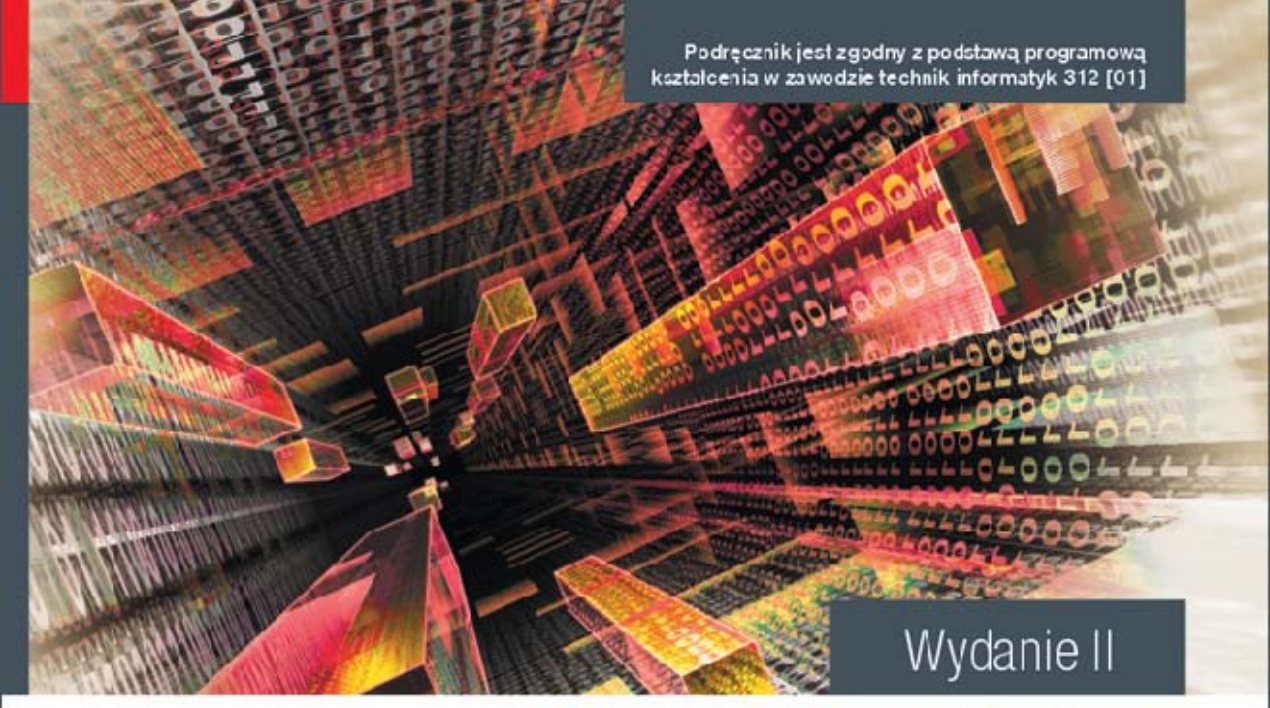


Podręcznik jest zgodny z podstawą programową kształcenia w zawodzie technik informatyk 312 [01]

Wydanie II

Podręcznik do nauki zawodu

TECHNIK INFORMATYK

PROGRAMOWANIE STRUKTURALNE I OBIEKTOWE

Zawiera CD

Helion EDUKACJA

Tomasz Rudny

Redaktor prowadzący: Marcin Borecki
Projekt okładki: Maciej Pasek

Fotografia na okładce została wykorzystana za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?prstk2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-3385-2

Printed in Poland.

Spis treści

Rozdziały, które znajdują się na płycie CD

Część V Programowanie w środowisku graficznym

38

Synchronizacja wątków

- W jaki sposób można zapewnić synchronizację wątków?
- Do czego służą metody `wait()` i `notify()`?

W poprzednim rozdziale widzieliśmy, jak nieprzewidywalne mogą być rezultaty uruchomienia programu wielowątkowego. To pokazuje wyraźnie, że w pewnych okolicznościach pożądane byłoby wymuszenie ograniczenia dostępu wątków do wspólnych danych — w tamtym przypadku do konsoli, na której wątki piszą. Java udostępnia mechanizmy synchronizacji, czyli zapewnienia dostępu do obiektu przez tylko jeden wątek na raz.

38.1. Metody synchronizowane

Metody synchronizowane to takie, które mogą być wywołane tylko przez jeden wątek na raz. Aby uczynić metodę synchronizowaną, dodajemy do jej definicji słowo kluczowe `synchronized`, np.

```
public void synchronized metoda();
```

Jeśli jeden wątek wywoła metodę synchronizowaną obiektu, żaden inny wątek nie będzie mógł wywołać tej ani żadnej innej metody synchronizowanej tego obiektu — będzie musiał poczekać, aż pierwszy wątek zakończy wykonywanie metody synchronizowanej. Bardzo ważne jest zrozumienie, że synchronizacja dotyczy konkretnego obiektu. Innymi słowy, wątek, wywołując metodę synchronizowaną, blokuje dla siebie obiekt. W niczym nie przeszkadza to innym wątkom wywoływać metody synchronizowane innych obiektów!

- wątek 1. wywołuje: `obiekt1.metoda1();`,
- wątek 2. nie może wywołać: `obiekt1.metoda1();`,
- ale może wywołać: `obiekt2.metoda1();`,

nawet jeśli `obiekt1` i `obiekt2` to obiekty tej samej klasy. (Zakładamy oczywiście, że `metoda1()` to metoda synchronizowana).

38.2. Synchronizowane bloki kodu

Inną metodą synchronizacji wątków w Javie jest zastosowanie synchronizowanych bloków kodu. Dowolny fragment kodu może być objęty klamrami `synchronized { }` i dzięki temu dostęp do niego zostanie ograniczony tylko do jednego wątku naraz. Metoda synchronizowana w naturalny sposób wiąże się z obiektem, na którym jest wywoływana. Synchronizowane bloki kodu muszą jawnie wskazywać, jakiego obiektu dotyczą:

```
synchronized (Obiekt, na którym synchronizujemy) {
  /* Kod, który może wykonać tylko jeden wątek na raz */
}
```

Ta początkowo dość dziwna konstrukcja staje się bardziej zrozumiała, gdy uświadomimy sobie, że Java blokuje wątkom dostęp do konkretnego obiektu. Oznacza to, że dowolny obiekt może być swoistą blokadą synchronizującą dla wątków. Dlatego właśnie w sekcji `synchronized` konieczne jest podanie obiektu, na którym chcemy synchronizować.

UWAGA

W programowaniu współbieżnym mówimy o **monitorach** (lub *semaforach*), czyli mechanizmach gwarantujących wyłączny dostęp dla jednego wątku. W Javie rolę monitora pełni obiekt, na którym synchronizujemy.

Jaki obiekt wybrać jako barierę synchronizacji? Zazwyczaj najlepiej, gdy jest to obiekt będący wspólnym zasobem, do którego dostęp uzyskują wątki. W przykładzie programu, w którym wątki piszą na konsoli, może to być jakikolwiek obiekt wykorzystywany do pisania.

W stosunku do przykładu 37.2 wprowadzono pewne różnice — wątki nie piszą bezpośrednio na ekranie, ale poprzez wspólny obiekt o nazwie `konsola`. Ponadto do tego obiektu przeniesiono algorytm pisania po znaku i usypiania na pewien losowy czas.

Warto jednak sprawdzić, że to nie przeniesienie kodu piszącego do nowego obiektu, ale dodanie synchronizacji wprowadza porządek — tekst wypisywany jest całymi słowami (choć słowa mogą się pojawiać w przypadkowej kolejności).

Przykład 38.1
Plik Watek.java

```
package podrecznik.synchronizacja;

public class Watek implements Runnable {
   private String slowo;
   private Thread watek;
   private int znak = 0;
```

```
   private Konsola kon;

   public Watek(String slowo, String id, Konsola kon) {
      this.slowo = new String(slowo);
      this.kon = kon;
      watek = new Thread(this, id);
      watek.start();
   }

   public void run() {
      kon.pisz(slowo);
   }
}
```

Plik Konsola.java

```
package podrecznik.synchronizacja;

public class Konsola {
   public synchronized void pisz(String s) {
      short znak = 0;
      while (znak < s.length()) {
         System.out.print(s.charAt(znak++));
         try {
            Thread.sleep((int) (Math.random() * 200));
         } catch (InterruptedException e) { }
      }
   }
}
```

Plik KlasaGlowna.java

```
package podrecznik.synchronizacja;

public class KlasaGlowna {

   public static void main(String[] args) {
      String zdanie[] = { "Na ", "ten ", "czas ", "Wojski ", "chwycił ",
"róg ", "długi, ", "cętkowany, ", "kręty " };
      Konsola kon = new Konsola();

      for (int i = 0; i < 9; i++) {
         new Watek(zdanie[i], "" + i, kon);
      }
   }

}
```

38.3. Komunikacja pomiędzy wątkami

Rozważmy przykład: dwa wątki operują na stosie. Jeden umieszcza elementy na stosie, drugi je ze stosu zdejmuje[36]. Oczywiście, nie można pozwolić, by oba wątki uzyskiwały dostęp do stosu jednocześnie, dlatego metody `push()` i `pop()` muszą być synchronizowane. Co jednak wtedy, gdy drugi wątek chce pobrać element ze stosu, ale na nim nie ma elementów? Musi poczekać, aż pierwszy wątek (producent) doda element na stos. Tu pojawia się problem — aktualnie to wątek konsumenta ma dostęp do obiektu stosu, zatem wątek producenta nie może niczego na nim umieścić!

Jak rozwiązać ten problem? Byłoby dobrze, gdyby wątek mógł „usnąć" i zwolnić synchronizowany obiekt (monitor) dla innego wątku, a potem „obudzić się", gdy jest już możliwe wykonanie jego zadania. I rzeczywiście, w Javie jest taki mechanizm — są to metody `wait()` i `notify()`. Działa on następująco:

1. Wątek wykonujący kod (lub metodę) synchronizowany, który czeka na jakieś zdarzenie, wywołuje metodę `wait()`. W ten sposób zwalnia dostęp do obiektu.
2. Gdy inny wątek kończy wykonywanie kodu synchronizowanego, wywołuje metodę `notify()`, aby powiadomić (obudzić) inne wątki, że zakończył działanie, co być może oznacza dla tych wątków możliwość wykonania ich zadania.

Ten schemat został zilustrowany w przykładzie poniżej. Wątki `producent` i `konsument` w losowych odstępach czasu dodają i usuwają elementy ze stosu. Stos jest zaimplementowany jako zwykła tablica. Zmienna typu całkowitego wskazuje aktualną liczbę elementów na stosie. Dla ułatwienia dodano dwie metody, `pelny()` i `pusty()`, informujące o aktualnym stanie stosu. Jeśli stos jest pusty, wątek konsumenta „usypia", wywołując metodę `wait()`. Podobnie zachowuje się wątek producenta, jeśli stos jest pełny. Każdy z wątków informuje drugi o wyjściu z sekcji synchronizowanej, wywołując metodę `notify()`.

Przykład 38.2
Plik Stos.java

```
package podrecznik.synchronizacja;

public class Stos {
   private Integer tablica[];
   private volatile int pozycja;
   private final int maxStos = 10;

   public Stos() {
      tablica = new Integer[maxStos];
      pozycja = -1;
   }
```

[36] Takie zadania noszą nazwę problemów producentów i konsumentów.

```
   public synchronized Integer pop() {
      if (pusty()) {
         try {
            System.out.println("Konsument czeka na elementy stosu.");
            wait();
         } catch (InterruptedException e) { } // Nic nie rób po wybudzeniu
      }
      Integer element = tablica[pozycja];
      tablica[pozycja] = null;
      pozycja--;
      System.out.println("Zdjęto element. Aktualna pozycja stosu: " +
pozycja);
      notify(); // Powiadom wątek producenta
      return element;
   }

   public synchronized void push(Integer i) {
      if (pelny()) {
         try {
            System.out.println("Producent czeka, aż się zwolni miejsce na
stosie.");
            wait();
         } catch (InterruptedException e) { }
      }
      tablica[++pozycja] = i;
      System.out.println("Dodano element. Aktualna pozycja stosu: " +
pozycja);
      notify();
   }

   private boolean pusty() {
      if (pozycja < 0)
         return true;
      return false;
   }

   private boolean pelny() {
      if (pozycja == maxStos - 1)
         return true;
      return false;
   }
}
```

Plik Producent.java

```
package podrecznik.synchronizacja;

public class Producent extends Thread {
   private Stos stos;

   public Producent(String id, Stos stos) {
      super(id);
      this.stos = stos;
      start();
   }

   public void run() {
      while (true) {
         System.out.println(getName() + ": dodaje na stos...");
         stos.push(new Integer(1));
         try {
            sleep((int)(Math.random()*200));
         } catch (InterruptedException e) {    }
      }
   }
}
```

Plik Konsument.java

```
package podrecznik.synchronizacja;

public class Konsument extends Thread {
   private Stos stos;

   public Konsument(String id, Stos stos) {
      super(id);
      this.stos = stos;
      start();
   }

   public void run() {
      while (true) {
         System.out.println(getName() + ": zdejmuję ze stosu...");
         stos.pop();
         try {
            sleep((int)(Math.random()*200));
         } catch (InterruptedException e) {    }
      }
   }
}
```

Plik Test.java

```
package podrecznik.synchronizacja;

public class Test {

   public static void main(String[] args) throws InterruptedException {
      Stos stos = new Stos();
      new Producent("producent", stos);
      new Konsument("konsument", stos);
      System.out.println("Koniec pracy wątku głównego");
   }

}
```

W naszym przykładzie wątki będą pracować w pętli nieskończonej (`while (true)`). Aby zatrzymać działanie programu, należy zakończyć proces w systemie. W środowisku Eclipse można to uczynić, klikając czerwony kwadracik w prawym dolnym rogu ekranu. Jako ćwiczenie pozostawiamy zmianę pętli, tak by np. zatrzymywała się po 10 sekundach (sprawdzanie czasu systemowego `System.currentTimeMillis()`).

PYTANIA KONTROLNE

1. Jakie są dwie metody synchronizacji wątków w Javie?
2. Wątek `A` wywołuje synchronizowaną metodę `m1()` obiektu `Ob1`. Czy wątek `B` może wywołać w tym czasie metodę synchronizowaną `m2()` obiektu `Ob1`? A obiektu `Ob2` tej samej klasy?
3. Do czego służą metody `wait()` i `notify()`?

ĆWICZENIA

1. Zmodyfikuj program z przykładu 38.2 tak, aby jednocześnie działało wielu konsumentów.
2. Zmodyfikuj program z przykładu 38.2 tak, aby jednocześnie działało wielu producentów.

39 Komunikacja sieciowa

- Jak nawiązać połączenie sieciowe?
- W jaki sposób można czytać i pisać przez sieć?
- Jaką strukturę ma schemat budowy aplikacji serwera i klienta?

Wraz z upowszechnieniem się internetu coraz więcej programów przesyła informacje w sieci. Komunikatory przesyłają i odbierają wiadomości pisane przez rozmawiające ze sobą osoby, przeglądarki internetowe pobierają treść stron WWW, programy użytkowe automatycznie aktualizują swoje dane itd. W jaki sposób nawiązać połączenie pomiędzy dwoma programami i przesyłać dane? Jak dowiemy się z tego rozdziału, należy skorzystać ze specjalnego obiektu, tzw. gniazda (ang. *socket*)[37].

39.1. Podstawy programowania sieciowego

Programy korzystające z sieci używają gniazd (ang. *socket*). Gniazdo to zakończenie dwukierunkowego łącza, podobnie jak gniazdko w ścianie jest zakończeniem np. linii telefonicznej. Za pomocą gniazda programy określają, z kim chcą się komunikować.

DEFINICJA

Gniazdo — zakończenie abstrakcyjnego łącza komunikacji sieciowej pomiędzy dwoma programami. Gniazdo jest opisane przez adres IP oraz numer portu.

Aby określić adresata komunikacji w sieci, konieczne jest podanie *adresu IP oraz numeru portu*. Adres IP składa się z czterech liczb z przedziału 0 – 255 rozdzielonych kropkami,

37 Oczywiście programy sieciowe można pisać w dowolnym języku. W podręczniku temat ten omówiono na przykładzie Javy.

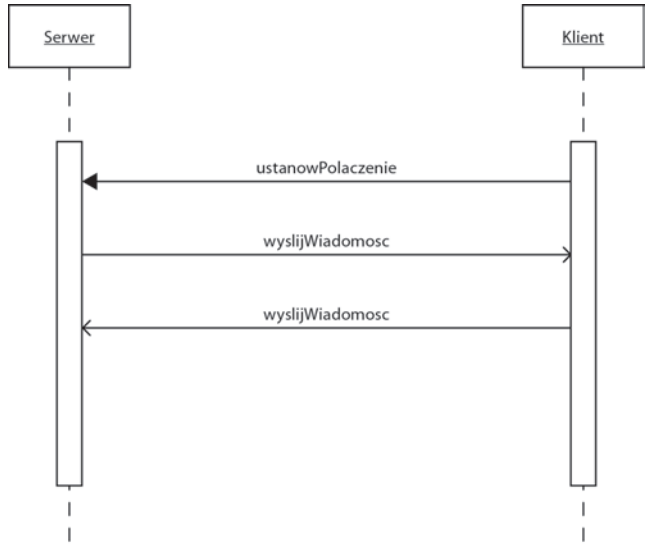
np. 194.178.29.1. Każdy komputer, nawet niepodłączony do internetu, ma swój lokalny adres 127.0.0.1. Dzięki temu możemy testować programy sieciowe, nie mając dostępu do sieci komputerowej. Z kolei numer portu pozwala na korzystanie z sieci przez wiele programów jednocześnie — każdy program korzysta z innego portu i dzięki temu system operacyjny potrafi dostarczyć właściwym programom właściwe paczki danych, mimo że do wszystkich odnosi się ten sam adres IP. To rozwiązanie przypomina trochę strukturę bloku mieszkalnego — ten sam numer ulicy, ale różne numery mieszkań.

UWAGA

Jeśli kod programu wygląda poprawnie, a mimo to program sieciowy nie chce działać, warto spróbować zmienić używany numer portu. Być może jakiś inny program już korzysta z tego portu, uniemożliwiając tym samym jego wykorzystanie przez nasz proces. Przyczyną może być również włączona zapora sieciowa.

Z punktu widzenia komunikacji sieciowej wyróżniamy dwa rodzaje programów — *serwery* i *klientów*. *Serwer* zazwyczaj udostępnia gniazdo do komunikacji i czeka, nasłuchując, czy ktoś chce się z nim połączyć. Jeśli jakiś inny program ustanowi połączenie z serwerem, ten przejdzie w tryb obsługi tego połączenia. Z kolei *klient* to właśnie ten program, który łączy się z serwerem, a następnie komunikuje się z nim („rozmawia"). Zasadnicza różnica polega na tym, że serwer nie wie, z kim będzie się komunikował — udostępnia gniazdo i czeka, z kolei klient musi znać adres i port serwera, z którym chce się połączyć.

Mechanizm ten przedstawiono symbolicznie na rysunku 39.1. Oś pionowa wyobraża czas, jej zwrot jest skierowany w dół. Wąskie, pionowe prostokąty pod obiektami **Serwer** i **Klient** wyobrażają wykonywanie ich kodu. Poziome strzałki to przesyłanie komunikatów. Najpierw klient zgłasza się do serwera, prosząc o ustanowienie połączenia. W odpowiedzi serwer konfiguruje połączenie i rozpoczyna się (zazwyczaj) naprzemienne przesyłanie wiadomości.



Rysunek 39.1. Schemat komunikacji serwer-klient

39.2. Prosty serwer

Jak zbudować prostą aplikację serwerową? Schemat postępowania jest zawsze taki sam:

1. Utwórz gniazdo serwera.
2. Czekaj na połączenia od klientów.
3. Gdy klient się podłączy, utwórz nowe gniazdo do komunikacji z klientem.
4. Otwórz strumienie do pisania i czytania z gniazda klienckiego.
5. Czytaj i pisz za pośrednictwem gniazda klienckiego.
6. Zamknij strumienie i gniazda.

Utworzenie gniazda serwerowego (do nasłuchiwania i czekania na klientów) jest wykonywane za pomocą konstruktora klasy `ServerSocket`:

```
gniazdoSerwera = new ServerSocket(port);
```

gdzie zmienna `port` typu `int` przechowuje numer portu, np. 5555. Jak widać, serwer nie potrzebuje znać swojego adresu IP. Następnie wywoływana jest metoda `accept` obiektu gniazda, która zawiesza swoje działanie do czasu zgłoszenia się klienta.

```
gniazdoObslugiKlienta = gniazdoSerwera.accept();
```

Metoda ta jest zwykle wywoływana bezpośrednio po utworzeniu gniazda, ale powrót z niej następuje dopiero po nawiązaniu połączenia. Zwracaną wartością jest referencja do nowego obiektu typu `Socket`. To nowe gniazdo służy do komunikacji z nowo podłączonym klientem. Dlaczego serwer otwiera nowe gniazdo? Chodzi o to, by na tym pierwszym w dalszym ciągu oczekiwać (nasłuchiwać) na kolejnych klientów.

UWAGA

W profesjonalnych programach zwykle tworzy się osobne wątki do obsługi każdego klienta, tak aby wątek główny programu mógł stale nasłuchiwać na nowych klientów. W omówionym tutaj prostym schemacie kolejni dołączający klienci będą czekać na gnieździe, aż wątek główny serwera skończy obsługiwać pierwszego klienta.

Kolejny, czwarty krok naszego schematu to otwarcie strumieni do pisania i czytania. Gniazdo jest interfejsem umożliwiającym przesyłanie danych przez sieć, ale — podobnie jak plik czy konsola — wymaga dostępu za pośrednictwem strumieni. Standardowo strumienie te otwiera się następująco:

```
BufferedReader in = new BufferedReader(new InputStreamReader(gniazdo.
getInputStream()));
PrintWriter out = new PrintWriter(gniazdo.getOutputStream(), true);
```

gdzie `gniazdo` oznacza gniazdo wykorzystywane do obsługi klienta. Mając do dyspozycji strumienie `in` i `out`, możemy z nich korzystać tak jak z innych strumieni i nie martwić się już o to, że odnoszą się do gniazda:

```
out.println("Witaj, kliencie!");
in.readLine();
```

Na końcu pracy programu konieczne jest jeszcze tylko zamknięcie strumieni i gniazd:

```
in.close();
out.close();
gniazdo.close();
```

Poniższy przykład zawiera pełny kod programu serwera. Warto zwrócić uwagę na kolejne kroki nawiązywania połączenia i przesyłania komunikatów. Serwer wysyła wiadomości do klienta, dopóki ten odpowiada (pętla `while(line != null)`). Wszelkie błędy zostały obsłużone przez mechanizm wyjątków.

Przykład 39.1

```
package podrecznik;

import java.io.*;
import java.net.*;

public class Serwer {
   private ServerSocket gniazdoSerwera = null;
   private Socket gniazdoObslugiKlienta = null;
   private BufferedReader in = null;
   private PrintWriter out = null;
   private String line = "";
   private int port = 5555;

   public Serwer() {
      sluchaj();
      ustanowPolaczenie();
      rozmawiaj();
      // Przyjmij jeszcze trzech klientów
      for (int i = 0; i < 3; i++) {
         ustanowPolaczenie();
         rozmawiaj();
      }
   }

   public void ustanowPolaczenie() {
      try {
         gniazdoObslugiKlienta = gniazdoSerwera.accept();
         System.out.println("[Serwer] Zaakceptowano połączenie.");
      } catch (IOException e) {
         System.err.println("[Serwer] Metoda accept zawiodła.");
         System.exit(-1);
      }

      try {
         in = new BufferedReader(new InputStreamReader(gniazdoObslugiKlie
nta.getInputStream()));
```

```
         out = new PrintWriter(gniazdoObslugiKlienta.getOutputStream(),
true);
      } catch (IOException e) {
         System.err.println("[Serwer] Nie można utworzyć strumieni.");
         System.exit(-1);
      }
   }

   public void rozmawiaj() {
      while (line != null) {
         try {
            line = in.readLine();
            if (line != null) {
               System.out.println("[Serwer] Otrzymano wiadomość: " +
line);
               try {
                  Thread.sleep(1000);
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
               out.println("dziękuję, kliencie.");
            }
         } catch (IOException e) {
            System.err.println("[Serwer] Błąd odczytu z gniazda -
prawdopodobnie klient rozłączył się.");
            break;
         }
      }
      line = "";
   }

   public void sluchaj() {
      try {
         System.out.println("[Serwer] Serwer rozpoczyna nasłuchiwanie na
porcie " + port);
         gniazdoSerwera = new ServerSocket(port);
      } catch (IOException e) {
         System.err.println("[Serwer] Nie można nasłuchiwać na porcie: "
+ port);
         System.exit(-1);
      }
   }

   public void zwolnijZasoby() {
      try {
```

```
            in.close();
            out.close();
            gniazdoSerwera.close();
         } catch (IOException e) {
            System.err.println("[Serwer] Błąd metody close.");
            System.exit(-1);
         }
      }

      public static void main(String[] args) {
         Serwer serwer = new Serwer();
         serwer.zwolnijZasoby();
      }
   }
```

39.3. Prosty klient

Po poznaniu schematu działania serwera łatwo będzie nam zrozumieć działanie aplikacji klienckiej. Jej działanie można opisać następującymi krokami:

1. Utwórz gniazdo serwera.
2. Otwórz strumienie do pisania i czytania z gniazda.
3. Czytaj i pisz za pośrednictwem gniazda.
4. Zamknij strumienie i gniazdo.

Nawiązanie połączenia polega na utworzeniu obiektu gniazda. Tym razem podajemy również adres IP serwera, do którego chcemy się podłączyć:

```
gniazdo = new Socket(ip, port);
```

Jeśli wszystko pójdzie zgodnie z planem, możemy — podobnie jak w przypadku serwera — otworzyć strumienie do czytania i pisania, a następnie rozpocząć komunikację. Reszta kodu klienta niewiele już różni się od kodu serwera. Cały kod przedstawiono w przykładzie poniżej.

Przykład 39.2

```
package podrecznik;

import java.io.*;
import java.net.*;

public class Klient {
   private Socket gniazdo = null;
   private PrintWriter out = null;
   private BufferedReader in = null;
   private int port = 5555;
   private String ip = "127.0.0.1";
```

```
    public Klient() {
        ustanowPolaczenie();
        wyslij("Witaj, serwerze!");
        try {
            String line = in.readLine();
            System.out.println("[Klient] Otrzymano wiadomość: " + line);
            out.println("Dziękuję, serwerze!");
        } catch (IOException e) {
            System.out.println("Read failed");
            System.exit(1);
        }
    }

    public void zwolnijZasoby() {
        try {
            in.close();
            out.close();
            gniazdo.close();
        } catch (IOException e) {
            System.err.println("[Klient] Błąd metody close.");
        }
    }

    private void wyslij(String s) {
        out.println(s);
    }

    public void ustanowPolaczenie() {
        try {
            gniazdo = new Socket(ip, port);
            out = new PrintWriter(gniazdo.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(gniazdo.
getInputStream()));
            System.out.println("[Klient] Podłączono do serwera " + ip + ": "
+ port);
        } catch (UnknownHostException e) {
            System.err.println("[Klient] Nie można połączyć z: " + ip);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("[Klient] Błąd wejścia-wyjścia");
            System.exit(1);
        }
    }
```

```
    public static void main(String[] args) {
        Klient klient = new Klient();
        klient.zwolnijZasoby();
    }
}
```

PYTANIA KONTROLNE

1. Co to jest gniazdo i do czego służy?
2. Wymień kolejne kroki w typowym schemacie działania serwera.
3. Wyjaśnij, czym różni się działanie serwera od działania klienta.
4. Do czego wykorzystuje się strumienie w programach sieciowych?

ĆWICZENIA

1. Jeśli masz dostęp do sieci, uruchom program serwera i klienta na dwóch różnych komputerach. Wskazówka: konieczne będzie wpisanie w kodzie poprawnego adresu IP.
2. Zmodyfikuj program klienta tak, aby przesyłał do serwera napisy wprowadzane przez użytkownika na konsoli.

40

Przykład podsumowujący: gra sieciowa

- Przykład podsumowujący poznane elementy programowania w Javie, w szczególności komunikację sieciową i graficzny interfejs użytkownika.

W celu utrwalenia poznanych technik programowania w języku Java przedstawimy program umożliwiający grę w kółko i krzyżyk w sieci. Napiszemy dwa programy — serwera i klienta. Ich rola w komunikacji sieciowej będzie inna, ale gdy połączenie zostanie nawiązane, oba będą w identyczny sposób umożliwiać grę.

40.1. Omówienie programu

Struktura obu programów — serwera i klienta — jest oparta na klasie bazowej `JFrame`. W konstruktorze tworzymy okienko aplikacji i dodajemy do niego dziewięć przycisków, które będą reprezentować pola planszy. Każdemu z nich przypisujemy jako słuchacza zdarzeń klasę okna aplikacji. Dodatkowo, aby móc rozróżniać zdarzenia pochodzące od przycisków, korzystamy z metody `setActionCommand(String)`, która dodaje zmienną napisową do każdego komunikatu (`Event`).

```
przyciski = new JButton[9];
for (int i = 0; i < 9; i++) {
   przyciski[i] = new JButton("");
   przyciski[i].setActionCommand("" + i); //Dodanie napisu (numer) do komunikatów
   przyciski[i].addActionListener(this);
   getContentPane().add(przyciski[i]);
}
```

Napis ten można wydobyć z klasy komunikatu o zdarzeniu za pomocą metody `getActionCommand` klasy `Event`. W ten sposób identyfikujemy przycisk, który został naciśnięty. Tę czynność wykonujemy w kodzie metody `actionPerformed`.

```
int j = Integer.parseInt(arg0.getActionCommand());
```

W procedurze obsługi zdarzeń sprawdzamy, czy aktualnie jest nasz ruch (służy do tego zmienna logiczna `mojRuch`). Jeśli nie, ignorujemy kliknięcia przycisków. Jeśli natomiast jest nasz ruch, zmieniamy tekst odpowiedniego przycisku na *O* oraz przesyłamy współrzędną pola (`ruch`) do klienta.

Komunikacja sieciowa jest zaimplementowana zgodnie z poznanym wcześniej schematem: najpierw metoda `sluchaj()` tworzy gniazdo serwera, następnie czeka na podłączenie klienta, wreszcie otwiera strumienie (buforowane) do czytania i pisania. Potem wywoływana jest metoda `odbieraj()`. Metoda ta w pętli próbuje czytać linię tekstu ze strumienia powiązanego z gniazdem. Gdy taka się pojawi, jest interpretowana i plansza zostaje odświeżona — pojawia się krzyżyk w odpowiednim miejscu. Zaktualizowana zostaje też zmienna `mojRuch`, ponieważ nadejście informacji o ruchu wykonanym przez przeciwnika jest równoznaczne z tym, że teraz nasza kolej.

```
while ((line = in.readLine()) != null) {
   int j = Integer.parseInt(line);
   System.out.println("[Serwer] <== " + j);
   if (j >= 0 && j < 9) {
      przyciski[j].setText("X");
   }
   mojRuch = true;
}
```

Oczywiście wszystkie wymagające tego fragmenty kodu są otoczone blokami `try-catch` w celu obsługi błędów. Dodatkowo dla lepszego zrozumienia w wielu miejscach program wypisuje na konsoli informacje pomocnicze.

W metodzie `odbieraj()` dziwić może pętla `while`. Jak to możliwe, że program, wchodząc w tę pętlę, dalej reaguje na kliknięcia myszą? Nie zaimplementowano przecież jawnie wielowątkowości. A jednak program działa. Aby lepiej zrozumieć tę kwestię, musimy przypomnieć sobie sposób działania strumieni buforowanych — wywołanie metody `readLine()` jest *blokowane* (zawieszane) do czasu, aż strumień będzie gotowy (pojawi się ciąg znaków zakończony `\n`). Przez ten czas program może obsługiwać komunikaty zdarzeń, np. kliknięcia przycisków. I tak jest w naszym programie.

Program klienta jest bardzo podobny do serwera. Jedyna różnica polega na sposobie nawiązywania połączenia.

40.2. Kod programu

Przykład 40.1
Plik Serwer.java

```
package podrecznik;

import java.awt.GridLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class Serwer extends JFrame implements ActionListener {
   protected JButton przyciski[];
   protected boolean krzyzyk = true;

   private ServerSocket gniazdoSerwera = null;
   private Socket gniazdoObslugiKlienta = null;
   private BufferedReader in = null;
   private PrintWriter out = null;
   private String line = "";
   private int port = 4567;

   // Serwer zaczyna
   private boolean mojRuch = true;

   public Serwer() {
      super("OXO Serwer");
      setLayout(new GridLayout(3, 3));
      przyciski = new JButton[9];
      for (int i = 0; i < 9; i++) {
         przyciski[i] = new JButton("");
         przyciski[i].setActionCommand("" + i);
         przyciski[i].addActionListener(this);
         getContentPane().add(przyciski[i]);
      }
      setSize(300, 300);
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent we) {
            System.exit(0);
            try {
               in.reset();
            } catch (IOException e) {
               e.printStackTrace();
```

```
            }

            zwolnijZasoby();
         }
      });

      setVisible(true);
      sluchaj();
      odbieraj();
   }

   public void actionPerformed(ActionEvent arg0) {
      if (!mojRuch)
         return;
      int j = Integer.parseInt(arg0.getActionCommand());
      if (j >= 0) {
         if ("".equals(przyciski[j].getText())) {
            przyciski[j].setText("O");
            // Wyślij informację o wykonanym ruchu do klienta
            System.out.println("[Serwer] ==> " + j);

            repaint();
            out.println(j);
         }
      }
      mojRuch = false;
      this.setTitle("Oczekiwanie na ruch przeciwnika");
   }

   public void odbieraj() {
      try {
         while ((line = in.readLine()) != null) {
            int j = Integer.parseInt(line);
            System.out.println("[Serwer] <== " + j);
            if (j >= 0 && j < 9) {
               przyciski[j].setText("X");
            }
            mojRuch = true;
            this.setTitle("Wykonaj ruch");
         }
      } catch (IOException e1) {
         e1.printStackTrace();
      }
   }
```

```
   public void sluchaj() {
      try {
         System.out.println("[Serwer] Serwer rozpoczyna nasłuchiwanie na
porcie "+ port);
         gniazdoSerwera = new ServerSocket(port);
      } catch (IOException e) {
         System.err.println("[Serwer] Nie można nasłuchiwać na porcie: "
               + port);
         System.exit(-1);
      }

      try {
         gniazdoObslugiKlienta = gniazdoSerwera.accept();
         System.out.println("[Serwer] Zaakceptowano połączenie.");
      } catch (IOException e) {
         System.err.println("[Serwer] Metoda accept zawiodła.");
         System.exit(-1);
      }

      try {
         in = new BufferedReader(new InputStreamReader(
               gniazdoObslugiKlienta.getInputStream()));
         out = new PrintWriter(gniazdoObslugiKlienta.getOutputStream(),
true);
      } catch (IOException e) {
         System.err.println("[Serwer] Nie można utworzyć strumieni.");
         System.exit(-1);
      }
   }

   public void zwolnijZasoby() {
      try {
         in.close();
         out.close();
         gniazdoSerwera.close();
         gniazdoObslugiKlienta.close();
      } catch (IOException e) {
         System.err.println("[Serwer] Błąd metody close.");
         System.exit(-1);
      }
   }

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

Plik Klient.java

```
package podrecznik;

import java.awt.GridLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Klient extends JFrame implements ActionListener {
   private Socket gniazdo = null;
   private PrintWriter out = null;
   private BufferedReader in = null;
   private int port = 4567;
   private String ip = "127.0.0.1";
   private String line = "";

   // Zaczyna serwer
   private boolean mojRuch = false;

   private JButton przyciski[];

   public Klient() {
      super("OXO Klient");
      setLayout(new GridLayout(3, 3));
      przyciski = new JButton[9];
      for (int i = 0; i < 9; i++) {
         przyciski[i] = new JButton("");
         przyciski[i].setActionCommand("" + i);
         przyciski[i].addActionListener(this);
         getContentPane().add(przyciski[i]);
      }
      setSize(300, 300);
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent we) {
            System.exit(0);
            try {
               in.reset();
            } catch (IOException e) {
               e.printStackTrace();
            }

            zwolnijZasoby();
         }
```

```
        });

        setVisible(true);

        ustanowPolaczenie();
        odbieraj();
    }

    public void actionPerformed(ActionEvent arg0) {
        if (!mojRuch)
            return;
        int j = Integer.parseInt(arg0.getActionCommand());
        if (j >= 0) {
            if (przyciski[j].getText().equals("")) {
                przyciski[j].setText("X");
                // Wyślij informację o wykonanym ruchu do serwera
                System.out.println("[Klient] ==> " + j);

                repaint();
                out.println(j);
            }
        }
        mojRuch = false;
        this.setTitle("Oczekiwanie na ruch przeciwnika");
    }

    public void odbieraj() {
        try {
            while ((line = in.readLine()) != null) {
                int j = Integer.parseInt(line);
                System.out.println("[Klient] <== " + j);
                if (j >= 0 && j < 9) {
                    przyciski[j].setText("O");
                }
                mojRuch = true;
                this.setTitle("Wykonaj ruch");
            }
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }

    public void zwolnijZasoby() {
        try {
```

```
            in.close();
            out.close();
            gniazdo.close();
         } catch (IOException e) {
            System.err.println("[Klient] Błąd metody close.");
         }
      }

      public void ustanowPolaczenie() {
         try {
            gniazdo = new Socket(ip, port);
            out = new PrintWriter(gniazdo.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                  gniazdo.getInputStream()));
            System.out.println("[Klient] Podłączono do serwera " + ip + ": "
+ port);

         } catch (UnknownHostException e) {
            System.err.println("[Klient] Nie można połączyć z: " + ip);
            System.exit(1);
         } catch (IOException e) {
            System.err.println("[Klient] Błąd wejścia-wyjścia");
            System.exit(1);
         }
      }

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

ĆWICZENIA

1. Napisz funkcję przeglądającą planszę w poszukiwaniu linii złożonej z krzyżyków lub kółek. Funkcja ta powinna przekazać informację o zakończeniu gry i o zwycięzcy (np. wyświetlić na konsoli).
2. Dodaj możliwość rozmów pomiędzy graczami.
3. Napisz algorytm logiki gry w kółko i krzyżyk, tak aby móc grać z przeciwnikiem komputerowym. Wybór najlepszego ruchu powinien uwzględniać następujące czynniki:
   a) Czy przeciwnik może wygrać w następnym ruchu? Jeśli tak, zablokuj go.
   b) Czy ja mogę wygrać w następnym ruchu? Jeśli tak, wykonaj taki ruch.

Skorowidz

Podręcznik jest zgodny z podstawą programową kształcenia w zawodzie technik informatyk 312 [01]

Podręcznik do nauki zawodu

TECHNIK INFORMATYK

Wydanie II

PROGRAMOWANIE STRUKTURALNE I OBIEKTOWE

Technik informatyk nie jest zwykłym użytkownikiem komputerów. Jeśli uczeń wybiera szkołę o takim profilu, z czasem staje się prawdziwym komputerowym ekspertem.

Programowanie komputerów osobistych stanowi esencję informatyki. Jest twórcze i wymiernie wpływa na działanie sprzętu, na którym pracujemy — w tym na jego użyteczność oraz efektywność. Bez odpowiedniego oprogramowania niemożliwe jest funkcjonowanie stron internetowych, programów biurowych i bankowych czy nawet sprzętu AGD. Umiejętność programowania w dzisiejszym świecie jest wręcz nieodzowna w zawodzie inżyniera, technika, webmastera czy naukowca.

Programowanie strukturalne i obiektowe. Podręcznik do nauki zawodu technik informatyk w przystępny sposób wyjaśnia wszystkie niezbędne terminy, tak aby nawet osoba, która nigdy nie miała styczności z programowaniem, mogła je bez przeszkód zrozumieć. Podręcznik został podzielony na części stanowiące trzy odrębne kursy programowania: w językach Pascal, C++ oraz Java. W czasie realizacji materiału uczniowie poznają najważniejsze elementy tych języków. Każdą część zamyka rozbudowany i ciekawy przykład podsumowujący.

Do książki została dołączona płyta CD zawierająca ważne i interesujące informacje dodatkowe, które ze względów praktycznych nie mogły ukazać się w wersji drukowanej. Znajduje się tam kurs wprowadzający do programowania w Windows za pomocą WinAPI, omówienie zagadnień pracy grupowej, projektowania oprogramowania i optymalizacji kodu, a także wybrane tematy uzupełniające, m.in. wiadomości dotyczące reprezentacji pozycyjnej liczb.

http://edukacja.helion.pl

helion.pl księgarnia internetowa

Nr katalogowy: 6861

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

Helion EDUKACJA

Sprawdź najnowsze promocje: http://helion.pl/promocje
Książki najchętniej czytane: http://helion.pl/bestsellery
Zamów informacje o nowościach: http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-246-3385-2

9 788324 633852

Informatyka w najlepszym wydaniu