
```

```
        }
        public void thread2()
        {
                int temp;
                for(int i = 0; i < 10; i++){
                        temp = account;
                        try{
                                sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                        account = temp + 1;
                        System.out.println(getName() + " " + account);
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account);
        }
}
```

Instrukcje `System.out.println` dokładnie pokazują nam, co się dzieje. Wynik oczywiście nie jest prawidłowy, gdyż pomiędzy pobraniem wartości `account` a jej modyfikacją i ponownym zapisaniem w każdym wątku występuje przerwa, umożliwiająca wykonanie operacji przez inny wątek. Skutek jest taki, że — mówiąc potocznie — „nie wie lewica, co robi prawica" i wynik jest zafałszowany.

**Rysunek 1.5.** *Widać wyraźnie, że wątki sobie wzajemnie przeszkadzają*




Jest to typowy przykład dostępu do zasobu współdzielonego przez pracujące współbieżnie wątki. Aby zatem nasz przykład był poprawny, musimy dokonać ich synchronizacji. W Javie służy do tego instrukcja `synchronized`. Możemy ją stosować zarówno w przypadku metod (ang. *synchronized methods*), jak i obiektów. Jeżeli zadeklarujemy metodę jako `synchronized`, np.:

```
public synchronized void show()
{
instrukcje
}
```

to wywołanie takiej metody powoduje zablokowanie obiektu, na rzecz którego jest ona wywoływana. Obiekt ten będzie zablokowany, aż do zakończenia wykonywania tejże instrukcji i inne wątki nie będą miały do niego dostępu. Druga metoda to zablokowanie obiektu w postaci:

```
Synchronized(obiekt)
{
instrukcje
}
```

przy czym obiekt użyty do synchronizacji nie musi być użyty w bloku instrukcji. Spróbujmy zatem zsynchronizować dostęp do zmiennej `account` z poprzedniego ćwiczenia.

## Ćwiczenie 1.13.

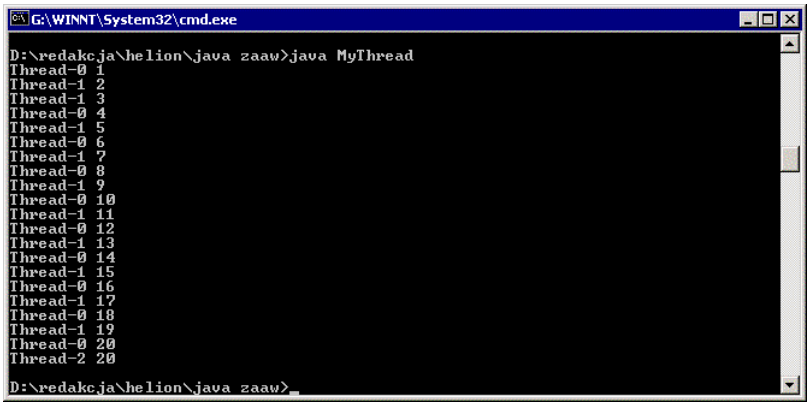
Dokonaj synchronizacji dostępu do zmiennej `account` z ćwiczenia 1.12.

```
public
class MyThread extends Thread
{
      private int whichThread;
      private int delay;
      private static int account = 0;
      private static Object semaphore;
      public MyThread(int whichThread, int delay)
      {
               super();
               this.delay = delay;
               this.whichThread = whichThread;
      }
      public void run()
      {
               switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
               }
      }
      public static void main(String args[])
      {
               semaphore = new Object();
               new MyThread(1, 1).start();
               new MyThread(2, 2).start();
               new MyThread(3, 0).start();
      }
      public void thread1()
      {
               int temp;
               for(int i = 0; i < 10; i++){
                        synchronized(semaphore){
                                 temp = account;
```

```
                                try{
                                        sleep(delay);
                                }
                                catch(InterruptedException e){
                                }
                                account = temp + 1;
                        }
                        System.out.println(getName() + " " + account);
                }
        }
        public void thread2()
        {
                int temp;
                for(int i = 0; i < 10; i++){
                        synchronized(semaphore){
                                temp = account;
                                try{
                                        sleep(delay);
                                }
                                catch(InterruptedException e){
                                }
                                account = temp + 1;
                        }
                        System.out.println(getName() + " " + account);
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account);
        }
}
```

Na rysunku 1.6 widać, że synchronizacja zakończyła się pełnym powodzeniem. Użyliśmy dodatkowego obiektu `semaphore`, który pełni rolę „strażnika" dostępu do zmiennej `account`. Jest to jego jedyna rola, do niczego innego nam w tym przykładzie nie służy. Oczywiście nic nie stoi na przeszkodzie, aby użyć obiektu, który jest wykorzystywany w kodzie programu, np. tablicy, jednakże w powyższym ćwiczeniu po prostu nie mieliśmy takiego pod ręką. Nie możemy natomiast użyć w tym celu zmiennej `account` (wszak to byłoby najwygodniejsze), gdyż jest ona typu `int`, a instrukcji `synchronized` możemy użyć tylko w stosunku do typów wyprowadzonych z klasy `Object`. Pokażmy jednak, że do synchronizacji można użyć obiektu, który będzie modyfikowany. Nie musimy wtedy wprowadzać dodatkowej zmiennej synchronizacyjnej. Aby tego dokonać, musimy napisać własną klasę enkapsulującą zmienną typu `int`. To zadanie powinno być zupełnie banalne.

**Rysunek 1.6.** *Synchronizacja powiodła się i otrzymany wynik jest teraz prawidłowy*




## Ćwiczenie 1.14.

Napisz kod klasy `Account` enkapsulującej zmienną typu `int`.

```
public
class Account
{
      public int value;
}
```

## Ćwiczenie 1.15.

Dokonaj synchronizacji dostępu do zmiennej `account` z ćwiczenia 1.12. Nie używaj dodatkowego obiektu klasy `Object`. Zamiast tego zmień typ `account` z `int` na `Account` i użyj tego obiektu do synchronizacji.

```
public
class MyThread extends Thread
{
      private int whichThread;
      private int delay;
      private static Account account;;
      public MyThread(int whichThread, int delay)
      {
               super();
               this.delay = delay;
               this.whichThread = whichThread;
      }
      public void run()
      {
               switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
               }
      }
      public static void main(String args[])
      {
               account = new Account();
               new MyThread(1, 1).start();
```

```
                new MyThread(2, 4).start();
                new MyThread(3, 0).start();
        }
        public void thread1()
        {
                int temp;
                for(int i = 0; i < 10; i++){
                        synchronized(account){
                                temp = account.value;
                                try{
                                        sleep(delay);
                                }
                                catch(InterruptedException e){
                                }
                                account.value = temp + 1;
                        }
                        System.out.println(getName() + " " + account.value);
                }
        }
        public void thread2()
        {
                int temp;
                for(int i = 0; i < 10; i++){
                        synchronized(account){
                                temp = account.value;
                                try{
                                        sleep(delay);
                                }
                                catch(InterruptedException e){
                                }
                                account.value = temp + 1;
                        }
                        System.out.println(getName() + " " + account.value);
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account.value);
        }
}
```

Jak widać, obiektem służącym do synchronizacji jest tu `account` i jednocześnie jest to obiekt, który modyfikujemy w bloku `synchronized`. Jest to bardzo wygodna metoda, gdyż nie musimy tworzyć dodatkowych zmiennych zaśmiecających system.

Skorzystajmy teraz z drugiego sposobu synchronizacji, czyli z metod synchronizowanych. Zgodnie z tym, co napisaliśmy powyżej, musimy utworzyć metodę, która będzie modyfikowała obiekt `Account` i zadeklarować ją jako `synchronized`. Może ona wyglądać w sposób następujący:

```
public static synchronized void updateAccount()
{
      int temp = account.value;
      account.value = temp + 1;
}
```

Pozostaje teraz wykorzystać ten kod w aplikacji.

### Ćwiczenie 1.16.

Dokonaj synchronizacji dostępu do zmiennej typu `Account`. Wykorzystaj synchronizowaną metodę `updateAccount()`.

```
public
class MyThread extends Thread
{
      private int whichThread;
      private int delay;
      private static Account account;;
      public MyThread(int whichThread, int delay)
      {
               super();
               this.delay = delay;
               this.whichThread = whichThread;
      }
      public void run()
      {
               switch(whichThread){
                        case 1: thread1();break;
                        case 2: thread2();break;
                        case 3: thread3();break;
               }
      }
      public static void main(String args[])
      {
               account = new Account();
               new MyThread(1, 1).start();
               new MyThread(2, 4).start();
               new MyThread(3, 0).start();
      }
      public static synchronized void updateAccount()
      {
               int temp = account.value;
               account.value = temp + 1;
      }
      public void thread1()
      {
               for(int i = 0; i < 10; i++){
                        updateAccount();
                        System.out.println(getName() + " " + account.value);
                        try{
                                 sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
               }
      }
```

```
        public void thread2()
        {
                for(int i = 0; i < 10; i++){
                        updateAccount();
                        System.out.println(getName() + " " + account.value);
                        try{
                                sleep(delay);
                        }
                        catch(InterruptedException e){
                        }
                }
        }
        public synchronized void thread3()
        {
                try{
                        wait(1000);
                }
                catch(InterruptedException e){
                        System.out.println(e);
                }
                System.out.println(getName() + " " + account.value);
        }
}
```