Rozdział 4 Programowanie współbieżne – wątki

4.1 Rys historyczny

Na początku człowiek stworzył maszynę. Dalsza historia rozwoju technologii to próby uzyskania jak największej efektywności działania maszyny. Stwierdzenie to jest również słuszne dla komputera - maszyny matematycznej.
W pierwszym okresie użytkowania komputerów, celem zwiększenia ich efektywności działania, stosowano programowanie wsadowe. W podejściu takim, programy wykonywały się cyklicznie: jeden po drugim. Komputer stał się maszyną wielozadaniową wykonującą różne operacje, np. liczył całkę, po czym rozwiązywał model fizyczny zjawiska, następnie obliczał wymaganą grubość pancerza czołgu, itd. Ponieważ typy zadań są często różne, wymagają odmiennych zasobów komputera, dlatego opracowano rozwiązanie umożliwiające dzielenie zasobów komputera pomiędzy poszczególnych użytkowników. W ten sposób każdy użytkownik otrzymywał prawo wykonania swojego programu na komputerze. Rozwiązanie to było niezwykle istotne w przypadku konieczności wykonania dwóch zadań, skrajnie obciążającego czasowo komputer oraz mało obciążającego czasowo komputer, np. policzenie parametrów skomplikowanego (wielowymiarowego) modelu i rozwiązanie układu równań. Wielozadaniowość nie stanowiła jednak wystarczająco efektywnego rozwiązania. Zaproponowano więc możliwość podziału zadań (programów) na mniejsze funkcjonalnie spójne fragmenty kodu wykonywanego (odseparowane strumienie wykonywania działań) zwane wątkami (threads). Wątek jest więc swoistym podprogramem (zbiorem wykonywanych operacji) umożliwiającym jego realizację w oderwaniu od innych wątków danego zadania. Relacja pomiędzy wątkami określana głównie poprzez dwa mechanizmy: synchronizację i zasadę pierwszeństwa. Synchronizacja wątków jest niezwykle istotnym zagadnieniem, szczególnie wówczas gdy są one ze sobą powiązane, np. korzystają z wspólnego zbioru danych. Określanie zasad pierwszeństwa - priorytetów jest pomocne wówczas, gdy efekt działania danego wątku jest o wiele bardziej istotny dla użytkownika niż efekt działania innego wątku. Oczywiście wątki mogą być generowane nie tylko jawnie przez użytkownika komputera ale również niejawnie poprzez wywołane programy komputerowe. Przykładowo dla Maszyny Wirtualnej Javy działa co najmniej kilka wątków: jeden obsługujący kod z metodą *main*(), drugi obsługujący zarządzanie pamięcią (GarbageCollection) jeszcze inny zajmujący się odświeżaniem ekranu, itp. Tworzenie wielu wątków jest docelowo związane z możliwością ich równoczesnego wykonywania (programowanie równoległe). Możliwe jest to jednak jedynie w systemach wieloprocesorowych. W większości obecnych komputerach osobistych i stacjach roboczych współbieżność wątków jest emulowana. Stosowanie podziału kodu na liczne wątki (procesy danego programu-

zadania) ma więc charakter zwiększenia efektywności działania (głównie w systemach wieloprocesorowych) oraz ma charakter porządkowania kodu (głównie w systemach jednoprocesorowych - podział zadań).
Programy Javy (zarówno aplety jak i aplikacje) są ze swej natury wielowątkowe. Świadczy o tym choćby najprostszy podział na dwa wątki: wątek obsługi kodu z metodą *main*() (dla aplikacji) oraz wątek zarządzania stertą GarbageCollection (posiadający znacznie niższy priorytet). Oczywiście programista tworząc własną aplikację może zapragnąć podzielić przepływ wykonywania działań w programie na szereg własnych wątków. Java umożliwia dokonanie takiego podziału.

4.2 Tworzenie wątków w Javie

W celu napisania kodu wątku w Javie konieczne jest albo bezpośrednie dziedziczenie z klasy *Thread* lub pośrednie poprzez implementację interfejsu *Runnable*. Wykorzystanie interfejsu jest szczególnie istotne wówczas, gdy dana klasa (klasa wątku) dziedziczy już z innej klasy, a ponieważ w Javie nie może dziedziczyć z dwóch różnych klas, dlatego konieczne jest zaimplementowanie interfejsu. Każdy wątek powinien być wywołany, mieć opisane zadania do wykonania oraz posiadać zdolność do zakończenia jego działania. Funkcjonalność tą, otrzymuje się poprzez stosowanie trzech podstawowych dla wątków metod klasy *Thread*:

start() - jawnie wywołuje rozpoczęcie wątku,
run() - zawiera zestaw zadań do wykonania,
interrupt() - umożliwia przerwanie działania wątku.

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*(). Do innych ważnych metod opisujących działanie wątków należy zaliczyć:

static int *activeCount*():
- wywołanie tej metody powoduje zwrócenie liczby wszystkich aktywnych wątków danej grupy,

static int *enumerate*(Thread[] tarray):
- wywołanie tej metody powoduje skopiowanie wszystkich aktywnych wątków danej grupy do tablicy oraz powoduje zwrócenie liczby wszystkich skopiowanych wątków,

static void *sleep* (long ms); gdzie ms to liczba milisekund:
- wywołanie tej metody powoduje uśpienie danego wątku na czas wskazany przez liczbę milisekund;

static *yield*():
- wywołanie tej metody powoduje przerwanie wykonywania aktualnego wątku kosztem wykonywania innego wątku (jeśli taki istnieje) na Maszynie Wirtualnej. Metodę tą stosowaliśmy omawiając proces GarbageCollection, zawieszając wątek działania programu na rzecz wywołania wątku zwalniania pamięci (*System.gc*()).

Ponadto istnieją jeszcze inne metody klasy *Thread*, np. *setName*(), *setDaemon*(), *setPriority*(), *getName*(), *getPriority*(), *isDaemon*(), *isAlive*(), *isInterrupt*(), *join*(), itd., które mogą być pomocne w pracy z wątkami. Część z tych metod zostanie poniżej zaprezentowana.

W celu zobrazowania możliwości generacji wątków w Javie posłużmy się następującym przykładem:

Przykład 4.1:

```
//Los.java:

import java.util.*;

class Jasnosc extends Thread {
	Thread j;
	Jasnosc(Thread j){
		this.j=j;
	}
	public void run(){
		int n=0;
		boolean b = false;
		Random r = new Random();
		do{
			if( !(this.isInterrupted())){
				n = r.nextInt();
				System.out.println("Jasność");
			} else {
				n=200000000;
				b=true;
			}
		}while(n<200000000);
		if(b){
			System.out.println(this+" jestem przerwany, kończę pracę");
		} else {
			Thread t = Thread.currentThread();
			System.out.println("Tu wątek "+t+" Jasność");
			System.out.println("Zatrzymuję wątek: "+j);
			j.interrupt();
			System.out.println("KONIEC: Jasność");
		}
	}
}// koniec class Jasnosc

class Ciemnosc extends Thread {
	Thread c;
	public void ustawC(Thread td){
		this.c=td;
	}
	public void run(){
		int n=0;
		Random r = new Random(12345678L);
		boolean b = false;
```

```
                do{
                        if( !(this.isInterrupted())){
                                n = r.nextInt();
                                System.out.println("Ciemność ");
                        } else {
                                n=200000000;
                                b=true;
                        }
                }while(n<200000000);
                if(b){
                        System.out.println(this+" jestem przerwany, kończę pracę");
                } else {
                        if (c.isAlive()) {
                                Thread t = Thread.currentThread();
                                System.out.println("Tu wątek "+t+" Ciemność");
                                System.out.println("Zatrzymuję wątek: "+c);
                                c.interrupt();
                        } else {
                                Thread t = Thread.currentThread();
                                System.out.println("Tu wątek "+t+" jestem jedyny ");
                        }
                        System.out.println("KONIEC: Ciemność");
                }
        }
}// koniec class Ciemnosc

public class Los{

  public static void main(String args[]){
        Ciemnosc zlo = new Ciemnosc();
        Jasnosc dobro = new Jasnosc(zlo);
        zlo.ustawC(dobro);
        zlo.start();
        dobro.start();
  }

}// koniec public class Los
```

Przykładowy rezultat działania powyższego programu może być następujący:

Ciemność
Jasność
Ciemność
Jasność
Ciemność
Tu wątek Thread[Thread-1,5,main] Jasność
Ciemność
Zatrzymuję wątek: Thread[Thread-0,5,main]
Ciemność
KONIEC: Jasność
Thread[Thread-0,5,main] jestem przerwany, kończę pracę

W prezentowanym przykładzie stworzono dwie klasy dziedziczące z klasy *Thread*. Obie klasy zawierają definicję metody *run*() konieczną dla pracy danego wątku. Metody *run*() zawierają generację liczb pseudolosowych w pętli, przerywanej w momencie otrzymania wartości progowej (=> 200 000 000). Dodatkowo wprowadzono w pętli *do-while*() obu metod warunek sprawdzający czy dany wątek nie został przerwany poleceniem *interrupt*(). Jeżeli wygenerowana zostanie liczba przekraczająca wartość progową to dany wątek zgłasza kilka komunikatów oraz przerywa pracę drugiego. W programie umieszczono kontrolę działania wątków (*isAlive()* - zwraca *true*, jeżeli dany wątek wykonuje jeszcze działanie zdefiniowane w jego metodzie run()), gdyż koniec wykonywania metody run() danego wątku jest równoważne z jego eliminacją. Nie można wówczas otrzymać informacji o działającym wątku poprzez domyślną konwersję „uchwytu" obiektu metodą *toString*() w ciele instrukcji *System.out.println*(). Jeśli wątek pracuje wówczas można uzyskać o nim prostą informację zawierającą dane o jego nazwie (np. Thread-0), priorytecie (np. 5) oraz o nazwie wątku naczelnego (np. *main*): np.: *Thread*[*Thread-0,5,main*]. Efekt działania powyższego programu może być różny ponieważ wartość zmiennej sprawdzanej w warunku pętli jest generowana losowo.
Kolejny przykład ukazuje możliwość nadawania nazw poszczególnym wątkom oraz ustawiania ich priorytetów:

Przykład 4.2:

```
//Widzenie.java:
//Kod klas Ciemnosc i Jasnosc musi byś dostępny dla klasy Widzenie,
//umieszczony np. w tym samym katalogu (ten sam pakiet).

import java.util.*;

public class Widzenie{

  public static void main(String args[]){
        System.out.println("Maksymalny priorytet wątku = "+Thread.MAX_PRIORITY);
        System.out.println("Minimalny priorytet wątku = "+Thread.MIN_PRIORITY);
        System.out.println("Normalny priorytet wątku = " + Thread.NORM_PRIORITY+"\n");

        Ciemnosc zlo = new Ciemnosc();
        Jasnosc dobro = new Jasnosc(zlo);
        zlo.setName("zlo");
        zlo.setPriority(4);
        dobro.setName("dobro");
        dobro.setPriority(6);
        zlo.ustawC(dobro);
        zlo.start();
        dobro.start();
 }

}// koniec public class Widzenie
```

4.3 Priorytety

W wyniku działania pierwszych trzech instrukcji powyższego programu wyświetlone są informacje o wartościach priorytetów: maksymalnej (10), minimalnej (1) i domyślnej (5). Następnie nadano nazwy poszczególnym wątkom, tak więc nie będzie już występowała nazwa domyślna tj.: *Thread-NUMER*, lecz ta ustawiona przez programistę. Wątek o nazwie „zlo" uzyskał priorytet „4", natomiast wątek o nazwie „dobro" uzyskał priorytet „6". Oznacza to, że pierwszeństwo dostępu do zasobów komputera uzyskał wątek „dobro". W rezultacie działania programu informacje generowane przez wątek „zlo" mogą się nie pojawić na ekranie monitora. Domyślna wartość priorytetu, co było zaprezentowane w przykładzie 4.1, wynosi 5. Priorytety wątków można zmieniać w czasie wykonywania działań.

Priorytety stanowią pewien problem z punktu widzenia uniwersalności kodu w Javie. Otóż system priorytetów danej platformy (systemu operacyjnego) musi być odwzorowany na zakres 10 stanów. Przykładowo dla systemu operacyjnego Solaris liczba priorytetów wynosi 2^{31}, podczas gdy dla Windows NT aktualnych priorytetów jest praktycznie 7. Trudno jest więc ocenić czy ustawiając priorytet w Javie na 9 uzyskamy dla Windows NT priorytet 5, 6 czy może 7. Co więcej poprzez mechanizm zwany „*priority boosting*" Windows NT może zmienić priorytet wątku, który związany jest z wykonywaniem operacji wejścia/wyjścia. Oznacza to, że nie można wykorzystywać uniwersalnie priorytetów do sterowania lub inaczej do wyzwalania wątków. Co może zrobić programista aby zastosować priorytety do sterowania wątkami? Może ograniczyć się do użycia stałych *Thread.MIN_PRIORITY*, *Thread.NORM_PRIORITY* oraz *Thread.MAX_PRIORITY*.

4.4 Przetwarzanie współbieżne a równoległe

Generalnie możliwe są dwa sposoby przetwarzania wątków: współbieżnie lub równolegle. Przetwarzanie współbieżne oznacza wykonywanie kilku zadań przez procesor w tym samym czasie poprzez przeplatanie wątków (fragmentów zadań). Przetwarzanie równoległe natomiast oznacza wykonywanie kilku zadań w tym samym czasie przez odpowiednią liczbę procesorów równą ilości zadań. Teoretycznie Java umożliwia jedynie przetwarzanie współbieżne ponieważ wszystkie wątki są wykonywane w otoczeniu Maszyny Wirtualnej, będącej jednym zadaniem dla danej platformy. Można oczywiście stworzyć klika kopii Maszyny Wirtualnej dla każdego procesora, i dla nich uruchamiać wątki (które mogą się komunikować). Innym rozwiązaniem przetwarzania równoległego w Javie jest odwzorowywanie wątków Maszyny Wirtualnej na wątki danej platformy (systemu operacyjnego). Oznacza to oczywiście odejście od uniwersalności kodu.

Sterowanie wątkami (przetwarzanie wątków) związane jest więc z dwoma istniejącymi modelami: wielozadaniowość kooperacyjna, bez wywłaszczania (*cooperative multitasking*) wielozadaniowość z wywłaszczaniem (szeregowania zadań - *preemptive multitasking*). Pierwszy z nich polega na tym, że dany wątek tak długo korzysta ze swoich zasobów (procesora) aż zdecyduje się zakończyć swoją pracę. Jeżeli dany wątek kończy swoje wykonywanie to uruchamiany jest wątek (przydzielane są mu zasoby) o najwyższym priorytecie wśród tych, które czekały na uruchomienie. Taki model sterowania wątkami umożliwia jedynie współbieżność wątków a wyklucza przetwarzanie równoległe. Równoległe przetwarzanie jest możliwe jedynie wtedy, gdy obsługa wątków jest wykonana w modelu szeregowania

zadań (preemptive). W modelu tym wykorzystywany jest zegar do sterowania pracą poszczególnych wątków (do przełączania pomiędzy wątkami). Przykładem systemu operacyjnego wykorzystującego pierwszy model przełączania pomiędzy wątkami jest Windows 3.1, natomiast przykładem implementacji drugiego modelu jest Windows NT. Co ciekawe Solaris umożliwia wykorzystanie obu modeli.
Wątki są odwzorowywane na procesy danej platformy według metody zależnej od systemu operacyjnego platformy. Dla NT jeden wątek jest odwzorowywany na jeden proces (odwołanie do jądra systemu - około 600 cykli maszynowych). Dla innych systemów operacyjnych może odwzorowywanie może wyglądać inaczej. Przykładowo Solaris wprowadza pojęcie tzw. *lightweight process*, oznaczające prosty proces systemu mogący zawierać jeden lub kilka wątków. Oczywiście dany proces można przypisać do konkretnego procesora (w systemie operacyjnym). Problem zarządzania wątkami i procesami to oddzielne zagadnienie. W rozdziale tym istotny jest tylko jeden wniosek dla programisty tworzącego programy w Javie: Sposób obsługi wątków (priorytety, przełączanie, odwzorowywanie na procesy, itp.) zależy wyłącznie od Maszyny Wirtualnej, czyli pośrednio od platformy pierwotnej. Praca z wątkami nie jest więc w pełni niezależna od platformy tak, jak to zakładała pradawna idea Javy: *„write once run anywhere*”.

4.5 Przerywanie pracy wątkom

Przerwanie pracy danego wątku może być rozumiane dwojako: albo jako chwilowe wstrzymanie pracy, lub jako likwidacja wątku. Likwidacja wątku może teoretycznie odbywać się na trzy sposoby: morderstwo, samobójstwo oraz śmierć naturalna. W początkowych wersjach JDK (do 1.2) w klasie *Thread* zdefiniowana była metoda stop(), która w zależności od wywołania powodowała zabicie lub samobójstwo wątku. Niestety ponieważ w wyniku wywołania metody *stop*() powstawały częste błędy związane z wprowadzeniem bibliotek DLL w środowisku Windows w stan niestabilny, dlatego metoda *stop*() uznana została za przestarzałą (jest wycofywana - *deprecated*). Likwidacja wątku może więc odbyć się jedynie poprzez jego naturalną śmierć. Naturalna śmierć wątku wyznaczana jest poprzez zakończenie działania metody run(). Konieczne jest więc zastosowanie takiej konstrukcji metody *run*(), aby można było sterować końcem pracy wątku. Najbardziej popularne są dwie metody: pierwsza wykorzystuje wiadomość przerwania pracy wątku *interrupt*(), druga bada stan przyjętej flagi. Pierwsza metoda została wykorzystana w przykładzie 4.1. Można jednak uprościć kod programu poprzez ustawienie jednego warunku, w którym będzie wykonywana treść wątku np.:

```
while(! Thread.interrupted()) {
(...)
}
```

Druga metoda związana jest z podobnym testem flagi. Wartość flagi generowana jest w wyniku działania wyrażenia. Przykładowo można testować zgodność dwóch wątków:

```
while(watekMoj==watekObecny){
(...)
}
```

wówczas można wywołać warunek końca używając metody:

```
public void stop() {
        watekObecny = null;
}
```

, lub można testować wartość zwracaną przez funkcję:

```
while(fun(arg1,arg2));
```

gdzie

```
public boolean fun(int arg1, int arg2){

        { (...)
        return true;
        }
        {(...)
        return false;
        }
}
```

itp.

Niestety nie zawsze można zakończyć pracę wątku poprzez odpowiednie ustawienie flagi. Dlaczego, otóż dlatego, że dany wątek może być blokowany ze względu na dostęp do danych, które są chwilowo niedostępne (zablokował je inny wątek). Wówczas możliwość testowania flagi pojawia się dopiero po odblokowaniu wątku (co może bardzo długo trwać). Dlatego pewną metodą zakończenia pracy wątku jest wywołanie metody *interrupt*(), która wygeneruje wyjątek dla danego wątku. Odpowiednia obsługa wyjątku może doprowadzić do zakończenia działania metody run, a więc i wątku. Należy jednak pamiętać o tym, że generacja wyjątku może spowodować taki stan pól obiektu lub klasy, który spowoduje niewłaściwe działanie programu. Jeśli takie zjawisko może wystąpić dla tworzonej aplikacji wówczas należy ustawić odpowiednie warunki kontrolne w obsłudze wyjątku, przed zakończeniem pracy wątku. Przykładowa obsługa zakończenia pracy wątku może wyglądać następująco:

```
public void run(){
        try{
                while(! (this.isInterrupted)){
                        /*wyrażenia + instrukcja generujące wyjątek IE, np. sleep(1)*/
                }
        } catch (InterruptedException ie){
        }// instrukcja pusta-> przejście do końca metody
}
```

4.6 Przerywanie tymczasowe

Możliwe jest również tymczasowe zawieszenie pracy wątku czyli wprowadzenie go w stan *Not Runnable*. Możliwe jest to na trzy sposoby:
- wywołanie metody *sleep*();
- wywołanie metody *wait*() w celu oczekiwania na spełnienie określonego warunku;
- blokowanie wątku przez operację wejścia/wyjścia (aż do jej zakończenia).

W pierwszym przypadku stosuje się jedną z metod *sleep*() zdefiniowaną w klasie *Thread*. Metoda *sleep*() umożliwia wprowadzenie w stan wstrzymania pracy wątku na określony czas podawany jako liczba milisekund stanowiąca argument wywołania metody. W czasie zaśnięcia wątku może pojawić się wiadomość przerywająca pracę wątku (*interrupt*()) dlatego konieczna jest obsługa wyjątku *InterruptedException*. Korzystanie z metody *wait*() zdefiniowanej w klasie *Object* polega na wstrzymaniu wykonywania danego wątku aż do pojawienia się wiadomości *notify*() lub *notifyAll*() wskazującej na dokonanie zmiany, na którą czekał wątek. Te trzy metody są zdefiniowane w klasie *Object* i są dziedziczone przez wszystkie obiekty w Javie. Należy tutaj wskazać na istotną różnicę pomiędzy dwoma pierwszymi sposobami tymczasowego wstrzymywania pracy wątku. Otóż wywołanie metody *sleep*() nie powoduje utraty praw (blokady) do danych (wątek dalej blokuje monitor - o czym dalej w tym rozdziale) związanych z danym wątkiem. Oznacza to, że inny wątek, który z tych danych chce skorzystać (o ile były zablokowane) nie może tego uczynić i będzie czekał na zwolnienie blokady. To, czy dany kod (zmienne) są blokowane czy nie określa instrukcja *sychronized*, która jest opisana szerzej w dalszej części tego rozdziału. Dla odróżnienia wywołanie metody *wait*() odblokowuje dane, i czeka na kolejne przejęcie tych danych (zablokowanie) po wykonaniu pewnego działania przez inne wątki.
Rozważmy następujący przykład obrazujący możliwość zatrzymywania oraz tymczasowego wstrzymywania (*sleep*) pracy wątku.

Przykład 4.3:

```
//Walka.java:

import java.util.*;

class Imperium extends Thread {
        private String glos;

        Imperium(String s){
                this.glos=s;
        }

        public void set(String s){
                this.glos=s;
        }

        public boolean imperator(){
                String mowca = Thread.currentThread().getName();
                if (mowca.equals(glos)){
                        System.out.println("Mówi Imperator !");
                        return true;
                }
```

```
                return false;
        }// koniec public boolean imperator()

        public void run(){
                while(imperator());
                System.out.println("Koniec pracy Imperium");
        }

}// koniec class Imperium

class Republika extends Thread {
        private String glos;

        Republika(String s){
                this.glos=s;
        }

        public void set(String s){
                this.glos=s;
        }

        public boolean senat(){
                String mowca = Thread.currentThread().getName();
                System.out.println("Mówi Senat !");
                        return true;
                }
                return false;
        }// koniec public boolean senat()

        public void run(){
                while(senat());
                System.out.println("Koniec pracy Senatu");
        }

}// koniec class Republika

class RadaJedi extends Thread {
        private String glos;
        private Imperium imp;
        private Republika rp;

        RadaJedi(String s, Imperium i, Republika r){
                this.glos=s;
                this.imp=i;
                this.rp=r;
        }

        public boolean rada(){
                String mowca = Thread.currentThread().getName();
                if (mowca.equals(glos)){
                        System.out.println("Zamach stanu - Rada Jedi u władzy Senat !");
                        imp.set(glos);
                        rp.set(glos);
                        try{
                                sleep(500);
```

```
                        } catch (InterruptedException ie){
                        }
                        return false;
                }
                return true;
        }// koniec public boolean imperator()

        public void run(){
                while(rada());
                System.out.println("Koniec pracy Rady");
        }

}// koniec class RadaJedi


public class Walka{

  public static void main(String args[]){
        Imperium im = new Imperium("Imperator");
        Republika rep = new Republika("Senat");
        RadaJedi rj = new RadaJedi("Rada",im,rep);
        im.setName("Imperator");
        rep.setName("Senat");
        rj.setName("Rada");

        im.start();
        rep.start();
        try{
                Thread.currentThread().sleep(6000);
        } catch (InterruptedException ie){
        }
        rj.start();
  }

}// koniec public class Walka
```

Powyższy przykład ukazuje metodę zakończenia pracy wątku w oparciu o jego śmierć naturalną spowodowaną końcem działania metody run(). Trzy klasy wątków Imperium, Republika i RadaJedi wykorzystują ten sam mechanizm zakończenia pracy metody run() poprzez ustawienie pola glos na wartość inną niż nazwa aktualnego wątku. W metodzie głównej wprowadzono również tymczasowe wstrzymanie wykonywania wątku głównego (*Thread*[*main,5,main*]) na okres 6 sekund, celem ukazania pracy pozostałych wątków. Po okresie 6 sekund uruchomiony zostaje wątek o nazwie „Rada" powodujący zakończenie pracy wszystkich stworzonych wątków przez wątek główny.
Istnieją również inne metody tymczasowego wstrzymania pracy wątku. Pierwsza z nich powoduje chwilowe wstrzymanie aktualnie wykonywanego wątku umożliwiając innym wątkom podjęcie pracy. Metoda ta opiera się o wykorzystanie poznanej już statycznej metody *yield*(). Inne przypadki wstrzymywania pracy wątku mogą być związane z blokowaniem wątku ze względu na brak dostępu do zablokowanych danych. Wątek rozpocznie dalszą pracę dopiero wtedy, gdy potrzebne dane będą dla niego dostępne. Ciekawą sytuacją może być również wstrzymanie pracy wątku

spowodowane blokowaniem danych przez przebudzony wątek o wyższym priorytecie.
Ponieważ uruchamianie wątku jest czasochłonne dla Maszyny Wirtualnej, często stosuje się metodę zwaną jako „pull threads”. Wyciąganie wątków polega na tym, że w czasie rozpoczęcia pracy programu tworzone są liczne wątki, które następnie są wprowadzane w stan czuwania i umieszczane są w pamięci (na stosie). Program w czasie pracy jeżeli potrzebuje wątku nie tworzy go, lecz pobiera z pamięci i budzi do pracy. Rozwiązanie takie jest szczególnie efektywne przy tworzeniu różnego rodzaju serwerów, kiedy czas uruchamiania programu nie jest tak istotny co czas wykonywania poleceń podczas realizacji programu. Przykładowym serwerem gdzie takie rozwiązanie może być wykorzystanie jest serwer WWW, w którym każde połączenie jest oddzielnym wątkiem.

Poniższy przykład ukazuje efekt działania metod *wait*() i *notify*() w celu tymczasowego wstrzymywania wątku i wraz ze zwolnieniem blokowanych danych w celu ich wczytania przez wątek konkurujący.

Przykład 4.4:

```
// Wojna.java:

class Strzal {
        private String strzelec = "nikt";
        private String tratata[]={ "PIF","PAF"};
        private int i=0;
        public synchronized boolean strzal(String wrog) {

                String kto = Thread.currentThread().getName();

                if (strzelec.compareTo("POKOJ") == 0)
                        return false;
                if (wrog.compareTo("POKOJ") == 0) {
                        strzelec = wrog;
                        notifyAll();
                        return false;
                }

                if (strzelec.equals("nikt")) {
                        strzelec = kto;
                         return true;
                }

                 if (kto.compareTo(strzelec) == 0) {
                         System.out.println(tratata[i]+"! ("+strzelec+")");
                         strzelec = wrog;
                         i=1-i;
                         notifyAll();
                 } else {
                 try {
                        long zwloka = System.currentTimeMillis();
                        wait(200);
                        if ((System.currentTimeMillis() - zwloka) > 200) {
                         System.out.println("Tu "+kto+", czekam na ruch osobnika:"+ strzelec);
```

```
                        }
                    } catch (InterruptedException ie) {
                    }
                }
                return true;
        }
}//koniec class Strzal


class Strzelec implements Runnable {
        Strzal s;
        String wrogPubliczny;

        public Strzelec(String wrog, Strzal st) {
                s = st;
                wrogPubliczny = wrog;
        }

        public void run() {
                while (s.strzal(wrogPubliczny)) ;
        }
}// koniec class Strzelec


public class Wojna {
        public static void main(String args[]) {
                Strzal st = new Strzal();
                Thread luke = new Thread(new Strzelec("Vader", st));
                Thread vader = new Thread(new Strzelec("Luke",st));
                luke.setName("Luke");
                vader.setName("Vader");
                luke.start();
                vader.start();

                try {
                        Thread.currentThread().sleep(1000);
                } catch (InterruptedException ie) {
                }

                st.strzal("POKOJ");
                System.out.println("Nastał pokój!!!");
        }
}// koniec public class Wojna
```

Przykładowy program skonstruowano z trzech klas. Pierwsza z nich jest odpowiedzialna za wysyłanie komunikatów w zależności od podanej nazwy zmiennej. Jeżeli nazwa strzelca jest inna niż bieżąca (czyli inna niż tego do kogo należy ruch) to następuje oczekiwanie na nowego strzelca. Druga klasa tworzy definicję metody run() konieczną dla opisania działania wątków. Ostatnia główna klasa programu tworzy obiekt klasy opisującej proces strzelania (klasy pierwszej) a następnie uruchamia dwa wątki podając nazwy wrogich strzelców względem danego wątku. Strzelanie trwa jedną sekundę po czym następuje „POKOJ”, czyli przerwanie pracy wątków poprzez zakończenie pracy metod run() (w pętli *while*() pojawia się wartość logiczna *false*). Efektem działania programu jest następujący wydruk:

```
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
PIF! (Luke)
PAF! (Vader)
Nastał pokój!!!
```

Warto wspomnieć coś na temat stanów wątku. Do podstawowych należy zaliczyć te opisywane przez metody *isAlive*() oraz *isInterrupted*(). Metoda *isAlive*() zwraca wartość logiczną *true* jeżeli dany wątek został uruchomiony (start()) i nie został jeszcze zakończony (nie umarł). Metoda *isAlive*() nie dostarcza informacji o tym, czy wykonywanie danego wątku jest zawieszone tymczasowo czy nie (brak rozróżnienia pomiędzy wątkiem *Runnable* i *Not Runnable*). Metoda *isInterrupted*() zwraca wartość *true* jeżeli dla danego wątku została przesłana wiadomość (wykonano metodę) *interrupt*().

4.7 Synchronizowanie wątków

Jeśli dwa wątki konkurują w celu uzyskania praw dostępu do danego obiektu to zachowanie kodu zależeć może od tego, który wątek wygra. Zjawisko *te (race condition* - wyścig) jest najczęściej bardzo niekorzystne ze względu na brak kontroli (nieokreśloność) w zapewnianiu kolejności obsługi wiadomości generowanych przez różne wątki do tego samego obiektu.
Załóżmy, że istnieje następująca klasa:

```
class Test{
        int n=0;
```

```
        int m=1;
        void zamianaLP(){
                n=m;
        }
        void zamianaPL(){
                m=n;
        }
}// koniec class Test
```

oraz stworzone są dwa wątki, jeden wykonuje metodę zamianaLP(), drugi wykonuje metodę zamianaPL(). O ile oba wątki są równouprawnione i wykonywane są równolegle, to problem polega na określeniu kolejności działania, czyli określeniu jakie wartości końcowe przyjmą pola n i m. obiektu klasy Test. Możliwe są tu różne sytuacje. Każda z dwóch metod wykonuje dla swoich potrzeb aktualne kopie robocze zmiennych. Następnie po wykonaniu zadania wartości tych zmiennych są przypisywane do zmiennych oryginalnych przechowywanych w pamięci głównej. Jak łatwo się domyśleć możliwe są więc następujące stany końcowe zmiennych:
- n=0,m=0; wartość zmiennej n została przepisana do m;
- n=1, m=1; wartość zmiennej m została przepisana do n;
- n=1, m=0; wartości zmiennych zostały zamienione.

W większości przypadków (poza generatorem pseudolosowym) zjawisko wyścigu jest niekorzystne. Konieczne jest więc zastosowanie takiej konstrukcji kodu aby można było to zjawisko wyeliminować. Zanim jednak zaprezentowane zostanie rozwiązanie problemu warto przeanalizować konkretny przykład.

Przykład 4.5:

//WyscigiNN.java:

```
public class WyscigiNN implements Runnable{
        private int n;

        WyscigiNN(int nr){
                this.n=nr;
        }
         void wyswietl(int i){
                System.out.println("Dobro = "+i);
                System.out.println("Zlo = "+i);
        }
         void zmianaZla(){
                 System.out.print("ZLO: ");
                 for(int l=0; l<10;l++,++n){
                         System.out.print(Thread.currentThread().getName()+": "+n+" ");
                 }
                 System.out.println(" ");
         }
         void zmianaDobra(){
                System.out.print("DOBRO: ");
                for(int l=0; l<20;l++,--n){
                         System.out.print(Thread.currentThread().getName()+": "+n+" ");
```

```
            }
            System.out.println(" ");
    }

    public void run(){
                while (!(Thread.interrupted())){
                    if ( (Thread.currentThread().getName()).equals("T1")){
                        zmianaZla();
                    } else
                        zmianaDobra();
                }
    }

    public static void main(String args[]){
            WyscigiNN w1= new WyscigiNN(100);
            Thread t1,t2;
            t1= new Thread(w1);
            t2=new Thread(w1);
            t1.setName("T1");
            t2.setName("T2");
            t2.start();
            t1.start();

            try{
                Thread.currentThread().sleep(600);
            } catch (InterruptedException ie){
            }
            t1.interrupt();
            t2.interrupt();
            try{
                Thread.currentThread().sleep(2000);
            } catch (InterruptedException ie){
            }
            System.out.println("\n");
            w1.wyswietl(w1.n);
    }
}// koniec public class WyscigiNN
```

Powyższy program umożliwia obserwację zjawiska wyścigu. Klasa główna aplikacji implementuje interfejs *Runnable* w celu definicji metody run() koniecznej dla pracy wątków. W ciele klasy głównej zadeklarowano jedną zmienną prywatną typu *int* oraz pięć metod. Pierwsza metod wyswietl() umożliwia wydruk wartości pola obiektu (zmiennej n). Druga i trzecia metoda, a więc zmianaZla() i zmianaDobra() wykonują wydruk modyfikowanej wartości pola obiektu (n). Każda drukowana wartość poprzedzana jest nazwa wątku („T1" lub T2"), który wywołał daną metodę. W metodzie *run*() w pętli warunkowej określającej koniec działania wątku (przerwanie - *interrupt*()) wywoływana jest albo metoda zmianaZla() albo zmianaDobra() w zależności od nazwy aktualnego wątku. Metoda statyczna aplikacji zawiera inicjację obiektu klasy głównej, inicjację obiektów wątków (inicjacja nie jest równoważna z rozpoczęciem pracy wątku; wątek nie jest obiektem), nadanie im nazw i ich uruchomienie. W celu obserwacji uruchomionych wątków wstrzymuje się wykonywanie wątku głównego (*main*) na okres 600 milisekund (*sleep*(600)). Następnie wysyłane są wiadomości przerwania pracy wątków, co powoduje

zakończenie działania metod run(). Kolejne uśpienie wątku głównego ma na celu wypracowanie czasu na zakończenie pracy przerywanych wątków zanim wyświetlone zostaną ostateczne rezultaty, czyli wartość pola obiektu (n). W wyniku działania programu można uzyskać następujący wydruk:

ZLO: DOBRO: T1: 100 T2: 100 T1: 101 T2: 100 T1: 101 T2: 100 T1: 100 T1: 101 T1: 102 T1: 103 T1: 104 T1: 105 T1: 106 T2: 107 ZLO: T1: 106 T1: 107 T1: 108 T1: 109 T1: 110 T1: 111 T2: 106 T1: 112 T2: 111 T1: 112 T2: 111 T1: 112 T2: 111
T2: 111 T2: 110 ZLO: T2: 109 T2: 108 T2: 107 T2: 106 T2: 105 T2: 104 T2: 103 T2: 102 T2: 101
DOBRO: T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81
DOBRO: T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T1: 73 T1: 74 T1: 75 T2: 75 T1: 76 T2: 75 T1: 76 T1: 76 T1: 77 T1: 78 T1: 79 T1: 80
ZLO: T1: 81 T1: 82 T1: 83 T1: 84 T1: 85 T1: 86 T1: 87 T1: 88 T1: 89 T1: 90
ZLO: T1: 91 T1: 92 T1: 93 T1: 94 T1: 95 T1: 96 T1: 97 T2: 75 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87
DOBRO: T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81 T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71 T2: 70 T2: 69 T2: 68 T2: 67
DOBRO: T2: 66 T2: 65 T2: 64 T2: 63 T2: 62 T2: 61 T2: 60 T2: 59 T2: 58 T2: 57 T2: 56 T1: 57 T2: 56 T1: 57 T2: 56 T1: 57 T2: 56
T2: 56 T2: 55 T2: 54 T2: 53 T2: 52 T2: 51

Dobro = 50
Zlo = 50

Łatwo zauważyć, że otrzymano nieregularne przeplatanie wątków T1 i T2, powodujące różne operację na polu obiektu. Ponieważ T1 i T2 są równomiernie uprawnione do dostępu do pola, uzyskano różne wartości (nieregularne - co uwypuklono na wydruku podkreśleniami tekstu) tego pola w czasie działania nawet tej samej pętli for metody zmianaZla() lub zamianaDobra(). Oznacza to sytuację, gdy jeden wątek korzysta z pola, które jest właśnie zmienione przez inny wątek. Ta sytuacja jest właśnie określana mianem „*race condition*”.

Konieczny jest więc tutaj mechanizm zabezpieczenia danych, w ten sposób, że jeżeli jeden wątek korzysta z nich to inne nie mogą z nich korzystać. Najprostszym rozwiązaniem byłoby zablokowanie fragmentu kodu, z którego korzysta dany wątek tak, że inne wątki muszą czekać tak długo, aż wątek odblokuje kod. Chroniony region kodu jest często nazywany monitorem (obowiązujące pojęcie w Javie). Monitor jest chroniony poprzez wzajemnie wykluczające się semafory. Oznacza to, że stworzenie monitora oraz ustawienie jego semafora (zamknięcie monitora) powoduje sytuację taką, że żaden wątek nie może z tego monitora korzystać. Jeżeli zmieni się stan semafora (zwolnienie danych) wówczas inny wątek może ten monitor sobie przywiązać (ustawić odpowiednio semafor tego wątku). W Javie blokada monitora odbywa się poprzez uruchomienie metody lub kodu oznaczonej jako *synchronized*. Jeżeli występuje dla danego wątku kod oznaczony jako *synchronized*, to kod ten staje się kodem chronionym i jego uruchomienie jest równoważne z ustanowieniem blokady na tym kodzie (o ile monitor ten nie jest już blokowany). Słowo kluczowe *synchronized* stosuje się jako oznaczenie metody (specyfikator) lub jako instrukcja. Przykładowo jeżeli metoda zmiana() przynależy do danego obiektu wówczas zapis:

```
synchronized void zmiana(){
        /*wyrażenia*/
}
```

jest równoważny praktycznie zapisowi:

```
void zmiana(){
        synchronized (this){
                /*wyrażenia*/
        }
}
```

Pierwszy zapis oznacza metodę synchronizowaną a drugi instrukcję synchronizującą. Blokada zakładana dla kodu oznaczonego poprzez *synchronized*, powoduje najpierw obliczenie odwołania (uchwytu) do danego obiektu (*this*) i założenie blokady. Wówczas żaden inny wątek nie będzie miał dostępu do monitora danego obiektu. Inaczej ujmując, żaden inny wątek nie może wykonać metody oznaczonej jako synchronized. Jeżeli zakończone zostanie działanie tak oznaczonej metody wówczas blokada jest zwalniana. Niestety stosowanie specyfikatora *synchronized* powoduje zwolnienie pracy metody nieraz i dziesięciokrotnie. Dlatego warto stosować instrukcję *synchronized* obejmując blokiem tylko to, co jest niezbędne.

W Javie (podobnie jak w innych językach programowania współbieżnego) możliwe są dwa typy obszarów działania wątku: dane dynamiczne - obiekt oraz dane statyczne. Obiekt, a więc konkretne i jednostkowe wystąpienie danej klasy jest opisywany poprzez pola i metody (w programowaniu obiektowym często nazywane z j. angielskiego jako: *instance variables* {*fields*, *methods*}). Klasa może być również opisywana poprzez pola i metody niezmienne (statyczne - *static*) dla dowolnego obiektu tej klasy (w języku angielskim elementy te nazywane są czasem *class variables{fields, methods*}). Pola i metody statyczne opisują więc stan wszystkich obiektów danej klasy, podczas gdy pola i metody obiektu opisują stan danego obiektu. Dualizm ten ma również swoje następstwa w sposobie korzystania z omawianych elementów przez wątki. Omawiany do tej pory monitor jest związany z obiektem danej klasy. Fragment kodu oznaczony przez słowo kluczowe *synchronized* (np. metoda) może być wykonywana równocześnie przez różne wątki, lecz pod warunkiem, że związane z nimi obiekty otrzymujące dane wiadomości są różne. Oznacza to, że w czasie wykonywania metody oznaczonej jako *synchronized* blokowany jest monitor danego obiektu, tak że inne wątki nie mają do niego dostępu. Poza monitorem spotyka się w literaturze [1][2][3] pojęcie kodu krytycznego - *critical section*. W [3] kod krytyczny jest definiowany jako ten, w którym wykonywany jest dostęp do tego samego obiektu z kilku różnych wątków. Kod taki, bez względu na to czy dotyczy klasy (*static*) czy obiektu oznaczany może być poprzez blok ze słowem kluczowym *synchronized*. W [2] ukazano jednak ścisły rozdział pomiędzy pojęciem *„critical section*" a monitorem: kod krytyczny to ten związany z klasą (a więc z kodem statycznym), monitor natomiast jest związany z obiektem danej klasy. Oznacza to, że kod krytyczny to taki kod, który może być wykonywany tylko poprzez jeden wątek w czasie! Nie możliwe jest bowiem stworzenie wielu obiektów z jednostkowymi metodami typu *static* (każdy obiekt widzi to samo pole lub metodę typu *static*). Oczywiście druga interpretacja jest słuszna. Dlaczego? Otóż jak już wiadomo metody i pola statyczne danej klasy należą do obiektu klasy *Class* związanego z daną metodą. Obiekt ten posiada również swój monitor, gdzie blokowanie następuje poprzez wywołanie instrukcji *synchronized static*. Maszyna Wirtualna Javy implementuje blokowanie monitora poprzez swoją instrukcję *monitorenter*, natomiast

zwalnia blokadę poprzez wywołanie instrukcji *monitorexit*. Niestety monitor obiektu klasy *Class* nie jest związany z monitorami obiektów jednostkowych danej klasy. Oznacza to, że metoda obiektu oznaczona jako *synchronized* (blokowany jest więc monitor - dostęp do obiektu) ma dostęp do pól statycznych (pola te nie należą bowiem do obiektu, czyli nie są blokowane). Rozpatrzmy następujący fragment kodu:

Przykład 4.6:

```
//KolorMiecza.java

class KolorMiecza{
        private static Color k = Color.red

        synchronized public ustawKolor(Color kolor){
                this.k=kolor;
        }
}// koniec class KolorMiecza
```

Jeżeli dwa wątki wywołają równocześnie metody ustawKolor() dla dwóch różnych obiektów klasy KolorMiecza to nie wiadomo jaką wartość będzie przechowywało pole klasy k. Występuje więc tutaj wyścig „*race condition*" czyli dwa wątki modyfikują (rywalizują o) tę samą zmienną. Oznacza to, że zablokowanie obiektu klasy KolorMiecza nie oznacza zablokowania odpowiadającemu mu obiektu klasy *Class*, czyli pola statyczne nie są wówczas blokowane. Dlatego bardzo ważne jest rozróżnienie pomiędzy monitorem (kodem obiektu) a kodem krytycznym (kodem klasy).

Na zakończenie omawiania problemu wykorzystywania instrukcji *synchronized* warto zmodyfikować prezentowany wcześniej przykład WyscigiNN.java dodając do metod zmianaZla() oraz zmianaDobra() instrukcje lub specyfikator *synchronized*.

Przykład 4.7:

```
// WyscigiSN.java:

public class WyscigiSN implements Runnable{
        private int n;

        WyscigiSN(int nr){
                this.n=nr;
        }
         void wyswietl(int i){
                System.out.println("Dobro = "+i);
                System.out.println("Zlo = "+i);
        }
         void zmianaZla(){
                synchronized (this){
                 System.out.print("ZLO: ");
                 for(int l=0; l<10;l++,++n){
                        System.out.print(Thread.currentThread().getName()+": "+n+" ");
                 }
                 System.out.println(" ");
                }
```

```
	}
	void zmianaDobra(){
		synchronized (this){
		 System.out.print("DOBRO: ");
		 for(int l=0; l<20;l++,--n){
			System.out.print(Thread.currentThread().getName()+": "+n+" ");
		 }
		 System.out.println(" ");
		}
	}

	public void run(){
			while (!(Thread.interrupted())){
				if ( (Thread.currentThread().getName()).equals("T1")){
					zmianaZla();
				} else
					zmianaDobra();
			}
	}

	public static void main(String args[]){
		WyscigiSN w1= new WyscigiSN(100);
		Thread t1,t2;
		t1= new Thread(w1);
		t2=new Thread(w1);
		t1.setName("T1");
		t2.setName("T2");
		t2.start();
		t1.start();

		try{
			Thread.currentThread().sleep(600);
		} catch (InterruptedException ie){
		}
		t1.interrupt();
		t2.interrupt();
		try{
			Thread.currentThread().sleep(2000);
		} catch (InterruptedException ie){
		}
		System.out.println("\n");
		w1.wyswietl(w1.n);
	}
}// koniec public class WyscigiSN
```

Rezultat działanie powyższego programu może być następujący:

ZLO: T1: 100 T1: 101 T1: 102 T1: 103 T1: 104 T1: 105 T1: 106 T1: 107 T1: 108 T1: 109
DOBRO: T2: 110 T2: 109 T2: 108 T2: 107 T2: 106 T2: 105 T2: 104 T2: 103 T2: 102 T2: 101 T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91
ZLO: T1: 90 T1: 91 T1: 92 T1: 93 T1: 94 T1: 95 T1: 96 T1: 97 T1: 98 T1: 99
DOBRO: T2: 100 T2: 99 T2: 98 T2: 97 T2: 96 T2: 95 T2: 94 T2: 93 T2: 92 T2: 91 T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81
ZLO: T1: 80 T1: 81 T1: 82 T1: 83 T1: 84 T1: 85 T1: 86 T1: 87 T1: 88 T1: 89
DOBRO: T2: 90 T2: 89 T2: 88 T2: 87 T2: 86 T2: 85 T2: 84 T2: 83 T2: 82 T2: 81 T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71

```
ZLO: T1: 70 T1: 71 T1: 72 T1: 73 T1: 74 T1: 75 T1: 76 T1: 77 T1: 78 T1: 79
DOBRO: T2: 80 T2: 79 T2: 78 T2: 77 T2: 76 T2: 75 T2: 74 T2: 73 T2: 72 T2: 71 T2: 70 T2: 69 T2: 68 T2: 67 T2: 66 T2: 65
T2: 64 T2: 63 T2: 62 T2: 61
ZLO: T1: 60 T1: 61 T1: 62 T1: 63 T1: 64 T1: 65 T1: 66 T1: 67 T1: 68 T1: 69
DOBRO: T2: 70 T2: 69 T2: 68 T2: 67 T2: 66 T2: 65 T2: 64 T2: 63 T2: 62 T2: 61 T2: 60 T2: 59 T2: 58 T2: 57 T2: 56 T2: 55
T2: 54 T2: 53 T2: 52 T2: 51
ZLO: T1: 50 T1: 51 T1: 52 T1: 53 T1: 54 T1: 55 T1: 56 T1: 57 T1: 58 T1: 59


Dobro = 60
Zlo = 60
```

Rezultat jasno przedstawia, że poszczególne metody są wykonywane sekwencyjnie, co oznacza blokowanie dostępu do metody (pola) przez wątek, który z niej korzysta. W wydruku wyraźnie można zaobserwować kolejne realizacje pętli for ujętych w metodach zmianaZla() i zmianaDobra().
Należy na koniec dodać, że możliwa jest sytuacja taka, że wszystkie istniejące wątki będą się wzajemnie blokować. Wówczas żaden kod nie jest wykonywany i powstaje tzw. impas, zakleszczenie (*deadlock*). Java nie dostarcza specjalnych mechanizmów detekcji impasu. Programista musi niestety przewidzieć ewentualną możliwość wystąpienia totalnej blokady, i temu zaradzić modyfikując kod.

4. 8 Grupy wątków – ThreadGroup

Poszczególne wątki można z sobą powiązać poprzez tworzenie grup wątków. Grupy wątków są przydatne ze względu na sposób organizacji pracy programu, a co za tym idzie możliwością sterowania prawami wątków. Przykładowo inne powinny być uprawnienia zawiązane z wątkami apletu niż te, związane z aplikacjami. Grupowanie wątków możliwe jest dzięki zastosowaniu klasy *ThreadGroup*. Stworzenie obiektu tej klasy umożliwia zgrupowanie nowo tworzonych wątków poprzez odwołanie się do obiektu *ThreadGroup* w konstruktorze każdego z wątków, np. *Thread(ThreadGroup tg, String nazwa).* Ponieważ w Javie wszystkie obiekty mają jedno główne źródło, również i dla obiektów typu *ThreadGroup* można przeprowadzić analizę hierarchii obiektów. Dodatkowo każdy obiekt klasy *ThreadGroup* może być jawnie stworzony jako potomek określonej grupy wątków, np. poprzez wywołanie konstruktora *ThreadGroup(ThreadGroup rodzic, String nazwa).* Maszyna Wirtualna Javy pracuje więc ze zbiorem zorganizowanymi w grupy wątków. Poniższa aplikacja ukazuje wszystkie pracujące wątki na danej platformie Javy:

Przykład 4.8:

```
//Duchy.java

public class Duchy {

  public static void main(String args[]) {

        ThreadGroup grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        Thread[] watki = new Thread[grupa.activeCount()];
```

```
            grupa.enumerate(watki);
            for (int k = 0; k < watki.length; k++) {
                        System.out.println(watki[k]);
            }
 }
}// koniec public class Duchy
```

W powyższym przykładzie zastosowano pętlę *while*, w ciele której poszukiwany jest obiekt klasy *ThreadGroup* będący na korzeniem w drzewie wszystkich grup wątków. Następnie dla tak określonego obiektu tworzona jest tablica obiektów klasy *Thread* o rozmiarze wynikającym z liczby aktywnych wątków zwracanych metodą *activeCount*(). W kolejnym kroku za pomocą metody *enumerate*() inicjowana jest tablica obiektami wątków podstawowej grupy wątków. Prezentacja otrzymanych wyników wykonana jest poprzez znaną już konwersję wątków na tekst. W rezultacie można uzyskać następujący wynik:

Thread[Signal dispatcher,5,system]
Thread[Reference Handler,10,system]
Thread[Finalizer,8,system]
Thread[main,5,main]

Metody klasy *ThreadGroup*, oprócz tych już poznanych (*activeCount*(), *enumerate*(), *getParent*()) są podobne do tych zdefiniowanych dla klasy Thread, lecz dotyczą obiektów klasy *ThreadGroup*, np. *getName*() – zwraca nazwę grupy; *interrupt*() – przerywa wszystkie wątki w grupie, itp. Ciekawą metodą jest metoda list() wysyłająca na standardowe urządzenie wyjścia (np. ekran) informacje o danej grupie. Zastosowanie tej metody w powyższym przykładzie znacznie skraca kod źródłowy:

Przykład 4.9:

```
//Duchy1.java

public class Duchy1 {

 public static void main(String args[]) {

            ThreadGroup grupa = Thread.currentThread().getThreadGroup();
            while(grupa.getParent() != null) {
                        grupa = grupa.getParent();
            }
            grupa.list();
 }
}// koniec public class Duchy1
```

W rezultacie działania tej metody można uzyskać następujący wydruk na ekranie:

java.lang.ThreadGroup[name=system,maxpri=10]
Thread[Signal dispatcher,5,system]
Thread[Reference Handler,10,system]
Thread[Finalizer,8,system]
java.lang.ThreadGroup[name=main,maxpri=10]
Thread[main,5,main]

Z wydruku tego jasno widać, że na danej platformie Javy aktywne są dwie grupy wątków: główna o nazwie „system” oraz potomna o nazwie „*main*”. W grupie głównej znajduje się między innymi wątek o nazwie „*Finalizer*”, związany z wykonywaniem zadań w czasie niszczenia obiektów. Jak wspomniano wcześniej w tym materiale zwalnianiem pamięci w *Javie* zajmuje się wątek *GarbageCollection*. Programista praktycznie nie ma na niego wpływu. Problem jednak polega na tym, że czasami konieczne jest wykonanie pewnych działań w czasie niszczenia obiektu, innych niż zwalnianie pamięci (np. zamknięcie strumienia do pliku). Obsługą takich zleceń zajmuje się wątek *Finalizer* (wykonywanie kodu zawartego w metodach *finalize*()).
Ciekawostką może być inny sposób uzyskania raportu o działających wątkach w systemie. Otóż w czasie działania programu w Javie należy w konsoli przywołać instrukcję przerwania: dla Windows – *CTRL-BREAK*, dla Solaris *kill –QUIT* (dla procesu Javy). Po instrukcji przerwania następuje wydruk stanu wątków i oczywiście przerwanie programu Javy.

4.9 Demony

Wszystkie wątki stworzone przez użytkownika giną w momencie zakończenia wątku naczelnego (zadania). Jeżeli programista pragnie stworzyć z danego wątku niezależny proces działający w tle (demon) wówczas musi dla obiektu danego wątku podać jawnie polecenie deklaracji demona: *setDaemon(true*). Metoda ta musi być wywołana zanim rozpoczęte zostanie działanie wątku poprzez zastosowanie metody start(). Poprzez odwołanie się do „uchwytu” wątku można sprawdzić czy jest on demonem czy nie. Do tego celu służy metoda *isDaemon()* zwracająca wartość *true* jeżeli dany wątek jest demonem. Rozważmy następujący przykład:

Przykład 4.10:

```
//Demony.java

public class Demony extends Thread{

  public void run(){
            while(!Thread.interrupted()){
            }
}
  public static void main(String args[]) {
            Demony d = new Demony();
            d.setName(”DEMON”);
            d.setDaemon(true);
            d.start();
            ThreadGroup grupa = Thread.currentThread().getThreadGroup();
            while(grupa.getParent() != null) {
                        grupa = grupa.getParent();
            }
            Thread[] watki = new Thread[grupa.activeCount()];
            grupa.enumerate(watki);
            for (int k = 0; k < watki.length; k++) {
                        if (watki[k].isDaemon()) System.out.println(”Demon: ”+watki[k]);
            }
```

```
            d.interrupt();
 }
}// koniec public class Demony
```

Powyższy program jest przerobioną wersją aplikacji Duchy.java tworzącą nowego demona w systemie oraz drukującą tylko te wątki, które są demonami. Wynik działania tego programu jest następujący:

Demon: Thread[Signal dispatcher,5,system]
Demon: Thread[Reference Handler,10,system]
Demon: Thread[Finalizer,8,system]
Demon: Thread[DEMON,5,main]

Okazuje się więc, że jedynym wątkiem w Javie po uruchomieniu programu (czyli bez własnych wątków) nie będącym demonem jest wątek *main*. Demon jest więc wątkiem działającym w tle. Jeżeli wszystkie istniejące wątki są demonami to Maszyna Wirtualna kończy pracę. Podobnie można oznaczyć grupę wątków jako demon. Nie oznacza to jednak, że automatycznie wszystkie wątki należące do tej grupy będą demonami.
Poniższy przykład ukazuje sytuację, w której wszystkie aktualnie wykonywane wątki przez Maszynę Wirtualną stanją się demonami. Wówczas maszyna kończy pracę.

Przykład 4.11:

```
//Duch.java

class Zly extends Thread{
        Zly(){
                super();
                setName("Zly_demon");
                //setDaemon(true);
        }
        public void run(){
                while(!this.isInterrupted()){

                }
        }
}// koniec class Zly

class Cacper extends Thread{

        Cacper(ThreadGroup g, String s){
                super(g,s);
        }

        public void run(){

                Zly z = new Zly();
                z.start();

                while(!this.isInterrupted()){

                }
```

```
        }

}//koniec class Cacper

public class Duch {

  public static void main(String args[]) {

        ThreadGroup zle_duchy = new ThreadGroup("zle_duchy");
        Cacper c = new Cacper(zle_duchy,"cacper");
        c.start();
        try{
                Thread.currentThread().sleep(4000);
        }catch (Exception e){}
        ThreadGroup grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        grupa.list();
        c.interrupt();
        try{
                Thread.currentThread().sleep(4000);
        }catch (Exception e){}

        grupa = Thread.currentThread().getThreadGroup();
        while(grupa.getParent() != null) {
                grupa = grupa.getParent();
        }
        grupa.list();
  }
}// koniec public class Duch
```

W powyższym kodzie stworzono trzy wątki: „main”-> „cacper” -> „Zly_demon”. Po określonym czasie (4 sekundy) wątek „cacper” ginie śmiercią naturalną, zostawiając pracujący wątek „Zly_demon”. Po uruchomieniu tego programu pokażą się dwa wydruki o stanie wątków, po czym program zawiesi się (wieczna pętla wątku „Zly_demon”). Co ciekawe, jeżeli wywoła się polecenie wydruku stanu wątków (CTRL-Break dla Windows) okaże się, że brak jest wątku o nazwie „main”. Jeżeli powyższy kod ulegnie zmianie w ten sposób, że wątek „Zly_demon” uzyska status demona, wówczas Maszyna Wirtualna przerwie pracę nawet jeżeli wątek „Zly_demon” wykonuje wieczne działania.

Bibliografia

[1] Sun, Java Language Specification, Sun 1998
[2] Allen Hollub, Java Toolbox: Programming Java threads in the real world, JavaWorld, http://www.javaworld.com/jw-04-1999/jw-04-toolbox.html, 1999
[3] Sun, The Java Tutorial, Sun 1998.