Jerzy Krawiec

Praktyczne aspekty programowania w Javie - wydajność programu w zakresie automatycznego zarządzania zasobami = Practical Aspect of Java Programming - Program Efficiency Related to Automatic Resource Management

Edukacja - Technika - Informatyka nr 1(19), 288-296

2017

©Wydawnictwo UR 2017 „Edukacja – Technika – Informatyka" nr 1/19/2017
ISSN 2080-9069 www.eti.rzeszow.pl
ISSN 2450-9221 online DOI: 10.15584/eti.2017.1.40

JERZY KRAWIEC

Praktyczne aspekty programowania w Javie – wydajność programu w zakresie automatycznego zarządzania zasobami

Practical Aspects of Java Programming – Program Efficiency Related to Automatic Resource Management

Doktor inżynier, Politechnika Warszawska, Wydział Inżynierii Produkcji, Instytut Organizacji Systemów Produkcyjnych, Polska

Streszczenie

Przedstawiono praktyczne aspekty programowania w języku Java. Zbadano możliwości zwiększenia wydajności programu w Javie w obrębie zarządzania zasobami. Przeprowadzono pomiary czasu wykonania programu dla operacji strumieniowych w zakresie odczytu i zapisu pliku z zastosowaniem zwykłej techniki zamykania pliku (instrukcja *try*) oraz automatycznego zarządzania zasobami (*try-with-resources*). Wykazano, że właściwie zapisany kod programu ma duży wpływ na wydajność programu Javy. Odpowiednia konstrukcja kodu Javy może znacznie skrócić czas wykonywania programu Javy.

Słowa kluczowe: programowanie**,** Java, strumień, wydajność kodu, zarządzanie zasobami

Abstract

The article discusses some aspects of the efficiency of programming in Java. Performance testing of Java code related to automatic resource management were conducted. The measurements of the runtime of the program for operation in the field of streaming read and write the file using the *try* and a *try-with-resources*. The research has shown that a properly written program code has a large impact on the efficiency of a Java program. The proper design of Java code can significantly shorten the runtime of the Java program.

Keywords: programming, Java, stream, code efficiency, resource management

Wstęp

Większość nowoczesnych języków programowania jest kompilowana do kodu wykonywalnego, co zapewnia większą wydajność systemu. Jednak w Javie wyjściem generowanym przez kompilator języka jest interpreter kodu bajtowego zwany maszyną wirtualną Javy (JVM). Interpretacja programu do kodu bajtowego znacznie ułatwia uruchomienie programu na wielu platformach (środowi-

skach), przez co uzyskuje się efekt przenośności – jedną z najważniejszych cech internetu. Takiej przenośności nie można uzyskać w przypadku kodu wykonywalnego. Ogólnie rzecz ujmując, program skompilowany do postaci pośredniej (bajtowej) i interpretowany przez JVM działa wolniej niż program skompilowany do postaci wykonywalnej (Oaks, 2014), jednak różnica w wydajności może okazać się nieznacząca, jeżeli kod programu jest odpowiednio zapisany. Już nie wystarczy nauczyć się programować np. w Javie. Teraz trzeba umieć programować efektywnie, aby aplikacja spełniła wymagania nie tylko w zakresie funkcjonalności, ale także pod względem wydajnościowym. Taki właśnie model nauki programowania jest stosowany na Wydziale Inżynierii Produkcji Politechniki Warszawskiej.

Jednym z elementów konstrukcji programistycznych mających wpływ na wydajność programu jest zarządzanie zasobami. Jednym z najważniejszych pakietów Javy jest pakiet *io*, który zawiera podstawowy podsystem WE-WY, w tym zarządzanie zasobami. Zarządzanie zasobami obejmuje przede wszystkim obsługę plików. Jedna czwarta błędów popełnianych w linii kodu to próba użycia niezwolnionej pamięci (Krawiec, 2012). Ze względu na dość skomplikowane zagadnienia wykraczające poza zakres niniejszego artykułu skoncentrujemy się tylko na podstawowych technikach zapisu i odczytu danych. Badamy wpływ automatycznego zarządzania zasobami na wydajność programu Javy. Obsługa podsystemu WE-WY w ramach pakietu *io* w Javie nie stanowi samego rdzenia tego języka, lecz jest oparta na bibliotekach. Java zapewnia elastyczną i spójną obsługę operacji WE-WY dotyczącą plików i sieci. Operacje WE-WY są wykonywane za pomocą strumieni (Schildt, 2014).

W języku Java zdefiniowano 2 rodzaje strumieni: bajtowe i znakowe. Strumień bajtowy jest stosowany do obsługi WE-WY bajtowego, np. do zapisu i odczytu danych binarnych (Downey, 2012). Natomiast strumień znaków zapewnia przekazywanie znaków, korzystając ze standardu *Unicode*, co umożliwia obsługę wielu języków. Należy podkreślić, że w pewnych sytuacjach strumienie znakowe zapewniają większą wydajność niż strumienie bajtowe. Warto o tym pamiętać, gdyż strumienie znakowe pojawiły się w Javie od wersji 1.1, a konsekwencją tego był zakaz stosowania pewnych klas i metod dotyczących strumieni bajtowych. Na najniższym poziomie komunikacja WE-WY odbywa się bajtowo, a strumienie znaków zapewniają łatwiejsze przekazywanie znaków (Gosling, 2011).

Badania wydajności programu

W Javie istnieje możliwość wykorzystania wielu klas i metod dotyczących zapisu i odczytu pliku. Zatem powszechnie wykorzystuje się klasy *FileInputStream* oraz *FileOutputStream*, które tworzą strumienie bajtów związane z plikami (Schildt, 2014). Najczęściej korzysta się z konstruktorów zdefiniowanych w następujący sposób:

```
FileInputStream(String nazwaPliku) throws FileNotFoundException
FileOuputStream(String nazwaPliku) throws FileNotFoundException
```

Po zakończeniu operacji na pliku powinien być on zamknięty za pomocą metody *close()* zdefiniowanej w wyżej wymienionych klasach w postaci:

```
void close() throws IOException
```

Zamknięcie pliku umożliwia zwolnienie zasobów związanych z tym plikiem, co stwarza możliwość wykorzystania tych zasobów w celu operacji na innym pliku. Niezamknięcie pliku może skutkować tzw. wyciekami pamięci, gdyż zasoby nieużywane nie będą zwalniane. Podstawowymi technikami stosowanymi do zamykania plików są:

– wywołanie metody *close()* dla pliku, który jest już niepotrzebny,
– użycie wyrażenia *try-with-resources*, które automatycznie zamyka niepotrzebny plik.

W celu odczytywania danych z pliku można użyć metody *read()* zdefiniowanej w klasie *FileInputStream*. Metoda ta ma następującą postać:

```
int read() throws IOException
```

Blok *try-catch* zapewnia obsługę błędów operacji WE-WY. W przypadku wystąpienia wyjątku należy go prawidłowo obsłużyć. Innym możliwym sposobem realizacji tego zadania jest wywołanie metody *close()* w bloku *finally*, co oznacza, że metody, które umożliwiają dostęp do pliku, muszą się znajdować w bloku *try*, a blok *finally* zapewnia zamknięcie tego pliku. Takie rozwiązanie gwarantuje ostateczne zamknięcie pliku, co przedstawia poniższy przykład:

```
try {
      do {
        i = abc.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException e) {
System.out.println("Błąd odczytu pliku ");
} finally {
    try {
      abc.close();
    } catch(IOException e) {
System.out.println("Błąd zamykania pliku ");
    }
  }
```

Istotną zaletą takiego rozwiązania jest pewność zamknięcia pliku w bloku *finally*. Dotyczy to nawet przypadku, gdy wykonanie kodu umożliwiającego

dostęp do pliku będzie przerwane z powodu powstania wyjątku niezwiązanego z operacjami WE-WY. Jednak w niektórych przypadkach lepszym rozwiązaniem jest zastosowanie jednego bloku *try* zawierającego wyrażenia otwierające plik i uzyskujące do niego dostęp, a następnie wykorzystanie bloku *finally* do zamknięcia wykorzystanego już pliku. W takiej sytuacji kod programu przedstawia się następująco:

```
        int i;
    FileInputStream abc = null;
        if(jekr.length != 1) {
          System.out.println("Sposób    użycia:    PokazPlik    na-
zwa_pliku");
    return;
        }
        try {
          abc = new FileInputStream(jekr[0]);
          do {
            i = abc.read();
            if(i != -1) System.out.print((char) i);
          } while(i != -1);
        } catch(IOException e) {
    System.out.println("Wystąpił błąd WE-WY");
    } finally {
          try {
            if(abc != null) abc.close();
          } catch(IOException e) {
            System.out.println("Błąd zamykania pliku");
    }
        }
      }
    }
```

Powyższy kod otwiera plik i odczytuje jego zawartość do momentu osiągnięcia końca pliku, a następnie zamyka plik w bloku *finally* pod warunkiem, że zmienna *abc* ma wartość różną od *null*. Zmienna *abc* ma wartość różną od *null* wyłącznie w sytuacji, gdy plik został otwarty prawidłowo, a to oznacza, że metoda *close()* nie zostanie wywołana, jeżeli wystąpi wyjątek w czasie otwierania pliku. Wszystkie błędy, w tym błąd podczas otwierania pliku, są obsługiwane przez wyrażenie *catch*. To rozwiązanie nie zapewni jednak reagowania na błędy podczas otwierania pliku, np. w sytuacji podania jego nieprawidłowej nazwy.

W celu zapisania danych do pliku stosuje się metodę *write()*, która jest zdefiniowana w klasie *FileOutputStream*. Metoda ta zapisuje wartość bajta do pliku i może być ujęta następująco:

```
void write(int bajt) throws IOException
```

Chociaż metoda przekazuje wartość typu *int*, zapisuje się 8 najmniej znaczących bitów liczby. Jeśli w trakcie zapisu wystąpi błąd, zgłaszany jest wyjątek *IOException*. Metoda *write()* służy do kopiowania pliku tekstowego.

Zastosowanie 2 niezależnych bloków *try* zapewnia zamknięcie plików nawet w sytuacji, gdy metoda *close()* zgłosi wyjątek. Błędy w języku Java są zgłaszane w formie wyjątków, warto więc zwrócić uwagę na obsługę potencjalnych błędów operacji WE-WY. Java stosuje mechanizm obsługi wyjątków, co zapewnia nie tylko wygodny sposób ich obsługi, ale przede wszystkim umożliwia odróżnienie błędów przy zamykaniu pliku od błędów w trakcie pobierania danych.

Jak wiadomo, do zamykania niepotrzebnego pliku stosuje się metodę *close()*. W nowym modelu Javy, począwszy od wersji JDK 7, wykorzystuje się mechanizm automatycznego zamykania zasobów zwany automatycznym zarządzaniem zasobami (ARM). Jego największą zaletą jest wyeliminowanie przypadku pozostawienia niezamkniętego pliku czy zasobu. Pozostawienie otwartego pliku, który nie jest już wykorzystywany, prowadzi do różnych poważnych problemów, np. wycieku pamięci. Automatyczne zarządzanie zasobami bazujące na rozszerzonej postaci wyrażenia *try* może być ogólnie zapisane następująco:

```
try (specyfikacja_zasobu) {
// użycie zasobu
}
```

Specyfikacja zasobu to w tym przypadku wyrażenie deklarujące i inicjujące dany zasób, np. zmienną z przypisaną do obiektu referencją. Po wykonaniu bloku *try* taki zasób jest automatycznie zwolniony, co oznacza również zamknięcie pliku – nie ma konieczności bezpośredniego wywołania metody *close()*. Nowa postać wyrażenia *try* jest określana jako *try-with-resources* i może być stosowana wyłącznie dla zasobów implementujących interfejs *AutoCloseable*. Ten interfejs definiuje tylko metodę *close()*, która jest automatycznie wywoływana w wyrażeniu *try-with-resources*, a to oznacza, że nie musi być jawnie wywoływana w kodzie programu. Interfejs *AutoCloseable* jest implementowany przez znaczną część klas, ale przede wszystkim przez klasy dotyczące operacji wejścia-wyjścia działające na strumieniach. Zmodyfikowana wersja programu pod kątem automatycznego zamykania pliku przedstawia się następująco:

```
class PokazPlik {
  public static void main(String jekr[])
  {
```

```
        int i;
        if(jekr.length != 1) {
          System.out.println("Sposób    użycia:    PokazPlik    na-
zwa_pliku");
    return;
        }
        try(FileInputStream abc = new FileInputStream(jekr[0])) {
          do {
            i = abc.read();
            if(i != -1) System.out.print((char) i);
          } while(i != -1);
        } catch(FileNotFoundException e) {
    System.out.println("Nie znaleziono pliku.");
        } catch(IOException e) {
          System.out.println("Wystąpił błąd WE-WY ");
        }
      }
    }
```

W programie zadeklarowano obiekt o nazwie *abc* klasy *FileInputStream*. Obiektowi przypisano referencję do otwartego pliku poprzez konstruktor *FileInputStream*, co oznacza, że obiekt (zmienna) *abc* ma zasięg lokalny w stosunku do bloku *try*. Po wykonaniu bloku *try* ciąg znaków odpowiadający zmiennej *abc* jest automatycznie zamykany za pomocą niejawnego wywołania metody *close()*.

Należy zaznaczyć, że zmienna zadeklarowana w bloku *try* oznacza zmienną finalną, a jej zasięg jest ograniczony wyłącznie do bloku *try-with-resources*.

Stosując wyrażenie *try*, możliwe jest zatem zarządzanie wieloma zasobami. Zmodyfikowany program do kopiowania plików zawiera pojedyncze wyrażenie *try-with-resources* do zarządzania strumieniem *abc* oraz *fgh*. W takiej sytuacji program ma następującą postać:

```
    class KopiujPlik {
      public static void main(String jekr[]) throws IOException
    {
        int i;
        if(jekr.length != 2) {
          System.out.println("Sposób   użycia:   KopiujPlik   źródło
cel");
    return;
        }
        try (FileInputStream abc = new FileInputStream(jekr[0]);
             FileOutputStream fgh = new FileOutputStream(jekr[1]))
```

```
    {
      do {
        i = abc.read();
        if(i != -1) fgh.write(i);
      } while(i != -1);
    } catch(IOException e) {
System.out.println("Błąd WE-WY:  " + e);
    }
  }
}
```

Po zakończeniu bloku *try* oba pliki reprezentowane przez zmienne *abc* i *fgh* są automatycznie zamykane. Stosowanie konstrukcji *try-with-resources* ma jeszcze jedną ważną zaletę. Otóż podczas wykonywania bloku *try* wyjątek zgłoszony w tym bloku może powodować wystąpienie innego wyjątku podczas zamykania zasobu w klauzuli *finally*. W wyrażeniu *try* pierwszy wyjątek jest tracony i zastępowany wyjątkiem zgłaszanym jako drugi. Natomiast w konstrukcji *try-with-resources* drugi wyjątek jest ukrywany i znajduje się na liście wyjątków kojarzonych z pierwszym wyjątkiem. Lista tych wyjątków jest definiowana za pomocą metody *getSuppressed()* z klasy *Throwable*.

Należy zdawać sobie sprawę z tego, że choć konstrukcja *try-with-resources* jest powszechnie stosowana, to w aplikacjach napisanych przed wersją JDK 7, a tych są miliony linii kodu, stosuje się ręczne wywołanie metody *close()*.

Pomiary przeprowadzono za pomocą wyżej wymienionych kodów źródłowych z użyciem instrukcji *try* oraz *try-with-resources* dla odczytu pustego pliku o nazwie Plik1.txt oraz kopiowania treści pustego pliku o nazwie Plik1.txt do pliku Plik2.txt. Mierzono czas wykonywania programu, wykorzystując następujący kod programu:

```
start1 = System.nanoTime();
       // Instrukcje do wykonania z try lub try-with-resources
       end1 = System.nanoTime();
                 System.out.println(end1-start1);
```

Wszystkie pomiary przeprowadzono za pomocą komputera o następujących parametrach:

- Procesor: Intel(R) Core™ i5-2520M CPU @ 2,5 GHz.
- Pamięć RAM: 4GB.
- Karta graficzna: Intel (R) HD Graphics 3000.
- Dysk: 112 GB SSD.
- System operacyjny: WINDOWS 7 64-bitowy.
- Java Platform (JDK) 8u73/8u74.

Wyniki badań

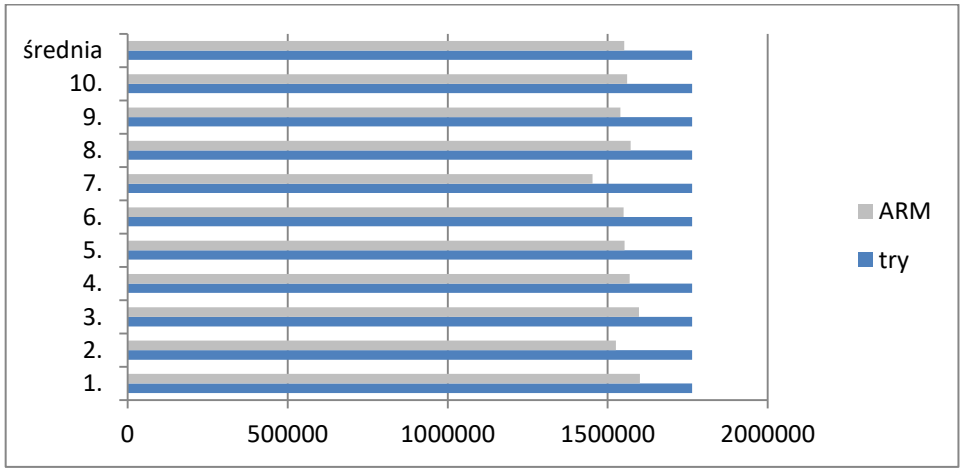
W związku z dużymi wahaniami wydajności procesora, które wpływają na wyniki pomiarów czasu wykonania programu, w tabelach podano 10 najmniejszych wartości czasu wykonania programu z populacji 100 pomiarów każdego wariantu. Wyniki pomiarów w przypadku odczytu pliku (Plik1.txt) oraz zapisu pliku (z kopiowaniem treści pliku Plik1.txt do pliku Plik2.txt) z klasycznym zamykaniem pliku (*try*) oraz z automatycznym zarządzaniem zasobami (*try-with-resources*) przedstawiono odpowiednio w tabelach 1–2.

Tabela 1. Pomiary czasu wykonywania programu odczytu pliku (w nanosekundach)

Instrukcja	Nr pomiaru										x_0
	1	2	3	4	5	6	7	8	9	10	
try	1763995	1763995	1763995	1763995	1763995	1763995	1763995	1763995	1763995	1763995	1763995
ARM	1600610	1525895	1597326	1568589	1552579	1549295	1452413	1571463	1539442	1560379	1551799

x_0 – wartość średnia.

Źródło: opracowanie własne.



Rys. 1. Porównanie czasów wykonania odczytu pliku dla 10 pomiarów [nanosekundy]

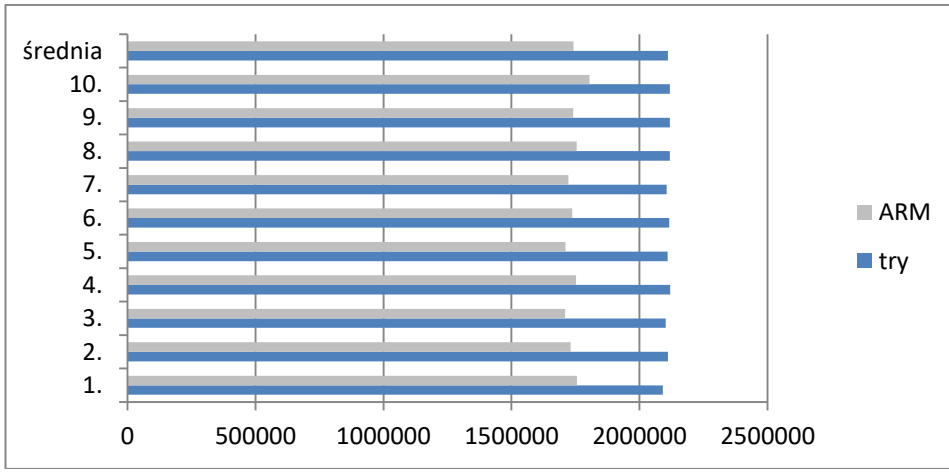
Źródło: opracowanie własne.

Tabela 2. Pomiary czasu wykonywania programu zapisu pliku [nanosekundy]

Instrukcja	Nr pomiaru										x_0
	1	2	3	4	5	6	7	8	9	10	
try	2091179	2111294	2102673	2119504	2109241	2115399	2105957	2118683	2118273	2119094	2111130
ARM	1755374	1730744	1709807	1751681	1710217	1736491	1721712	1754143	1741007	1804226	1741540

x_0 – wartość średnia.

Źródło: opracowanie własne.



Rys. 2. Porównanie czasów wykonania zapisu pliku dla 10 pomiarów [nanosekundy]
Źródło: opracowanie własne.

Podsumowanie

Odpowiednio zapisany kod programu ma duży wpływ na wydajność programu Javy. Konstrukcja kodu Javy może znacznie skrócić czas wykonywania programu Javy. Nieznaczne wahania czasu wykonania operacji dla różnych pomiarów wynikają z różnej wydajności samego procesora przy każdym pomiarze.

Przedstawione badania dowodzą, że wprowadzenie mechanizmu automatycznego zarządzania zasobami zwiększa wydajność programu (skraca kod operujący na zasobach oraz poprawia jego czytelność).

Średni czas wykonywania kodu programu wyświetlającego treść pliku z zastosowaniem mechanizmu *try-with-resources* wynosi 1551799 ns, a programu ze standardowym wyrażeniem *try* – 1763995 ns, co oznacza wzrost wydajności o 12%. Natomiast w przypadku programu do kopiowania plików średni czas wykonywania programu wynosi 1741540 ns (*try-with-resources*) oraz 2111130 ns (*try*), co przekłada się na wzrost wydajności o 18%. Wzrost wydajności jest osiągany przede wszystkim dzięki sposobowi deklaracji zasobu, który jest traktowany jako zmienna finalna.

Literatura

Downey, A. (2012). *Think Java. How to Think Like a Computer Scientist*. Pobrane z: http://thinkapjava.com (2.2017).

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. (2011). *The Java™ Language Specification. Java SE 7 Edition*. ORACLE.

Krawiec, J. (2012). *Zabezpieczanie danych*. Cz. 3. *Projektowanie systemów informatycznych*. *ITprofessional, 8.*

Oaks, S. (2014). *Java Performance – the Definite Guide*. Sebastopol: O'Reilly Media Inc.

Schildt, H. (2014). *Java The Complete Reference*. McGraw-Hill Companies, Inc.