

Cele

W tym rozdziale:

- Implementacja aplikacji sieciowych wykorzystujących gniazda sieciowe i datagramy
- Implementacja klientów i serwerów Javy komunikujących się między sobą poprzez sieć
- Implementacja wykorzystującej sieć aplikacji do wspólnej pracy

- | | |
|---|--|
| 26.1. Wprowadzenie | 26.6. Datagramy — bezpołączeniowa interakcja między klientem i serwerem |
| 26.2. Odczyt pliku z serwera WWW | 26.7. Kółko i krzyżyk w wersji klient – serwer z serwerem wielowątkowym |
| 26.3. Wykonanie prostego serwera przy użyciu gniazd strumieniowych | 26.8. Opcjonalne studium przypadku — aplikacja DeitelMessenger |
| 26.4. Wykonanie prostego klienta przy użyciu gniazd strumieniowych | 26.9. Podsumowanie |
| 26.5. Interakcja klienta i serwera wykorzystująca gniazda strumieniowe | |

Streszczenie | Ćwiczenia do samooceny | Odpowiedzi do samooceny | Ćwiczenia

26.1. Wprowadzenie

Java zapewnia wiele różnych rozwiązań dotyczących obsługi sieci, co pozwala tworzyć aplikacje internetowe lub obsługujące strony WWW. Java umożliwia tworzenie programów przeszukujących światową pajęczynę lub współpracujących z programami działającymi na innych komputerach w dowolnym zakątku świata lub tylko w sieci firmowej (ze względów bezpieczeństwa).

Podstawowe możliwości sieciowe Javy są zadeklarowane w klasach i interfejsach pakietu **java.net**. Dzięki nim Java oferuje **komunikację opartą na strumieniach**, która pozwala aplikacjom traktować połączenie sieciowe jako strumień danych. Klasy i interfejsy pakietu **java.net** oferują również komunikację pakietową związaną z transmisją pojedynczych pakietów informacji — najczęściej wykorzystuje się ją do przesyłania obrazów, dźwięku i wideo poprzez internet. W tym rozdziale pokażemy, jak realizować komunikację zarówno w wersji pakietowej, jak i strumieniowej.

Skupimy się na obu stronach związku **klient – serwer**. **Klient** żąda przeprowadzenia pewnej akcji, a **serwer** wykonuje tę akcję i **odpowiada** klientowi. Najbardziej rozpowszechnioną implementacją **modelu żądanie – odpowiedź** jest komunikacja między serwerami WWW i przeglądarkami internetowymi. Gdy użytkownik wybierze witrynę do przeglądania w przeglądarce internetowej (aplikacja kliencka), następuje wysłanie żądania do odpowiedniego serwera WWW (aplikacja serwerowa). Serwer w standardowej sytuacji odpowiada klientowi, wysyłając odpowiednią treść strony WWW, którą ma wyświetlić klientowi przeglądarka.

Najpierw zajmiemy się **komunikacją wykorzystującą gniazda sieciowe**, które pozwalają traktować sieć tak, jakby były to pliki — program odczytuje dane z **gniazda** lub je do niego zapisuje w taki sam sposób, jakby odczytywał dane z pliku lub do niego zapisywał. Gniazdo sieciowe to pewna konstrukcja programowa reprezentująca jedną końcówkę połączenia. Pokażemy, jak tworzyć i obsługiwać **gniazda strumieniowe** i **gniazda datagramowe**. **Gniazdo strumieniowe** powoduje ustanowienie stałego **połączenia** z innym procesem. Po jego ustanowieniu dane przechodzą między procesami jako ciągłe **strumienie**. Mówi się, że gniazda strumieniowe zapewniają **usługi zorientowane połączeniowo**. Protokołem używanym w trakcie takiej transmisji jest TCP (ang. *Transmission Control Protocol*).

Gniazda datagramowe wysyłają poszczególne **pakiety** informacji. Używany tu protokół **UDP** (ang. *User Datagram Protocol*) jest usługą bezpołączeniową i nie gwarantuje, że pakiety dotrą w jakiejś konkretnej **kolejności**. W przypadku UDP pakiety mogą zostać **utracone** lub **zduplikowane**. Niezbędny jest dodatkowy kod, aby poprawnie obsłużyć takie sytuacje (jeśli programista zdecyduje się na ich obsługę). UDP stosuje się w aplikacjach sieciowych, które nie wymagają sprawdzania błędów ani pewności transmisji oferowanej przez TCP. Gniazda strumieniowe i protokół TCP to rozwiązania odpowiednie dla większości aplikacji sieciowych pisanych w języku Java.



26.1. Wskazówka poprawiająca wydajność

Usługi bezpołączeniowe najczęściej zapewniają lepszą wydajność, ale mniejszą pewność działania niż usługi połączeniowe.



26.1. Wskazówka poprawiająca przenośność kodu

Protokoły TCP, UDP i inne umożliwiają wzajemną współpracę systemów heterogenicznych (czyli wykorzystujących różne procesory i systemy operacyjne).

Osoby zainteresowane znajdą pod adresem:

<http://www.deitel.com/books/jhttp11>

studium przypadku pochodzące z jednego ze starszych wydań książki. Zawiera ono implementację aplikacji czatowej działającą na zasadzie **rozgłaszania**, w której serwer publikuje informacje, a **wiele** klientów może je **subskrybować**. Gdy serwer publikuje informację, otrzymują ją **wszyscy** subskrybenci.

26.2. Odczyt pliku z serwera WWW

W aplikacji z rysunku 26.1 został użyty komponent **JEditorPane** z frameworku Swing (pakiet `javax.swing`), aby wyświetlić zawartość pliku pobranego z serwera WWW. Użytkownik wpisuje adres URL w `TextField` u góry okna, a aplikacja wyświetla odpowiedni dokument (jeśli istnieje) w `JEditorPane`. Klasa `JEditorPane` potrafi renderować zarówno zwykły tekst, jak i tekst sformatowany za pomocą kodu HTML, co ilustrują dwa zrzuty ekranu (rysunek 26.2). Aplikacja może więc działać jako bardzo prosta przeglądarka internetowa. Aplikacja ilustruje sposób użycia zdarzeń **HyperLinkEvent** występujących w momencie, gdy użytkownik kliknie hiperłącze w dokumencie HTML.

```
1 // Rysunek 26.1. ReadServerFile.java
2 // Odczyt pliku przez otwarcie połączenia na podstawie adresu URL
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.io.IOException;
7 import javax.swing.JEditorPane;
8 import javax.swing.JFrame;
9 import javax.swing.JOptionPane;
```

Rysunek 26.1. Odczyt pliku przez otwarcie połączenia na podstawie adresu URL

```

10 import javax.swing.JScrollPane;
11 import javax.swing.JTextField;
12 import javax.swing.event.HyperlinkEvent;
13 import javax.swing.event.HyperlinkListener;
14
15 public class ReadServerFile extends JFrame
16 {
17     private JTextField enterField; // Kontrolka JTextField do wpisania adresu
18     private JEditorPane contentsArea; // Kontrolka wyświetlająca zawartość strony
19
20     // Konfiguracja interfejsu graficznego
21     public ReadServerFile()
22     {
23         super("Prosta przeglądarka internetowa");
24
25         // Utwórz pole tekstowe enterField i zarejestruj kod nasłuchujący zdarzeń
26         enterField = new JTextField("Wpisz tu adres URL");
27         enterField.addActionListener(
28             new ActionListener()
29             {
30                 // Pobierz dokument wskazany przez użytkownika
31                 public void actionPerformed(ActionEvent event)
32                 {
33                     getPage(event.getActionCommand());
34                 }
35             }
36         );
37
38         add(enterField, BorderLayout.NORTH);
39
40         contentsArea = new JEditorPane(); // Tworzenie contentsArea
41         contentsArea.setEditable(false);
42         contentsArea.addHyperlinkListener(
43             new HyperlinkListener()
44             {
45                 // Jeśli użytkownik kliknie hiperłącze, przejdź do wskazanej strony
46                 public void hyperlinkUpdate(HyperlinkEvent event)
47                 {
48                     if (event.getEventType() ==
49                         HyperlinkEvent.EventType.ACTIVATED)
50                         getPage(event.getURL().toString());
51                 }
52             }
53         );
54
55         add(new JScrollPane(contentsArea), BorderLayout.CENTER);
56         setSize(400, 300); // Ustaw rozmiar okna
57         setVisible(true); // Pokaż okno
58     }
59
60     // Wczytaj dokument
61     private void getPage(String location)
62     {
63         try // Wczytaj dokument i wyświetl lokalizację
64         {
65             contentsArea.setPage(location); // Ustaw stronę

```

Rysunek 26.1. Odczyt pliku przez otwarcie połączenia na podstawie adresu URL — ciąg dalszy

```

66         enterField.setText(location); // Ustaw tekst
67     }
68     catch (IOException ioException)
69     {
70         JOptionPane.showMessageDialog(this,
71             "Błąd pobierania wskazanego adresu URL", "Niepoprawny URL",
72             JOptionPane.ERROR_MESSAGE);
73     }
74 }
75 }

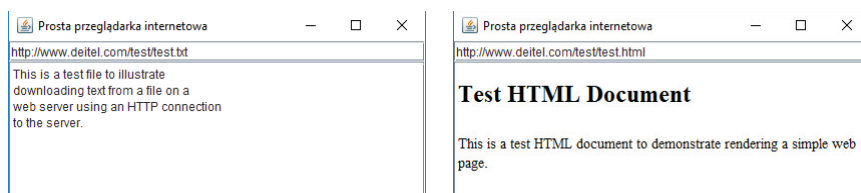
```

Rysunek 26.1. Odczyt pliku przez otwarcie połączenia na podstawie adresu URL

```

1 // Rysunek 26.2. ReadServerFileTest.java
2 // Tworzenie i uruchamianie ReadServerFile
3 import javax.swing.JFrame;
4
5 public class ReadServerFileTest
6 {
7     public static void main(String[] args)
8     {
9         ReadServerFile application = new ReadServerFile();
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11    }
12 }

```



Rysunek 26.2. Tworzenie i uruchamianie ReadServerFile

Klasa aplikacji `ReadServerFile` zawiera pole `enterField` typu `JTextField`, w którym użytkownik wpisuje adres URL pliku do odczytania, oraz pole `contentsArea` typu `JEditorPane`, które wyświetla zawartość pobranego pliku. Gdy użytkownik naciśnie klawisz *Enter* w polu `enterField`, aplikacja wywoła metodę `actionPerformed` (wiersze od 31. do 34.). Wiersz 33. używa metody `getActionCommand` z `ActionEvent`, aby pobrać tekst wpisany przez użytkownika w `JTextField` i przekazać go do metody `getPage` (wiersze od 61. do 74.).

Wiersz 65. wywołuje metodę `setPage` z `JEditorPane`, aby pobrać dokument wskazany przez `location` i wyświetlić go w `JEditorPane`. Jeśli pojawi się błąd pobierania dokumentu, metoda `setPage` zgłosi `IOException`. Jeśli zostanie wskazany niepoprawny adres URL, metoda zgłosi wyjątek `MalformedURLException` (podklasę klasy `IOException`). Jeśli dokument wczyta się bez przeszkód, wiersz 66. wyświetli aktualną lokalizację w polu `enterField`.

Najczęściej dokument HTML zawiera **hiperłącza**, które po kliknięciu przenoszą użytkownika do innego dokumentu. Jeżeli `JEditorPane` zawiera dokument HTML i użytkownik kliknie hiperłącze, `JEditorPane` wygeneruje zdarzenie `HyperlinkEvent` (pakiet `javax.swing.event`) i poinformuje wszystkie nasłuchujące

obiekty **HyperlinkListener** (pakiet `javax.swing.event`) o zajściu zdarzenia. Wiersze od 42. do 53. rejestrują **HyperlinkListener** w celu obsługi zdarzeń **HyperlinkEvent**. Gdy zajdzie zdarzenie, program wywoła metodę **hyperlinkUpdate** (wiersze od 46. do 51.). Wiersze 48. i 49. wykorzystują metodę **getEventType** z **HyperlinkEvent** do określenia rodzaju zdarzenia. Klasa **HyperlinkEvent** zawiera publiczną, zagnieżdżoną klasę o nazwie **EventType**, która deklaruje trzy statyczne obiekty **EventType** reprezentujące typy zdarzeń. Obiekt **ACTIVATED** wskazuje, że użytkownik kliknął hiperłącze, aby zmienić stronę WWW, obiekt **ENTERED** wskazuje, że użytkownik umieścił kursor myszy na łączu, a **EXITED** wskazuje, że użytkownik przesunął kursor myszy poza hiperłącze. Jeśli hiperłącze zostało uaktywnione (**ACTIVATED**), wiersz 50. używa metody **getURL** z **HyperlinkEvent**, aby pobrać adres URL reprezentowany przez hiperłącze. Metoda **toString** konwertuje zwrócony adres URL na tekst, aby można go było przekazać do metody **getPage**.



26.1. Obserwacja związana z wyglądem i działaniem

*Komponent **JEditorPane** generuje zdarzenia **HyperlinkEvent** tylko w sytuacji, gdy ma wyłączony tryb edycji.*

26.3. Wykonanie prostego serwera przy użyciu gniazd strumieniowych

Przedstawiony wcześniej przykład wykorzystywał **wysokopoziomowe** funkcjonalności Javy związane z komunikacją między aplikacjami. W przykładzie nie dokonywaliśmy samodzielnego zestawienia połączenia między klientem i serwerem. To komponent **JEditorPane** odpowiadał za ustanowienie połączenia z serwerem i pobranie danych. W tym podrozdziale sami zaczniemy tworzyć aplikacje komunikujące się ze sobą.

Krok 1. Utworzenie obiektu **ServerSocket**

Wykonanie prostego serwera w Javie wymaga pięciu kroków. Pierwszy z nich polega na utworzeniu obiektu **ServerSocket**. Wywołanie konstruktora **ServerSocket** w postaci:

```
ServerSocket server = new ServerSocket(numerPortu, długośćKolejki);
```

rejestruje dostępny **numer portu** TCP i wskazuje maksymalną liczbę klientów, które mogą czekać na połączenie z serwerem (czyli określa **długość kolejki**). Numer portu służy klientom do znalezienia aplikacji na serwerze. Nazywa się to czasem **punktem połączenia**. Jeśli kolejka jest pełna, serwer odmawia połączenia następnym klientom. Konstruktor ustanawia port, na którym serwer będzie czekał na połączenia od klientów — to tak zwane **dowiązanie portu na serwerze**. Każdy klient będzie prosił serwer o połączenie właśnie na tym **porcie**. W danym momencie tylko jedna aplikacja może być dowiązana na wybranym serwerze do konkretnego portu.



26.1. Obserwacja z poziomu inżynierii oprogramowania

Numer portu może się mieścić w zakresie od 0 do 65 535. Większość systemów operacyjnych rezerwuje porty o numerach poniżej 1024 na usługi systemowe (np. obsługę serwerów e-mail i WWW). W większości sytuacji portów tych nie należy stosować jako portów połączeniowych we własnych aplikacjach. Niektóre systemy operacyjne wymagają specjalnych uprawnień dostępowych, aby móc dowiązać aplikację do portu o numerze poniżej 1024.

Krok 2. Oczekiwanie na połączenie

Programy zarządzają połączeniem z klientem za pomocą obiektów **Socket**. W tym kroku serwer czeka nieskończenie długo (w sposób **blokujący**) na próbę połączenia podjętą przez klient. Aby nasłuchiwać połączeń od klientów, program wywołuje metodę **accept** klasy `ServerSocket`:

```
Socket connection = server.accept();
```

Metoda zwraca obiekt `Socket` reprezentujący połączenie nawiązane z klientem. Obiekt umożliwia serwerowi interakcję z klientem. W rzeczywistości interakcja z klientem jest realizowana na innym porcie niż **punkt połączenia**. Dzięki temu port wskazany w **kroku 1.** może być ponownie użyty w aplikacji wielowątkowej do nawiązania połączenia z innym klientem. Zilustrujemy to w podrozdziale 26.7.

Krok 3. Pobranie strumienia wejścia – wyjścia dla gniazda

Następny krok polega na pobraniu obiektów `OutputStream` i `InputStream` pozwalających na komunikację z klientem poprzez wysyłanie i odbieranie bajtów. Serwer wysyła informacje do klienta poprzez `OutputStream` i otrzymuje informacje od klienta poprzez `InputStream`. Do pobrania referencji do obiektu `OutputStream` służy metoda **getOutputStream** klasy `Socket`, a do pobrania referencji do obiektu `InputStream` — metoda **getInputStream** klasy `Socket`.

Obiekty strumieni służą do wysyłania lub odbierania poszczególnych bajtów lub ciągów bajtów za pomocą odpowiednio metody `write` z `OutputStream` i metody `read` z `InputStream`. Często znacznie wygodniejszym rozwiązaniem jest przesyłanie wartości typów podstawowych (np. `int` lub `double`) albo obiektów typu `Serializable` (np. tekstów lub innych obiektów obsługujących serializację). W takiej sytuacji zastosuj techniki poznane w rozdziale 15., aby otoczyć obiekty `OutputStream` i `InputStream` uzyskane z `Socket` innymi typami strumieni (np. `ObjectOutputStream` i `ObjectInputStream`). Oto przykład:

```
ObjectInputStream input =
    new ObjectInputStream(connection.getInputStream());
ObjectOutputStream output =
    new ObjectOutputStream(connection.getOutputStream());
```

Piękno tego rozwiązania polega na tym, że wszystko, co serwer zapisze w `ObjectOutputStream`, jest wysyłane poprzez `OutputStream` i trafia do obiektu `InputStream` klienta, a wszystko, co klient zapisze w swoim `OutputStream` (dzięki `ObjectOutputStream`), pojawi się w obiekcie `InputStream` serwera. Przesyłanie danych poprzez sieć jest dla aplikacji w zasadzie niewidoczne; w pełni obsługuje to kod bibliotek Javy.

Krok 4. Przetwarzanie

Ten krok odpowiada za przetwarzanie, czyli klient i serwer komunikują się ze sobą za pomocą obiektów `OutputStream` i `InputStream`.

Krok 5. Zamknięcie połączenia

W tym kroku, gdy zostanie zakończona transmisja, serwer zamyka połączenie, wywołując metodę `close` obu strumieni i obiektu `Socket`.

**26.2. Obserwacja z poziomu inżynierii oprogramowania**

Dzięki gniazdom sieciowym obsługa sieciowego wejścia – wyjścia wygląda w Javie bardzo podobnie do sekwencyjnej obsługi plików. Obiekty `Socket` ukrywają większość złożoności związanej z programowaniem sieciowym.

**26.3. Obserwacja z poziomu inżynierii oprogramowania**

Wielowątkowy serwer może pobrać obiekt `Socket` utworzony przez każde z wywołań metody `accept`, a następnie utworzyć wątek obsługujący to połączenie. W alternatywnym rozwiązaniu serwer stosuje pulę wątków (zbiór utworzonych wcześniej wątków) gotowych do obsługi ruchu sieciowego po otrzymaniu nowych obiektów `Socket`. Techniki te pozwalają serwerom na jednoczesną obsługę wielu połączeń od klientów.

**26.2. Wskazówka poprawiająca wydajność**

W wysoce wydajnych systemach, w których istnieje dostęp do dużej ilości wolnej pamięci operacyjnej, serwer wielowątkowy tworzy wcześniej pulę wątków, aby móc szybko przekazywać do nich nowe obiekty `Socket`. W takiej sytuacji, gdy serwer otrzyma połączenie, *nie pojawia się narzut związany z utworzeniem nowego wątku*. Po zamknięciu połączenia obsługujący je wątek wraca do puli, by zostać ponownie użytym.

26.4. Wykonanie prostego klienta przy użyciu gniazd strumieniowych

Wykonanie w Javie klienta, który używa gniazd strumieniowych, wymaga czterech kroków.

Krok 1. Utworzenie obiektu `Socket` w celu połączenia z serwerem

W tym kroku stworzymy obiekt `Socket`, aby połączyć się z serwerem. Konstruktor `Socket` powoduje nawiązanie połączenia. Instrukcja:

```
Socket connection = new Socket(adresSerwera, port);
```

używa konstruktora dwuargumentowego przyjmującego adres serwera i numer portu. Jeśli uda się nawiązać połączenie, instrukcja zwróci obiekt `Socket`. Nieudane połączenie spowoduje zgłoszenie wyjątku będącego podklasą klasy `IOException`, więc wiele programów po prostu wychwytuje wszystkie wyjątki `IOException`. Wyjątek `UnknownHostException` wystąpi, gdy system nie będzie w stanie rozwiązać nazwy serwera wskazanej w pierwszym argumencie na adres IP.

Krok 2. Pobranie strumienia wejścia – wyjścia dla gniazda

Następny krok polega na użyciu metod `getInputStream` i `getOutputStream` klasy `Socket` w celu pobrania obiektów `InputStream` i `OutputStream`. Jak wspomnieliśmy w poprzednim podrozdziale, można zastosować techniki przedstawione w rozdziale 15., aby otoczyć obiekty `InputStream` i `OutputStream` innymi typami obiektów. Jeśli serwer wysyła informacje w postaci pewnego konkretnego typu, klient powinien stosować ten sam format. Jeżeli serwer wysyła wartości za pomocą `ObjectOutputStream`, klient powinien je odczytywać za pomocą `ObjectInputStream`.

Krok 3. Przetwarzanie

Ten krok odpowiada za przetwarzanie, czyli klient i serwer komunikują się ze sobą za pomocą obiektów `OutputStream` i `InputStream`.

Krok 4. Zamknięcie połączenia

W tym kroku, gdy zostanie zakończona transmisja, klient zamyka połączenie, wywołując metodę `close` obu strumieni i obiektu `Socket`. Klient musi stwierdzić, czy serwer zakończył już przysyłanie informacji, zanim wywoła metody `close`. Przykładowo metoda `read` z `InputStream` zwraca `-1`, jeśli wykryje koniec strumienia (nazywany też końcem pliku, w skrócie EOF). Jeśli to obiekt `ObjectInputStream` odczytuje informacje z serwera, zgłosi wyjątek `EOFException` w sytuacji próby odczytu wartości ze strumienia i wykrycia końca strumienia.

26.5. Interakcja klienta i serwera wykorzystująca gniazda strumieniowe

W kodach z rysunków 26.3 i 26.5 zostały użyte gniazda strumieniowe oraz obiekty `ObjectInputStream` i `ObjectOutputStream` do stworzenia prostej aplikacji czatowej typu klient – serwer. Serwer czeka na próbę połączenia z klientem. Gdy klient połączy się z serwerem, serwer wysyła do niego obiekt typu `String` (przypomnijmy, że teksty to obiekty typu `Serializable`) wskazujący na udane połączenie. Klient wyświetla komunikat. Zarówno część kliencka, jak i serwerowa aplikacji zawiera pola tekstowe pozwalające użytkownikowi napisać komunikat i wysłać go do drugiej części aplikacji. Gdy klient lub serwer wyśle tekst "PRZERWIJ", połączenie ulega przerwaniu. W tym momencie serwer zaczyna oczekiwać połączenia od nowego klienta. Deklarację klasy `Server` przedstawia rysunek 26.3. Deklarację klasy `Client` przedstawia rysunek 26.5. Kilka zrzutów ekranu ilustrujących interakcję między klientem i serwerem przedstawia rysunek 26.6.

Klasa *Server*

Konstruktor `Server` (rysunek 26.3, wiersze od 30. do 55.) tworzy interfejs graficzny serwera, który składa się z komponentów `TextField` i `TextArea`. Serwer wyświetla wyniki w komponencie `TextArea`. Gdy wykona się metoda `main` (wiersze od 6. do 11. z rysunku 26.4), utworzy ona obiekt `Server`, wskaże domyślną operację zamknięcia okna i wywoła metodę `runServer` (wiersze od 57. do 86. z rysunku 26.3).

```

1 // Rysunek 26.3. Server.java
2 // Część serwerowa połączenia typu klient – serwer za pomocą gniazda strumieniowego
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9 import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Server extends JFrame
19 {
20     private JTextField enterField; // Pobiera komunikat od użytkownika
21     private JTextArea displayArea; // Wyświetla informacje użytkownikowi
22     private ObjectOutputStream output; // Strumień wyjściowy do klienta
23     private ObjectInputStream input; // Strumień wejściowy od klienta
24     private ServerSocket server; // Gniazdo serwerowe
25     private Socket connection; // Połączenie do klienta
26     private int counter = 1; // Licznik połączeń
27
28     // Konfiguracja interfejsu graficznego
29     public Server()
30     {
31         super("Serwer");
32
33         enterField = new JTextField(); // Utworzenie enterField
34         enterField.setEditable(false);
35         enterField.addActionListener(
36             new ActionListener()
37             {
38                 // Wysłanie komunikatu do klienta
39                 public void actionPerformed(ActionEvent event)
40                 {
41                     sendData(event.getActionCommand());
42                     enterField.setText("");
43                 }
44             }
45         );
46
47         add(enterField, BorderLayout.NORTH);
48
49         displayArea = new JTextArea(); // Utworzenie displayArea
50         add(new JScrollPane(displayArea), BorderLayout.CENTER);
51
52         setSize(300, 150); // Ustaw rozmiar okna
53         setVisible(true); // Wyświetl okno
54     }
55
56     // Skonfiguruj i uruchom serwer

```

Rysunek 26.3. Część serwerowa połączenia typu klient – serwer za pomocą gniazda strumieniowego

```

57 public void runServer()
58 {
59     try // Skonfiguruj serwer, aby odbierać połączenia; przetwarzanie połączeń
60     {
61         server = new ServerSocket(12345, 100); // Utworzenie ServerSocket
62
63         while (true)
64         {
65             try
66             {
67                 waitForConnection(); // Czekaj na połączenie
68                 getStreams(); // Pobierz strumienie wejściowy i wyjściowy
69                 processConnection(); // Przetwórz połączenie
70             }
71             catch (EOFException eofException)
72             {
73                 displayMessage("\nPoleczenie zakończone przez serwer");
74             }
75             finally
76             {
77                 closeConnection(); // Zamknięcie połączenia
78                 ++counter;
79             }
80         }
81     }
82     catch (IOException ioException)
83     {
84         ioException.printStackTrace();
85     }
86 }
87
88 // Czekaj na otrzymanie połączenia, a następnie wyświetl informacje o połączeniu
89 private void waitForConnection() throws IOException
90 {
91     displayMessage("Oczekiwanie na połączenie\n");
92     connection = server.accept(); // Umożliw serwerowi odbieranie połączeń
93     displayMessage("Połączenie numer " + counter + " odebrane od: " +
94         connection.getInetAddress().getHostName());
95 }
96
97 // Pobierz strumienie dla odbierania i wysyłania danych
98 private void getStreams() throws IOException
99 {
100     // Skonfiguruj strumień wyjściowy dla obiektów
101     output = new ObjectOutputStream(connection.getOutputStream());
102     output.flush(); // Opróżnij bufor, aby wymusić wysłanie informacji
103                     // → nagłówkowych
104
105     // Skonfiguruj strumień wejściowy dla obiektów
106     input = new ObjectInputStream(connection.getInputStream());
107
108     displayMessage("\nOtrzymano strumienie wejściowy i wyjściowy\n");
109 }
110
111 // Przetwórz połączenie z klientem
112 private void processConnection() throws IOException

```

Rysunek 26.3. Część serwerowa połączenia typu klient – serwer za pomocą gniazda strumieniowego — ciąg dalszy

```

112     {
113         String message = "Połączenie udane";
114         sendData(message); // Wyślij komunikat o udanym połączeniu
115
116         // Włącz enterField, aby serwer mógł wysyłać komunikaty
117         setTextFieldEditable(true);
118
119         do // Przetwarzaj komunikaty wysłane przez klient
120         {
121             try // Odczytaj komunikat i go wyświetl
122             {
123                 message = (String) input.readObject(); // Odczytaj nowy
124                                                         ↳ komunikat
125                 displayMessage("\n" + message); // Wyświetl komunikat
126             }
127             catch (ClassNotFoundException classNotFoundException)
128             {
129                 displayMessage("\nOtrzymano nieznany typ obiektu");
130             }
131         } while (!message.equals("KLIENT>>> PRZERWIJ"));
132     }
133
134     // Zamknij strumień i gniazdo
135     private void closeConnection()
136     {
137         displayMessage("\nKończenie połączenia\n");
138         setTextFieldEditable(false); // Wyłączenie enterField
139
140         try
141         {
142             output.close(); // Zamknij strumień wyjściowy
143             input.close(); // Zamknij strumień wejściowy
144             connection.close(); // Zamknij gniazdo
145         }
146         catch (IOException ioException)
147         {
148             ioException.printStackTrace();
149         }
150     }
151
152     // Wyślij wiadomość do klienta
153     private void sendData(String message)
154     {
155         try // Wyślij obiekt do klienta
156         {
157             output.writeObject("SERWER>>> " + message);
158             output.flush(); // Wymuś wysłanie do klienta
159             displayMessage("\nSERWER>>> " + message);
160         }
161         catch (IOException ioException)
162         {
163             displayArea.append("\nBłąd zapisu obiektu");
164         }
165     }
166

```

Rysunek 26.3. Część serwerowa połączenia typu klient – serwer za pomocą gniazda strumieniowego

```

167 // Modyfikuje displayArea w wątku obsługi zdarzeń
168 private void displayMessage(final String messageToDisplay)
169 {
170     SwingUtilities.invokeLater(
171         new Runnable()
172         {
173             public void run() // Aktualizuje displayArea
174             {
175                 displayArea.append(messageToDisplay); // Dodaje komunikat
176             }
177         }
178     );
179 }
180
181 // Modyfikuje enterField w wątku obsługi zdarzeń
182 private void setTextFieldEditable(final boolean editable)
183 {
184     SwingUtilities.invokeLater(
185         new Runnable()
186         {
187             public void run() // Ustawia możliwość edycji enterField
188             {
189                 enterField.setEditable(editable);
190             }
191         }
192     );
193 }
194 }

```

Rysunek 26.3. Część serwerowa połączenia typu klient – serwer za pomocą gniazda strumieniowego

```

1 // Rysunek 26.4. ServerTest.java
2 // Test aplikacji Server
3 import javax.swing.JFrame;
4
5 public class ServerTest
6 {
7     public static void main(String[] args)
8     {
9         Server application = new Server(); // Utworzenie serwera
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.runServer(); // Uruchomienie aplikacji
12    }
13 }

```

Rysunek 26.4. Test aplikacji Server

Metoda *runServer*

Metoda *runServer* (wiersze od 57. do 86. na rysunku 26.3) konfiguruje serwer w kwestii otrzymywania połączeń i przetwarza po jednym połączeniu. Wiersz 61. tworzy obiekt *ServerSocket* jako zmienną *server*, aby czekać na połączenia. Obiekt *ServerSocket* oczekuje połączeń od klientów na porcie 12345. Drugim argumentem konstruktora jest liczba połączeń z serwerem, które mogą czekać w kolejce (w tym przykładzie jest ich 100). Jeśli kolejka jest pełna, gdy klient próbuje nawiązać połączenie, zostaje ono odrzucone.



26.1. Typowy błąd programistyczny

*Wskazanie portu, który już znajduje się w użyciu, lub niepoprawnego numeru portu spowoduje zgłoszenie wyjątku **BindException** przy tworzeniu obiektu **ServerSocket**.*

Wiersz 67. wywołuje metodę `waitForConnection` (zadeklarowaną w wierszach od 89. do 95.), aby czekać na połączenie od klienta. Po ustanowieniu połączenia wiersz 68. wywołuje metodę `getStreams` (zadeklarowaną w wierszach od 98. do 108.), aby pobrać referencje do obiektów strumieni. Wiersz 69. wywołuje metodę `processConnection` (zadeklarowaną w wierszach od 111. do 132.), aby wysłać do klienta komunikat początkowy oraz przetworzyć wszystkie komunikaty otrzymane od klienta. Block `finally` (wiersze od 75. do 79.) kończy połączenie z klientem, wywołując metodę `closeConnection` (wiersze od 135. do 150.), nawet w przypadku wystąpienia wyjątku. W przypadku wyjątku metoda `displayMessage` (wiersze od 168. do 179.) wyświetla odpowiedni komunikat w komponencie `JTextArea`, używając wątku obsługi zdarzeń. Metoda `invokeLater` ze `SwingUtilities` otrzymuje obiekt `Runnable` i umieszcza go w kolejce zadań do wykonania przez wątek obsługi zdarzeń. W ten sposób mamy pewność, że nie zmieniamy komponentów interfejsu graficznego w innych wątkach niż wątek obsługi zdarzeń, co jest istotne, bo komponenty Swinga nie są bezpieczne wątkowo. Podobnej techniki używamy w metodzie `setTextFieldEditable` (wiersze od 182. do 193.), aby ustawić edytowalność pola `enterField`. Więcej informacji na temat interfejsu `Runnable` znajdziesz w rozdziale 23.

Metoda `waitForConnection`

Metoda `waitForConnection` (wiersze od 89. do 95.) używa metody `accept` z `ServerSocket` (wiersz 92.), aby czekać na połączenie od klienta. Gdy takie połączenie nastąpi, otrzymany obiekt `Socket` trafia do zmiennej `connection`. Metoda `accept` blokuje się do momentu otrzymania połączenia (wątek, który wywoła metodę `accept`, zostanie zatrzymany do momentu otrzymania połączenia od klienta). Wiersze 93. i 94. wyświetlają nazwę hosta komputera, który dokonał połączenia. Metoda `getInetAddress` zwraca obiekt `InetAddress` (pakiet `java.net`) zawierający informacje na temat komputera klienta. Metoda `getHostName` tego obiektu zwraca nazwę hosta komputera klienckiego. Na przykład specjalny adres IP (**127.0.0.1**) i nazwa hosta (**localhost**) są stosowane do testowania połączeń sieciowych lokalnego komputera (to tak zwany **adres pętli zwrotnej**). Jeśli metoda `getHostName` zostanie wywołana dla obiektu `InetAddress` zawierającego adres `127.0.0.1`, nazwą hosta zwróconą przez metodę będzie `localhost`.

Metoda `getStreams`

Metoda `getStreams` (wiersze od 98. do 108.) pobiera strumień z obiektu `Socket` i używa ich do inicjalizacji obiektów `ObjectOutputStream` (wiersz 101.) i `ObjectInputStream` (wiersz 105.). Zwróć uwagę na wywołanie metody `flush` z `ObjectOutputStream` w wierszu 102. Powoduje ona wysłanie przez serwerowy obiekt `ObjectOutputStream` nagłówka strumienia do obiektu `ObjectInputStream` po stronie klienckiej. Nagłówek strumienia zawiera między innymi takie informacje jak wersja serializacji obiektów używana do wysyłania obiektów. Informacje te są niezbędne obiektowi `ObjectInputStream`, aby przygotować się do odbioru serializowanych obiektów.



26.4. Obserwacja z poziomu inżynierii oprogramowania

Korzystając z obiektów *ObjectOutputStream* i *ObjectInputStream* do wysyłania i odbierania danych poprzez połączenie sieciowe, zawsze najpierw utwórz obiekt *ObjectOutputStream* i wywołaj metodę *flush*, aby przygotować obiekt *ObjectInputStream* znajdujący się po drugiej stronie połączenia na odbiór danych. To niezbędny element poprawnej komunikacji między aplikacjami sieciowymi korzystającymi z obiektów *ObjectOutputStream* i *ObjectInputStream*.



26.3. Wskazówka poprawiająca wydajność

Komponenty wejścia – wyjścia komputera są zazwyczaj znacznie wolniejsze od jego pamięci. Bufory wyjściowe służą do zwiększenia wydajności aplikacji przez wysyłanie większych paczek danych mniejszą liczbę razy, co ogranicza liczbę użyć komponentów wejścia – wyjścia.

Metoda *processConnection*

Wiersz 114. metody *processConnection* (wiersze od 111. do 132.) wywołuje metodę *sendData*, aby przesłać do klienta tekst "SERWER>>> Połączenie udane". Pętla z wierszy od 119. do 131. wykonuje się dopóty, dopóki serwer nie otrzyma komunikatu "KLIENT>>> PRZERWIJ". Wiersz 123. używa metody *readObject* z *ObjectInputStream*, aby odczytać tekst od klienta. Wiersz 124. wywołuje metodę *displayMessage*, aby dodać otrzymany komunikat do komponentu *JTextArea*.

Metoda *closeConnection*

Gdy transmisja została zakończona, metoda *processConnection* kończy się, a program wywołuje metodę *closeConnection* (wiersze od 135. do 150.), aby zamknąć strumienie powiązane z obiektem *Socket*, a także zamknąć sam obiekt *Socket*. W tym momencie serwer może rozpocząć oczekiwanie na nowy klient, przechodząc z pętli *while* do wiersza 67.

Obiekt *Server* przyjmuje połączenie, obsługuje je, zamyka i czeka na następne. W bardziej realistycznym scenariuszu serwer przyjmowałby połączenie, konfigurowałby jego użycie w nowym wątku i kontynuował je przy jednoczesnym oczekiwaniu na nowe połączenia od innych klientów. W ten sposób serwer działałby bardziej wydajnie, bo żądania byłyby przetwarzane współbieżnie. Przykład serwera wielowątkowego prezentujemy w podrozdziale 26.7.

Przetwarzanie interakcji użytkownika

Gdy użytkownik aplikacji serwerowej wpisze tekst w polu tekstowym i naciśnie klawisz *Enter*, program wywoła metodę *actionPerformed* (wiersze od 39. do 43.), która odczyta zawartość pola tekstowego i wywoła metodę pomocniczą *sendData* (wiersze od 153. do 165.) wysyłającą tekst do klienta. Metoda *sendData* zapisuje obiekt, wymusza opróżnienie bufora wyjściowego i dodaje ten sam tekst do obszaru tekstowego okna serwera. W tym przypadku do modyfikacji obszaru tekstu nie potrzebujemy metody *displayMessage*, bo metoda *sendData* działa jako część procedury obsługi zdarzenia, więc stanowi element wątku obsługi zdarzeń.

Klasa *Client*

Podobnie jak w klasie *Server*, także konstruktor klasy *Client* (wiersze od 29. do 56. na rysunku 26.5) tworzy interfejs graficzny aplikacji (komponenty *TextField*

i JTextArea). Obiekt Client wyświetla wyniki w obszarze tekstowym. Gdy wykona się metoda main (wiersze od 7. do 19. z rysunku 26.6), utworzy ona obiekt Client, wskaże domyślną operację zamknięcia okna i wywoła metodę runClient (wiersze od 59. do 79. na rysunku 26.5). Przykładową aplikację klienta można uruchomić z dowolnego komputera podłączonego do internetu, wskazując adres IP lub nazwę hosta komputera z uruchomionym serwerem w wierszu poleceń. Na przykład polecenie:

```
java Client 192.168.1.15
```

spowoduje próbę połączenia się z serwerem uruchomionym pod adresem IP 192.168.1.15.

```

1 // Rysunek 26.5. Client.java
2 // Część kliencka połączenia między klientem i serwerem za pomocą gniazda
  ↳ strumieniowego
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.InetAddress;
8 import java.net.Socket;
9 import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Client extends JFrame
19 {
20     private JTextField enterField; // Pobiera komunikat od użytkownika
21     private JTextArea displayArea; // Wyświetla informacje użytkownikowi
22     private ObjectOutputStream output; // Strumień wyjściowy do serwera
23     private ObjectInputStream input; // Strumień wejściowy od serwera
24     private String message = ""; // Komunikat z serwera
25     private String chatServer; // Serwer dla aplikacji
26     private Socket client; // Gniazdo do połączenia z serwerem
27
28     // Inicjalizacja chatServer i konfiguracja interfejsu graficznego
29     public Client(String host)
30     {
31         super("Klient");
32
33         chatServer = host; // Ustaw serwer, z którym komunikuje się klient
34
35         enterField = new JTextField(); // Utworzenie enterField
36         enterField.setEditable(false);
37         enterField.addActionListener(
38             new ActionListener()
39             {
40                 // Wyślij komunikat do serwera
41                 public void actionPerformed(ActionEvent event)

```

Rysunek 26.5. Część kliencka połączenia typu klient – serwer za pomocą gniazda strumieniowego

```

42         {
43             sendData(event.getActionCommand());
44             enterField.setText("");
45         }
46     }
47 );
48
49     add(enterField, BorderLayout.NORTH);
50
51     displayArea = new JTextArea(); // Utworzenie displayArea
52     add(new JScrollPane(displayArea), BorderLayout.CENTER);
53
54     setSize(300, 150); // Ustaw rozmiar okna
55     setVisible(true); // Pokaż okno
56 }
57
58 // Połącz z serwerem i przetwórz komunikaty od serwera
59 public void runClient()
60 {
61     try // Połącz z serwerem, pobierz strumienie i przetwórz połączenie
62     {
63         connectToServer(); // Utwórz Socket w celu utworzenia połączenia
64         getStreams(); // Pobierz strumienie wejściowy i wyjściowy
65         processConnection(); // Przetwórz połączenie
66     }
67     catch (EOFException eofException)
68     {
69         displayMessage("\nPołączenie zakończone przez klient");
70     }
71     catch (IOException ioException)
72     {
73         ioException.printStackTrace();
74     }
75     finally
76     {
77         closeConnection(); // Zamknij połączenie
78     }
79 }
80
81 // Połącz z serwerem
82 private void connectToServer() throws IOException
83 {
84     displayMessage("Próba połączenia\n");
85
86     // Utwórz obiekt Socket w celu połączenia z serwerem
87     client = new Socket(InetAddress.getByName(chatServer), 12345);
88
89     // Wyświetl informacje o połączeniu
90     displayMessage("Połączono z: " +
91         client.getInetAddress().getHostName());
92 }
93
94 // Pobierz strumienie do pobierania i wysyłania danych
95 private void getStreams() throws IOException
96 {
97     // Skonfiguruj strumień wyjściowy dla obiektów

```

Rysunek 26.5. Część klienta połączenia typu klient – serwer za pomocą gniazda strumieniowego — ciąg dalszy

```

98     output = new ObjectOutputStream(client.getOutputStream());
99     output.flush(); // Opróżnij bufor, aby wymusić wysłanie informacji
                      ↪nagłówkowych
100
101     // Skonfiguruj strumień wejściowy dla obiektów
102     input = new ObjectInputStream(client.getInputStream());
103
104     displayMessage("\nOtrzymano strumienie wejściowy i wyjściowy\n");
105 }
106
107 // Przetwórz połączenie z serwerem
108 private void processConnection() throws IOException
109 {
110     // Włącz enterField, aby klient mógł wysyłać komunikaty
111     setTextFieldEditable(true);
112
113     do // Przetwórz komunikaty wysłane z serwera
114     {
115         try // Odczytaj komunikat i go wyświetl
116         {
117             message = (String) input.readObject(); // Odczytaj nowy
                                                    ↪komunikat
118             displayMessage("\n" + message); // Wyświetl komunikat
119         }
120         catch (ClassNotFoundException classNotFoundException)
121         {
122             displayMessage("\nOtrzymano nieznany typ obiektu");
123         }
124     } while (!message.equals("SERWER>>> PRZERWIJ"));
125 }
126
127 // Zamknij strumienie i gniazdo
128 private void closeConnection()
129 {
130     displayMessage("\nKończenie połączenia\n");
131     setTextFieldEditable(false); // Wylączenie enterField
132
133     try
134     {
135         output.close(); // Zamknij strumień wyjściowy
136         input.close(); // Zamknij strumień wejściowy
137         client.close(); // Zamknij gniazdo
138     }
139     catch (IOException ioException)
140     {
141         ioException.printStackTrace();
142     }
143 }
144
145 // Wysłanie komunikatu na serwer
146 private void sendData(String message)
147 {
148     try // Wysłanie obiektu na serwer
149     {
150         output.writeObject("KLIENT>>> " + message);

```

Rysunek 26.5. Część klienta połączenia typu klient – serwer za pomocą gniazda strumieniowego — ciąg dalszy

```

152         output.flush(); // Wymuś wysłanie danych
153         displayMessage("\nKLIENT>>> " + message);
154     }
155     catch (IOException ioException)
156     {
157         displayArea.append("\nBłąd zapisu obiektu");
158     }
159 }
160
161 // Modyfikuje displayArea w wątku obsługi zdarzeń
162 private void displayMessage(final String messageToDisplay)
163 {
164     SwingUtilities.invokeLater(
165         new Runnable()
166         {
167             public void run() // Aktualizuje displayArea
168             {
169                 displayArea.append(messageToDisplay); // Dodaje komunikat
170             }
171         }
172     );
173 }
174
175 // Modyfikuje enterField w wątku obsługi zdarzeń
176 private void setTextFieldEditable(final boolean editable)
177 {
178     SwingUtilities.invokeLater(
179         new Runnable()
180         {
181             public void run() // Ustawia możliwość edycji enterField
182             {
183                 enterField.setEditable(editable);
184             }
185         }
186     );
187 }
188 }

```

Rysunek 26.5. Część klienta połączenia typu klient – serwer za pomocą gniazda strumieniowego — *ciąg dalszy*

```

1 // Rysunek 26.6. ClientTest.java
2 // Test aplikacji Client
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main(String[] args)
8     {
9         Client application; // Deklaracja aplikacji klienckiej
10
11         // Jeśli nie podano argumentów wiersza polecenia
12         if (args.length == 0)
13             application = new Client("127.0.0.1"); // Połącz z localhost
14         else
15             application = new Client(args[0]); // Użyj args jako danych połączenia

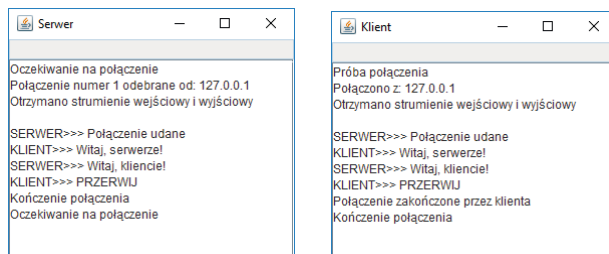
```

Rysunek 26.6. Test aplikacji Client

```

16
17     application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     application.runClient(); // Uruchom aplikację kliencką
19 }
20 }

```



Rysunek 26.6. Test aplikacji Klient

Metoda `runClient`

Metoda `runClient` klasy `Client` (wiersze od 59. do 79. z rysunku 26.5) konfiguruje połączenie z serwerem, przetwarza otrzymane od serwera informacje i zamyka połączenie po zakończeniu komunikacji. Wiersz 63. wywołuje metodę `connect` `↳ToServer` (zadeklarowaną w wierszach od 82. do 92.) w celu połączenia z serwerem. Po uzyskaniu połączenia wiersz 64. wywołuje metodę `getStreams` (wiersze od 95. do 105.), aby pobrać referencje do obiektów strumieni z `Socket`. Następnie wiersz 65. wywołuje metodę `processConnection` (wiersze od 108. do 126.), aby otrzymać i wyświetlić wiadomości przesyłane przez serwer. Blok `finally` (wiersze od 75. do 78.) wywołuje metodę `processConnection` (wiersze od 129. do 144.), aby zamknąć strumienie i gniazdo nawet w sytuacji, gdy wystąpi wyjątek. Metoda `displayMessage` (wiersze od 162. do 173.) jest przez te metody wywoływana, aby przy użyciu wątku obsługi zdarzeń wyświetlić odpowiednie komunikaty w obszarze tekstowym.

Metoda `connectToServer`

Metoda `connectToServer` (wiersze od 82. do 92.) tworzy obiekt `Socket` o nazwie `client` (wiersz 87.), aby ustanowić nowe połączenie. Argumentami konstruktora są adres IP serwera i numer portu (12345), na którym aplikacja na serwerze oczekuje połączenia. W pierwszym argumencie metoda statyczna `getByName` z `Inet` `↳Address` zwraca obiekt zawierający adres IP wskazany jako argument wiersza poleceń (lub 127.0.0.1, jeśli nie wskazano żadnego adresu). Metoda `getByName` może przyjąć rzeczywisty adres IP lub nazwę hosta. Pierwszy argument można zapisać również w inny sposób. Jeśli pierwszym argumentem jest lokalny komputer, można zastosować jedną z następujących wersji:

```

InetAddress.getByName("localhost")
InetAddress.getLocalHost()

```

Inne wersje konstruktora `Socket` przyjmują adres IP lub nazwę hosta jako tekst. W takiej sytuacji dla lokalnego komputera można użyć wartości "127.0.0.1" lub "localhost". Zdecydowaliśmy się na zaprezentowanie związku klient – serwer przez łączenie dwóch aplikacji działających na tym samym komputerze. Obiekt `InetAddress` dla innego komputera można uzyskać, wskazując adres IP lub nazwę

hosta jako argument metody `getByName` z `InetAddress`. Drugim argumentem konstruktora `Socket` jest numer portu po stronie serwerowej. **Musi** on pasować do portu, na którym serwer oczekuje połączeń (**punktu połączenia**). Po uzyskaniu połączenia wiersze 90. i 91. wyświetlają komunikat w obszarze tekstu wskazujący nazwę komputera, z którym się połączono.

Klasa `Client` używa `ObjectOutputStream` do wysyłania danych na serwer i `ObjectInputStream` do pobierania danych z serwera. Metoda `getStreams` (wiersze od 95. do 105.) tworzy obiekty `ObjectOutputStream` i `ObjectInputStream`, które używają strumieni powiązanych z gniazdem `client`.

Metody `processConnection` i `closeConnection`

Metoda `processConnection` (wiersze od 108. do 126.) zawiera pętlę, która wykonuje się do momentu otrzymania komunikatu "SERWER>>> PRZERWIJ". Wiersz 117. odczytuje obiekt `String` z serwera. Wiersz 118. wywołuje `displayMessage`, aby dodać nowy komunikat do obszaru tekstowego. Gdy zakończy się przesyłanie informacji, metoda `closeConnection` (wiersze od 129. do 144.) zamknie strumień i gniazdo sieciowe.

Przetwarzanie interakcji użytkownika

Gdy użytkownik aplikacji klienckiej wpisze tekst w polu tekstowym i naciśnie klawisz *Enter*, program wywoła metodę `actionPerformed` (wiersze od 41. do 45.), która odczyta zawartość pola tekstowego i wywoła metodę pomocniczą `sendData` (wiersze od 147. do 159.) wysyłającą tekst do serwera. Metoda `sendData` zapisuje obiekt, wymusza opróżnienie bufora wyjściowego i dodaje ten sam tekst do obszaru tekstowego okna klienta. W tym przypadku do modyfikacji obszaru tekstu nie potrzebujemy metody `displayMessage`, bo metoda `sendData` działa jako część procedury obsługi zdarzenia.

26.6. Datagramy — bezpołączeniowa interakcja między klientem i serwerem

Do tej pory zajmowaliśmy się transmisją opartą na strumieniach i wykorzystującą trwałe połączenia. Teraz przyjrzymy się **bezpoleczeniowej transmisji za pomocą datagramów**.

Transmisja połączeniowa przypomina w działaniu system telefoniczny: dzwoniczymy, a po drugiej stronie odbiera osoba, z którą chcemy rozmawiać. Połączenie na czas tej rozmowy jest trwałe, **nawet w sytuacji, gdy przez pewien czas nikt nie mówi**.

Transmisja bezpołączeniowa z użyciem datagramów przypomina bardziej komunikowanie się listami za pośrednictwem poczty. Jeśli dłuższa wiadomość nie zmieści się do jednej koperty, dzieli się ją na mniejsze i umieszcza w odpowiednio ponumerowanych kopertach. Wszystkie listy wysyłamy jednocześnie. Listy mogą przyjść **we właściwej kolejności, w innej kolejności lub w ogóle nie dotrzeć do adresata** (ostatni przypadek jest rzadki). Odbiorca **układa** koperty w odpowiednim porządku, zanim zacznie odczytywać treść wiadomości.

Jeśli wiadomość jest na tyle mała, że mieści się w jednej kopercie, nie trzeba martwić się problemem nieodpowiedniej kolejności, ale wciąż może się zdarzyć, że informacja nie dotrze do adresata. Pewną przewagą datagramów nad usługami pocztowymi jest to, że czasem odbiorca otrzymuje zduplikowane wiadomości.

Na rysunkach od 26.7 do 26.10 wykorzystujemy datagramy do wysyłania między aplikacją kliencką i aplikacją serwerową pakietów informacji protokołem UDP (*User Datagram Protocol*). W aplikacji klienckiej (rysunek 26.9) użytkownik wpisuje komunikat w polu tekstowym i naciska klawisz *Enter*. Program zamienia treść wiadomości na tablicę byte i umieszcza ją w datagramie wysylnym na serwer. Serwer (rysunki 26.7 i 26.8) otrzymuje pakiet, wyświetla treść wiadomości i **odsyła** pakiet do klienta. Po otrzymaniu pakietu klient wyświetla jego zawartość.

Klasa Server

Klasa Server (rysunek 26.7) deklaruje dwa obiekty **DatagramPacket** używane przez serwer do wysyłania i otrzymywania wiadomości oraz jeden obiekt **DatagramSocket**, który odpowiada za wysyłanie i otrzymywanie pakietów. Konstruktor (wiersze od 19. do 37.) wywoływany w metodzie *main* (wiersze od 7. do 12. na rysunku 26.8) tworzy interfejs graficzny pozwalający na wyświetlenie treści komunikatów. Wiersz 30. tworzy obiekt **DatagramSocket** w bloku try. Wiersz 30. z rysunku 26.7 używa konstruktora **DatagramSocket**, który przyjmuje numer portu w postaci liczby jako argument (w przykładzie jest to 5000), aby dołączyć serwer do portu, na który klient będzie przysyłać informacje. Klasa **Client** musi używać tego samego numeru portu w kodzie wysyłającym pakiety. Wyjątek **SocketException** pojawi się, jeśli konstruktorowi obiektu **DatagramSocket** nie uda się dokonać dołączenia do wskazanego portu.



26.2. Typowy błąd programistyczny

*Wskazanie portu będącego aktualnie w użyciu lub portu niepoprawnego w trakcie tworzenia obiektu **DatagramSocket** będzie skutkowało wyjątkiem **SocketException**.*

```

1 // Rysunek 26.7. Server.java
2 // Część serwerowa bezpołączeniowej komunikacji typu klient – serwer wykorzystującej
   ↳ datagramy
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.SocketException;
7 import java.awt.BorderLayout;
8 import javax.swing.JFrame;
9 import javax.swing.JScrollPane;
10 import javax.swing.JTextArea;
11 import javax.swing.SwingUtilities;
12
13 public class Server extends JFrame
14 {
15     private JTextArea displayArea; // Wyświetla otrzymane pakiety
16     private DatagramSocket socket; // Gniazdo zapewniające komunikację z klientem
17
18     // Konfiguracja interfejsu graficznego i DatagramSocket
19     public Server()
20     {
21         super("Serwer");

```

Rysunek 26.7. Część serwerowa bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy


```

22
23     displayArea = new JTextArea(); // Utworzenie displayArea
24     add(new JScrollPane(displayArea), BorderLayout.CENTER);
25     setSize(400, 300); // Ustaw rozmiar okna
26     setVisible(true); // Pokaż okno
27
28     try // Utworzenie DatagramSocket w celu wysyłania i otrzymywania pakietów
29     {
30         socket = new DatagramSocket(5000);
31     }
32     catch (SocketException socketException)
33     {
34         socketException.printStackTrace();
35         System.exit(1);
36     }
37 }
38
39 // Oczekiwanie na otrzymanie pakietów; wyświetlenie danych i wysłanie odpowiedzi
   ↪ do klienta
40 public void waitForPackets()
41 {
42     while (true)
43     {
44         try // Odbierz pakiet, wyświetl zawartość i zwróć kopię klientowi
45         {
46             byte[] data = new byte[100]; // Konfiguracja pakietu
47             DatagramPacket receivePacket =
48                 new DatagramPacket(data, data.length);
49
50             socket.receive(receivePacket); // Czekaj na otrzymanie pakietu
51
52             // Wyświetl informacje z otrzymanego pakietu
53             displayMessage("\nOtrzymany pakiet:" +
54                 "\nZ hosta: " + receivePacket.getAddress() +
55                 "\nPort hosta: " + receivePacket.getPort() +
56                 "\nDługość: " + receivePacket.getLength() +
57                 "\nZawartość:\n\t" + new String(receivePacket.getData(),
58                     0, receivePacket.getLength()));
59
60             sendPacketToClient(receivePacket); // Wysłanie pakietu do klienta
61         }
62         catch (IOException ioException)
63         {
64             displayMessage(ioException + "\n");
65             ioException.printStackTrace();
66         }
67     }
68 }
69
70 // Wysłanie kopii pakietu do klienta
71 private void sendPacketToClient(DatagramPacket receivePacket)
72     throws IOException
73 {
74     displayMessage("\n\nWysyłanie kopii danych do klienta...");
75 }

```

Rysunek 26.7. Część serwerowa bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy
— ciąg dalszy

```

76      // Utworzenie pakietu do wysłania
77      DatagramPacket sendPacket = new DatagramPacket(
78          receivePacket.getData(), receivePacket.getLength(),
79          receivePacket.getAddress(), receivePacket.getPort());
80
81      socket.send(sendPacket); // Wysłanie pakietu do klienta
82      displayMessage("Pakiet wysłany\n");
83  }
84
85  // Modyfikacja displayArea w wątku obsługi zdarzeń
86  private void displayMessage(final String messageToDisplay)
87  {
88      SwingUtilities.invokeLater(
89          new Runnable()
90          {
91              public void run() // Aktualizuje displayArea
92              {
93                  displayArea.append(messageToDisplay); // Wyświetl komunikat
94              }
95          }
96      );
97  }
98  }

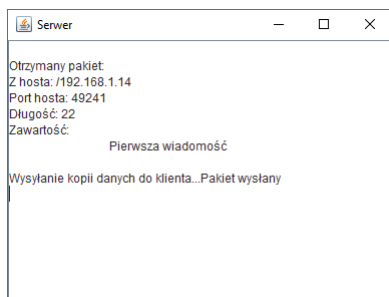
```

Rysunek 26.7. Część serwerowa bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy – ciąg dalszy

```

1  // Rysunek 26.8. ServerTest.java
2  // Klasa testująca klasę Server
3  import javax.swing.JFrame;
4
5  public class ServerTest
6  {
7      public static void main(String[] args)
8      {
9          Server application = new Server(); // Utwórz obiekt Server
10         application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11         application.waitForPackets(); // Uruchom aplikację serwerową
12     }
13 }

```



Okno serwera po otrzymaniu pakietu danych od klienta

Rysunek 26.8. Klasa testująca klasę Server

Metoda *waitForPackets*

Metoda `waitForPackets` klasy `Server` (wiersze od 40. do 68. na rysunku 26.7) używa pętli nieskończonej, aby czekać na pakiety docierające do serwera. Wiersze 47. i 48. tworzą obiekt `DatagramPacket`, w którym można przechować informacje z otrzymanego pakietu. Konstruktor `DatagramPacket` wykorzystuje w tym celu dwa argumenty — tablicę `byte`, do której trafią dane, i długość tej tablicy. Wiersz 50. używa metody `receive` z `DatagramSocket`, aby rozpocząć oczekiwanie na otrzymanie pakietu. Metoda ta blokuje działanie wątku do momentu otrzymania pakietu, a następnie umieszcza otrzymany pakiet w obiekcie `DatagramPacket` przekazanym jako argument. Metoda zgłasza wyjątek `IOException`, jeśli w trakcie otrzymywania pakietu pojawi się błąd.

Metoda *displayMessage*

Gdy pakiet zostanie odebrany, wiersze od 53. do 58. wywołują metodę `displayMessage` (zadeklarowaną w wierszach od 86. do 97.), aby dodać zawartość pakietu do obszaru tekstowego. Metoda `getAddress` z `DatagramPacket` (wiersz 54.) zwraca obiekt `InetAddress` zawierający adres IP komputera, z którego został wysłany pakiet. Metoda `getPort` (wiersz 55.) zwraca liczbę całkowitą z numerem portu, z którego klient wysłał pakiet. Metoda `getLength` (wiersz 56.) zwraca liczbę całkowitą reprezentującą liczbę otrzymanych bajtów. Metoda `getData` (wiersz 57.) zwraca tablicę `byte` z otrzymanymi danymi. Wiersze 57. i 58. inicjalizują obiekt `String` konstruktorem trójargumentowym, który przyjmuje tablicę `byte`, przesunięcie i długość. Uzyskany w ten sposób tekst jest wyświetlany w obszarze tekstowym.

Metoda *sendPacketToClient*

Po wyświetleniu pakietu wiersz 60. wywołuje metodę `sendPacketToClient` (zadeklarowaną w wierszach od 71. do 83.), aby utworzyć nowy pakiet i wysłać go do klienta. Wiersze od 77. do 79. tworzą obiekt `DatagramPacket`, przekazując do konstruktora cztery argumenty. Pierwszy argument wskazuje tablicę `byte` do wysłania, drugi określa liczbę bajtów do wysłania, trzeci wskazuje adres IP komputera, który ma otrzymać pakiet, a czwarty określa port, na którym klient oczekuje otrzymania pakietu. Wiersz 81. wysyła pakiet poprzez sieć. Metoda `send` z `DatagramSocket` zgłosi wyjątek `IOException`, jeśli w trakcie wysyłki pakietu wystąpi błąd.

Klasa *Client*

Klasa `Client` (rysunki 26.9 i 26.10) działa bardzo podobnie do klasy `Server`, ale klient wysyła pakiety tylko wówczas, gdy użytkownik wpisze treść wiadomości w polu tekstowym i naciśnie klawisz `Enter`. Kiedy dojdzie do takiej sytuacji, program wywoła metodę `actionPerformed` (wiersze od 32. do 57. na rysunku 26.9), która skonwertuje tekst wpisany przez użytkownika na tablicę `byte` (wiersz 41.). Wiersze 44. i 45. tworzą i inicjalizują obiekt `DatagramPacket`, podając tablicę `byte`, liczbę znaków, adres IP, pod który ma być wysłany pakiet (w tym przypadku `InetAddress.getLocalHost()`), i numer portu, na którym nasłuchuje część serwerowa (w przykładzie jest to port 5000). Wiersz 47. wysyła pakiet. Klient w przykładzie musi wiedzieć, że serwer oczekuje na pakiety na porcie 5000, bo w przeciwnym razie ich nie otrzyma.

Wywołanie konstruktora `DatagramSocket` (wiersz 71. na rysunku 26.9) tym razem nie jest związane z żadnymi argumentami. Konstruktor bezargumentowy umożliwia komputerowi wybór następnego wolnego portu dla datagramów. Klient nie potrzebuje żadnego konkretnego portu, ponieważ serwer, otrzymując datagram od klienta, uzyska też informację o porcie i adresie wysyłającym. W ten sposób serwer może odesłać odpowiedź do miejsca, z którego przyszła.

```

1 // Rysunek 26.9. Client.java
2 // Część kliencka bezpołączeniowej komunikacji typu klient – serwer wykorzystującej
   ↪ datagramy
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.InetAddress;
7 import java.net.SocketException;
8 import java.awt.BorderLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Client extends JFrame
18 {
19     private JTextField enterField; // Umożliwia wpisanie komunikatu
20     private JTextArea displayArea; // Wyświetlanie komunikatów
21     private DatagramSocket socket; // Gniazdo do połączenia z serwerem
22
23     // Konfiguracja interfejsu użytkownika i DatagramSocket
24     public Client()
25     {
26         super("Klient");
27
28         enterField = new JTextField("Tu wpisz wiadomość");
29         enterField.addActionListener(
30             new ActionListener()
31             {
32                 public void actionPerformed(ActionEvent event)
33                 {
34                     try // Utwórz i wyślij pakiet
35                     {
36                         // Pobierz wiadomość z pola tekstowego
37                         String message = event.getActionCommand();
38                         displayArea.append("\nWysyłanie pakietu zawierającego: " +
39                             message + "\n");
40
41                         byte[] data = message.getBytes(); // Konwersja na bajty
42
43                         // Utworzenie sendPacket
44                         DatagramPacket sendPacket = new DatagramPacket(data,
45                             data.length, InetAddress.getLocalHost(), 5000);
46
47                         socket.send(sendPacket); // Wysłanie pakietu

```

Rysunek 26.9. Część kliencka bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy

```

48         displayArea.append("Pakiet wysłany\n");
49         displayArea.setCaretPosition(
50             displayArea.getText().length());
51     }
52     catch (IOException ioException)
53     {
54         displayMessage(ioException + "\n");
55         ioException.printStackTrace();
56     }
57 }
58 }
59 );
60
61 add(enterField, BorderLayout.NORTH);
62
63 displayArea = new JTextArea();
64 add(new JScrollPane(displayArea), BorderLayout.CENTER);
65
66 setSize(400, 300); // Ustaw rozmiar okna
67 setVisible(true); // Wyświetl okno
68
69 try // Utworzenie DatagramSocket w celu wysyłania i odbierania pakietów
70 {
71     socket = new DatagramSocket();
72 }
73 catch (SocketException socketException)
74 {
75     socketException.printStackTrace();
76     System.exit(1);
77 }
78 }
79
80 // Oczekiwanie na otrzymanie pakietów z serwera i wyświetlanie ich zawartości
81 public void waitForPackets()
82 {
83     while (true)
84     {
85         try // Pobierz pakiet i wyświetl zawartość
86         {
87             byte[] data = new byte[100]; // Konfiguracja pakietu
88             DatagramPacket receivePacket = new DatagramPacket(
89                 data, data.length);
90
91             socket.receive(receivePacket); // Oczekiwanie na pakiet
92
93             // Wyświetlenie zawartości pakietu
94             displayMessage("\nOtrzymano pakiet:" +
95                 "\nZ hosta: " + receivePacket.getAddress() +
96                 "\nPort hosta: " + receivePacket.getPort() +
97                 "\nDługość: " + receivePacket.getLength() +
98                 "\nZawartość:\n\t" + new String(receivePacket.getData(),
99                     0, receivePacket.getLength()));
100         }
101         catch (IOException exception)
102         {

```

Rysunek 26.9. Część kliencka bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy
— ciąg dalszy

```

103         displayMessage(exception + "\n");
104         exception.printStackTrace();
105     }
106 }
107 }
108
109 // Modyfikacja displayArea w wątku obsługi zdarzeń
110 private void displayMessage(final String messageToDisplay)
111 {
112     SwingUtilities.invokeLater(
113         new Runnable()
114         {
115             public void run() // Aktualizacja displayArea
116             {
117                 displayArea.append(messageToDisplay);
118             }
119         }
120     );
121 }
122 }

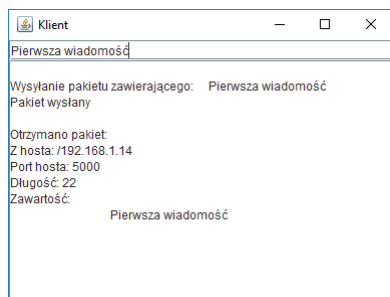
```

Rysunek 26.9. Część kliencka bezpołączeniowej komunikacji typu klient – serwer wykorzystującej datagramy
— ciąg dalszy

```

1 // Rysunek 26.10. ClientTest.java
2 // Klasa testująca klasę Client
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main(String[] args)
8     {
9         Client application = new Client(); // Utworzenie klienta
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.waitForPackets(); // Uruchomienie aplikacji klienckiej
12    }
13 }

```



Okno klienta po wysłaniu pakietu do serwera i otrzymaniu go z powrotem

Rysunek 26.10. Klasa testująca klasę Client

Metoda waitForPackets

Metoda `waitForPackets` klasy `Client` (wiersze od 81. do 107.) używa pętli nieskończonej, aby czekać na pakiety z serwera. Wiersz 91. blokuje działanie wątku do momentu otrzymania pakietu. Nie blokuje to użytkownika przed wysłaniem pakietów, ponieważ zdarzenia interfejsu graficznego są realizowane w wątku obsługi zdarzeń. Blokada zapobiega jedynie dalszemu przejściu przez kod pętli `while` w trakcie oczekiwania na nadejście pakietu. Gdy pakiet dotrze, wiersz 91. zapamiętuje go w `receivePacket`, a wiersze od 94. do 99. wywołują metodę `displayMessage`, która wyświetla w obszarze tekstowym otrzymaną treść.

26.7. Kółko i krzyżyk w wersji klient – serwer z serwerem wielowątkowym

W tym podrozdziale zaimplementujemy popularną grę w kółko i krzyżyk, używając gniazd strumieniowych i architektury klient – serwer. Program składa się z aplikacji `TicTacToeServer` (rysunki 26.11 i 26.12), która umożliwia dwóm aplikacjom `TicTacToeClient` (rysunki 26.13 i 26.14) połączenie się z serwerem i zagranie w grę. Rysunek 26.15 przedstawia przykładowe wyniki działania aplikacji.

Klasa TicTacToeServer

Gdy `TicTacToeServer` otrzyma połączenie z klientem, tworzy instancję klasy wewnętrznej `Player` (wiersze od 182. do 304. na rysunku 26.11) w celu obsłużenia klienta w osobnym wątku. Dzięki temu rozwiązaniu dwa klienty mogą się połączyć i prowadzić jedną rozgrywkę. Pierwszy klient łączący się z serwerem to gracz X, a drugi to gracz O. Gracz X wykonuje ruch jako pierwszy. Serwer przechowuje informacje o stanie tablicy, więc może sprawdzić, czy ruch gracza jest poprawny.

```

1 // Rysunek 26.11. TicTacToeServer.java
2 // Część serwerowa gry w kółko i krzyżyk
3 import java.awt.BorderLayout;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6 import java.io.IOException;
7 import java.util.Formatter;
8 import java.util.Scanner;
9 import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13 import java.util.concurrent.locks.Condition;
14 import javax.swing.JFrame;
15 import javax.swing.JTextArea;
16 import javax.swing.SwingUtilities;
17
18 public class TicTacToeServer extends JFrame
19 {
20     private String[] board = new String[9]; // Tablica
21     private JTextArea outputArea; // Wyświetlanie ruchów

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk

```

22 private Player[] players; // Tablica obiektów Players
23 private ServerSocket server; // Gniazdo serwerowe do nawiązywania połączeń
    ↳przez klienty
24 private int currentPlayer; // Śledzi gracza, który aktualnie wykonuje ruch
25 private final static int PLAYER_X = 0; // Stała dla pierwszego gracza
26 private final static int PLAYER_O = 1; // Stała dla drugiego gracza
27 private final static String[] MARKS = {"X", "O"}; // Tablica oznaczeń
28 private ExecutorService runGame; // Uruchamia graczy
29 private Lock gameLock; // Blokada gry w celu synchronizacji
30 private Condition otherPlayerConnected; // W celu oczekiwania na innego
    ↳gracza
31 private Condition otherPlayerTurn; // W celu oczekiwania na ruch innego
    ↳gracza
32
33 // Konfiguracja serwera gry i interfejsu graficznego do wyświetlania komunikatów
34 public TicTacToeServer()
35 {
36     super("Serwer gry w kółko i krzyżyk"); // Konfiguracja tytułu okna
37
38     // Utworzenie ExecutorService z wątkami dla każdego gracza
39     runGame = Executors.newFixedThreadPool(2);
40     gameLock = new ReentrantLock(); // Utworzenie blokady dla gry
41
42     // Warunek pozwalający czekać na połączenie obu graczy
43     otherPlayerConnected = gameLock.newCondition();
44
45     // Warunek dla kolei innego gracza
46     otherPlayerTurn = gameLock.newCondition();
47
48     for (int i = 0; i < 9; i++)
49         board[i] = new String(""); // Utworzenie tablicy do gry
50     players = new Player[2]; // Utworzenie tablicy graczy
51     currentPlayer = PLAYER_X; // Niech aktualnym graczem będzie pierwszy gracz
52
53     try
54     {
55         server = new ServerSocket(12345, 2); // Konfiguracja ServerSocket
56     }
57     catch (IOException ioException)
58     {
59         ioException.printStackTrace();
60         System.exit(1);
61     }
62
63     outputArea = new JTextArea(); // Utworzenie JTextArea wyświetlającego
    ↳komunikaty
64     add(outputArea, BorderLayout.CENTER);
65     outputArea.setText("Serwer czeka na połączenia\n");
66
67     setSize(300, 300); // Ustaw rozmiar okna
68     setVisible(true); // Wyświetl okno
69 }
70
71 // Czekaj na dwa połączenia, aby móc rozpocząć grę
72 public void execute()
73 {

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

74      // Czekaj na połączenie każdego z graczy
75      for (int i = 0; i < players.length; i++)
76      {
77          try // Czekaj na połączenie, utwórz obiekt Player i rozpocznij jego
              ↳ wykonywanie
78          {
79              players[i] = new Player(server.accept(), i);
80              runGame.execute(players[i]); // Rozpoczęcie wykonywania obiektu
              ↳ Player
81          }
82          catch (IOException ioException)
83          {
84              ioException.printStackTrace();
85              System.exit(1);
86          }
87      }
88
89      gameLock.lock(); // Zablokuj grę, aby poinformować wątek gracza X
90
91      try
92      {
93          players[PLAYER_X].setSuspended(false); // Wznów gracza X
94          otherPlayerConnected.signal(); // Obudź wątek gracza X
95      }
96      finally
97      {
98          gameLock.unlock(); // Odblokuj grę po poinformowaniu gracza X
99      }
100 }
101
102 // Wyświetl komunikat w outputArea
103 private void displayMessage(final String messageToDisplay)
104 {
105     // Wyświetl komunikat z poziomu wątku obsługi zdarzeń
106     SwingUtilities.invokeLater(
107         new Runnable()
108         {
109             public void run() // Uaktualnij outputArea
110             {
111                 outputArea.append(messageToDisplay); // Dodaj komunikat
112             }
113         }
114     );
115 }
116
117 // Sprawdź, czy ruch jest poprawny
118 public boolean validateAndMove(int location, int player)
119 {
120     // Jeśli to nie aktualny gracz, musi poczekać na swoją kolej
121     while (player != currentPlayer)
122     {
123         gameLock.lock(); // Zablokuj grę w oczekiwaniu na ruch drugiego gracza
124
125         try
126         {
127             otherPlayerTurn.await(); // Zaczekaj na kolej gracza

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

128         }
129         catch (InterruptedException exception)
130         {
131             exception.printStackTrace();
132         }
133         finally
134         {
135             gameLock.unlock(); // Odblokuj grę po oczekiwaniu
136         }
137     }
138
139     // Jeśli pole nie jest zajęte, wykonaj ruch
140     if (!isOccupied(location))
141     {
142         board[location] = MARKS[currentPlayer]; // Ustaw ruch na tablicy
143         currentPlayer = (currentPlayer + 1) % 2; // Zmień gracza
144
145         // Poinformuj nowego aktualnego gracza o wykonanym ruchu
146         players[currentPlayer].otherPlayerMoved(location);
147
148         gameLock.lock(); // Zablokuj grę, aby poinformować innego gracza o ruchu
149
150         try
151         {
152             otherPlayerTurn.signal(); // Poinformuj innego gracza, aby
153                                     ↪ kontynuował
154         }
155         finally
156         {
157             gameLock.unlock(); // Odblokuj grę po poinformowaniu
158         }
159
160         return true; // Poinformuj gracza, że ruch był poprawny
161     }
162     else // Ruch nie był poprawny
163     {
164         return false; // Poinformuj gracza, że ruch nie był poprawny
165     }
166
167     // Sprawdź, czy pole jest zajęte
168     public boolean isOccupied(int location)
169     {
170         if (board[location].equals(MARKS[PLAYER_X]) ||
171             board[location].equals(MARKS[PLAYER_O]))
172             return true; // Pole jest zajęte
173         else
174             return false; // Pole jest wolne
175     }
176
177     // Umieść tu kod sprawdzający zakończenie gry
178     public boolean isGameOver()
179     {
180         return false; // Pozostawiamy to jako ćwiczenie
181     }
182
183     // Prywatna klasa wewnętrzna zarządza każdym graczem jako obiektem wykonywalnym
184     private class Player implements Runnable

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

183     {
184         private Socket connection; // Połączenie z klientem
185         private Scanner input; // Dane od klienta
186         private Formatter output; // Dane do klienta
187         private int playerNumber; // Przechowuje informację o numerze gracza
188         private String mark; // Znak tego gracza
189         private boolean suspended = true; // Informacja, czy wątek jest zawieszony
190
191         // Skonfiguruj wątek gracza
192         public Player(Socket socket, int number)
193         {
194             playerNumber = number; // Zapamiętaj numer tego gracza
195             mark = MARKS[playerNumber]; // Określ znak gracza
196             connection = socket; // Zapamiętaj gniazdo klienta
197
198             try // Pobierz strumień z obiektu Socket
199             {
200                 input = new Scanner(connection.getInputStream());
201                 output = new Formatter(connection.getOutputStream());
202             }
203             catch (IOException ioException)
204             {
205                 ioException.printStackTrace();
206                 System.exit(1);
207             }
208         }
209
210         // Wyślij informację, że drugi gracz wykonał ruch
211         public void otherPlayerMoved(int location)
212         {
213             output.format("Przeciwnik wykonał ruch\n");
214             output.format("%d\n", location); // Wyślij położenie ruchu
215             output.flush(); // Wymuś wysłanie danych
216         }
217
218         // Sterowanie wykonywaniem wątku
219         public void run()
220         {
221             // Wyślij klientowi jego znak (X lub O) i przetwórz wiadomości od klienta
222             try
223             {
224                 displayMessage("Gracz " + mark + " dołączył do gry\n");
225                 output.format("%s\n", mark); // Wyślij znak gracza
226                 output.flush(); // Wymuś wysłanie danych
227
228                 // Jeśli to gracz X, zaczekaj na pojawienie się drugiego gracza
229                 if (playerNumber == PLAYER_X)
230                 {
231                     output.format("%s\n%s", "Gracz X dołączył do gry",
232                                     "Czekam na drugiego gracza\n");
233                     output.flush(); // Wymuś wysłanie danych
234
235                     gameLock.lock(); // Zablokuj grę w celu oczekiwania na drugiego
236                                     ↳ gracza
237
238                     try

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

238         {
239             while(suspended)
240             {
241                 otherPlayerConnected.await(); // Oczekiwanie na gracza O
242             }
243         }
244         catch (InterruptedException exception)
245         {
246             exception.printStackTrace();
247         }
248         finally
249         {
250             gameLock.unlock(); // Odblokuj grę po dołączeniu drugiego
                                ↳ gracza
251         }
252
253         // Wyślij informację, że dołączył drugi gracz
254         output.format("Drugi gracz dołączył do gry. Twój ruch.\n");
255         output.flush(); // Wymuś wysłanie danych
256     }
257     else
258     {
259         output.format("Gracz 0 dołączył do gry. Czekaj.\n");
260         output.flush(); // Wymuś wysłanie danych.
261     }
262
263     // Gdy gra się nie skończyła
264     while (!isGameOver())
265     {
266         int location = 0; // Inicjalizacja położenia ruchu
267
268         if (input.hasNext())
269             location = input.nextInt(); // Pobierz położenie ruchu
270
271         // Sprawdź poprawność ruchu
272         if (validateAndMove(location, playerNumber))
273         {
274             displayMessage("\nPołożenie: " + location);
275             output.format("Ruch poprawny.\n"); // Poinformuj klienta
276             output.flush(); // Wymuś wysłanie danych
277         }
278         else // Ruch nie jest poprawny
279         {
280             output.format("Ruch niepoprawny. Spróbuj ponownie.\n");
281             output.flush(); // Wymuś wysłanie danych
282         }
283     }
284 }
285 finally
286 {
287     try
288     {
289         connection.close(); // Zamknij połączenie z klientem
290     }
291     catch (IOException ioException)
292     {

```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

293         IOException.printStackTrace();
294         System.exit(1);
295     }
296 }
297 }
298
299 // Ustaw, czy wątek jest zawieszony czy nie
300 public void setSuspended(boolean status)
301 {
302     suspended = status; // Ustaw wartość zawieszenia
303 }
304 }
305 }

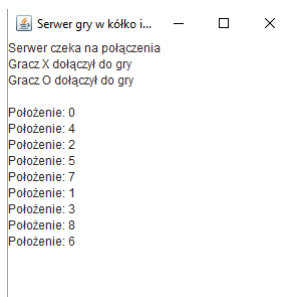
```

Rysunek 26.11. Część serwerowa gry w kółko i krzyżyk — ciąg dalszy

```

1 // Rysunek 26.12. TicTacToeServerTest.java
2 // Klasa testująca serwer gry w kółko i krzyżyk.
3 import javax.swing.JFrame;
4
5 public class TicTacToeServerTest
6 {
7     public static void main(String[] args)
8     {
9         TicTacToeServer application = new TicTacToeServer();
10        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        application.execute();
12    }
13 }

```



Rysunek 26.12. Klasa testująca serwer gry w kółko i krzyżyk

Zacznijmy od omówienia części serwerowej gry. Gdy uruchamiamy aplikację, metoda `main` (wiersze od 7. do 12. na rysunku 26.12) tworzy obiekt `TicTacToeServer` o nazwie `application`. Konstruktor (wiersze od 34. do 69. na rysunku 26.11) próbuje skonfigurować obiekt `ServerSocket`. Jeśli operacja się powiedzie, program wyświetli okno serwera, a metoda `main` wywoła metodę `execute` z `TicTacToeServer` (wiersze od 72. do 100.). Metoda `execute` przechodzi przez pętlę dwukrotnie, blokując się w wierszu 79. w oczekiwaniu na połączenie od klienta. Gdy klient się połączy, wiersz 79. tworzy obiekt `Player`, aby zarządzać połączeniem jako osobnym wątkiem, a wiersz 80. wykonuje obiekt `Player` w puli wątków `runGame`.

Gdy `TicTacToeServer` tworzy obiekt `Player`, konstruktor klasy `Player` (wiersze od 192. do 208.) otrzymuje obiekt `Socket` reprezentujący połączenie z klientem, a także pobiera strumienie wejściowy i wyjściowy. Wiersz 201. tworzy obiekt `Formatter` (rozdział 15.), otaczając nim strumień wyjściowy. Metoda `run` z `Player` (wiersze od 219. do 297.) steruje informacjami wysyłanymi do klienta i otrzymywanymi od klienta. Przede wszystkim przekazuje klientowi znak, którego będzie używał do wykonywania swoich ruchów (wiersz 225.). Wiersz 226. wywołuje metodę `flush` z `Formatter`, aby wymusić wysłanie danych do klienta. Wiersz 241. zawiesza wątek gracza X, ponieważ gracz ten może wykonać ruch dopiero po podłączeniu się gracza O.

Gdy gracz O połączy się z serwerem, gra może się rozpocząć, więc metoda `run` wchodzi do pętli `while` (wiersze od 264. do 283.). Każda iteracja pętli odczytuje liczbę całkowitą (wiersz 269.) reprezentującą lokalizację, w której klient chce umieścić znak (blokują się na czas odczytu wartości). Wiersz 272. wywołuje metodę `validateAndMove` z `TicTacToeServer` (zadeklarowaną w wierszach od 118. do 163.), aby sprawdzić ruch. Jeśli ruch jest poprawny, wiersz 275. wysyła informację do klienta z potwierdzeniem. Jeśli ruch nie jest poprawny, wysyła do klienta stosowny komunikat. Program przechowuje pola na tablicy jako wartości od 0 do 8 (pierwszy wiersz to wartości od 0 do 2, drugi od 3 do 5, a trzeci od 6 do 8).

Metoda `validateAndMove` (wiersze od 118. do 163. klasy `TicTacToeServer`) umożliwia wykonanie ruchu tylko przez jednego z graczy, co zapobiega równoczesnej próbie zmiany tablicy przez obu graczy. Jeśli gracz prześle ruch, ale to nie jego kolej, trafi on do stanu *oczekiwania* na swoją kolej. Jeśli pozycja wskazana w ruchu jest już zajęta na tablicy, metoda zwróci wartość `false`. W przeciwnym razie serwer umieści odpowiedni znak w miejscu wskazanym przez gracza (wiersz 142), poinformuje drugiego gracza (obiekt `Player`, wiersz 146.), że wykonano ruch (więc może teraz wykonać swój ruch), wywoła metodę `signal` (wiersz 152.), aby poinformować oczekujący obiekt `Player` (jeśli istnieje), że może sprawdzić ruch, i zwróci `true` (wiersz 159.), wskazując, że ruch był poprawny.

Klasa `TicTacToeClient`

Każda aplikacja `TicTacToeClient` (rysunki 26.13 i 26.14; przykładowe wyniki prezentuje rysunek 26.15) przechowuje własną wersję interfejsu użytkownika na potrzeby gry. Klient może umieścić znak tylko w pustym kwadracie. Klasa wewnętrzna `Square` (wiersze od 205. do 261. na rysunku 26.13) implementuje jedno z dziewięciu pól tablicy. Gdy aplikacja się uruchamia, tworzy komponent `JTextArea`, w którym pojawiają się komunikaty z serwera, a także używa dziewięciu obiektów `Square`. Metoda `startClient` (wiersze od 80. do 100.) otwiera połączenie z serwerem i pobiera odpowiednie strumienie z obiektu `Socket`. Wiersze 85. i 86. tworzą połączenie z serwerem. Klasa `TicTacToeClient` implementuje interfejs `Runnable`, więc osobny wątek może odczytywać komunikaty z serwera. W ten sposób użytkownik może wchodzić w interakcje z tablicą (dzięki wątkowi obsługi zdarzeń), a jednocześnie czekać na komunikaty z serwera. Po ustanowieniu połączenia w wierszu 99. następuje wykonanie klienta dzięki metodzie `execute` z `ExecutorService`. Metoda `run` (wiersze od 103. do 126.) steruje osobnym wątkiem wykonywania. Metoda najpierw odczytuje z serwera stosowany znak (X lub O, wiersz 105.), następnie, działając w pętli (wiersze od 121.

```

1 // Rysunek 26.13. TicTacToeClient.java
2 // Część kliencka gry w kółko i krzyżyk
3 import java.awt.BorderLayout;
4 import java.awt.Dimension;
5 import java.awt.Graphics;
6 import java.awt.GridLayout;
7 import java.awt.event.MouseAdapter;
8 import java.awt.event.MouseEvent;
9 import java.net.Socket;
10 import java.net.InetAddress;
11 import java.io.IOException;
12 import javax.swing.JFrame;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18 import java.util.Formatter;
19 import java.util.Scanner;
20 import java.util.concurrent.Executors;
21 import java.util.concurrent.ExecutorService;
22
23 public class TicTacToeClient extends JFrame implements Runnable
24 {
25     private JTextField idField; // Pole tekstowe do wyświetlenia znaku gracza
26     private JTextArea displayArea; // Komponent JTextArea do wyświetlania
                                   ↳ komunikatów
27     private JPanel boardPanel; // Panel dla tablicy do gry
28     private JPanel panel2; // Panel zawierający tablicę
29     private Square[][] board; // Tablica do gry
30     private Square currentSquare; // Aktualne pole
31     private Socket connection; // Połączenie z serwerem
32     private Scanner input; // Dane z serwera
33     private Formatter output; // Dane do serwera
34     private String ticTacToeHost; // Nazwa hosta dla serwera
35     private String myMark; // Znak klienta
36     private boolean myTurn; // Określa, czy to ruch tego gracza
37     private final String X_MARK = "X"; // Znacznik dla pierwszego klienta
38     private final String O_MARK = "O"; // Znacznik dla drugiego klienta
39
40     // Konfiguracja interfejsu użytkownika i tablicy
41     public TicTacToeClient(String host)
42     {
43         ticTacToeHost = host; // Ustaw nazwę serwera
44         displayArea = new JTextArea(4, 30); // Skonfiguruj komponent JTextArea
45         displayArea.setEditable(false);
46         add(new JScrollPane(displayArea), BorderLayout.SOUTH);
47
48         boardPanel = new JPanel(); // Ustaw panel dla pól tablicy
49         boardPanel.setLayout(new GridLayout(3, 3, 0, 0));
50
51         board = new Square[3][3]; // Utwórz tablicę
52
53         // Przejdź przez wiersze tablicy
54         for (int row = 0; row < board.length; row++)
55         {

```

Rysunek 26.13. Część kliencka gry w kółko i krzyżyk — ciąg dalszy

```

56      // Przejdź przez kolumny tablicy
57      for (int column = 0; column < board[row].length; column++)
58      {
59          // Utwórz pole
60          board[row][column] = new Square(" ", row * 3 + column);
61          boardPanel.add(board[row][column]); // Dodaj pole
62      }
63  }
64
65  idField = new JTextField(); // Skonfiguruj pole tekstowe
66  idField.setEditable(false);
67  add(idField, BorderLayout.NORTH);
68
69  panel2 = new JPanel(); // Skonfiguruj panel przechowujący boardPanel
70  panel2.add(boardPanel, BorderLayout.CENTER); // Dodaj panel z tablicą
71  add(panel2, BorderLayout.CENTER); // Dodaj do kontenera
72
73  setSize(300, 225); // Ustaw rozmiar okna
74  setVisible(true); // Wyświetl okno
75
76  startClient();
77  }
78
79  // Uruchom wątek klienta
80  public void startClient()
81  {
82      try // Połącz z serwerem i pobierz strumienie
83      {
84          // Połącz się z serwerem
85          connection = new Socket(
86              InetAddress.getByName(ticTacToeHost), 12345);
87
88          // Pobierz strumienie wejściowe i wyjściowe
89          input = new Scanner(connection.getInputStream());
90          output = new Formatter(connection.getOutputStream());
91      }
92      catch (IOException ioException)
93      {
94          ioException.printStackTrace();
95      }
96
97      // Utwórz i uruchom wątek roboczy dla tego klienta
98      ExecutorService worker = Executors.newFixedThreadPool(1);
99      worker.execute(this); // Uruchom klient
100  }
101
102  // Wątek sterujący umożliwia stałą aktualizację obszaru displayArea
103  public void run()
104  {
105      myMark = input.nextLine(); // Pobierz znak gracza (X lub O)
106
107      SwingUtilities.invokeLater(
108          new Runnable()
109          {
110              public void run()
111              {

```

Rysunek 26.13. Część kliencka gry w kółko i krzyżyk — ciąg dalszy

```

112         // Wyświetl znak gracza
113         idField.setText("Jesteś graczem \"" + myMark + "\"");
114     }
115 }
116 );
117
118 myTurn = (myMark.equals(X_MARK)); // Sprawdź, czy to ruch tego klienta
119
120 // Pobierz komunikaty przesłane do klienta i je wyświetl
121 while (true)
122 {
123     if (input.hasNextLine())
124         processMessage(input.nextLine());
125 }
126 }
127
128 // Przetwórz komunikaty wysłane do klienta
129 private void processMessage(String message)
130 {
131     // Wykonano poprawny ruch
132     if (message.equals("Ruch poprawny."))
133     {
134         displayMessage("Ruch poprawny. Czekaj.\n");
135         setMark(currentSquare, myMark); // Ustaw znak w polu
136     }
137     else if (message.equals("Ruch niepoprawny. Spróbuj ponownie."))
138     {
139         displayMessage(message + "\n"); // Wyświetl niepoprawny ruch
140         myTurn = true; // To nadal ruch tego klienta
141     }
142     else if (message.equals("Przeciwnik wykonał ruch"))
143     {
144         int location = input.nextInt(); // Pobierz położenie ruchu
145         input.nextLine(); // Pomiń wiersz po wartości całkowitej
146         int row = location / 3; // Oblicz wiersz
147         int column = location % 3; // Oblicz kolumnę
148
149         setMark(board[row][column],
150             (myMark.equals(X_MARK) ? O_MARK : X_MARK)); // Oznacz ruch
151         displayMessage("Przeciwnik wykonał ruch. Twoja kolej.\n");
152         myTurn = true; // Teraz kolej tego klienta
153     }
154     else
155         displayMessage(message + "\n"); // Wyświetl komunikat
156 }
157
158 // Zmodyfikuj displayArea w wątku obsługi zdarzeń
159 private void displayMessage(final String messageToDisplay)
160 {
161     SwingUtilities.invokeLater(
162         new Runnable()
163         {
164             public void run()
165             {
166                 displayArea.append(messageToDisplay); // Aktualizacja wartości
167             }
168         }
169     );

```

Rysunek 26.13. Część kliencka gry w kółko i krzyżyk — ciąg dalszy

```

168     }
169   );
170 }
171
172 // Metoda pomocnicza do ustawienia znaku na tablicy w wątku obsługi zdarzeń
173 private void setMark(final Square squareToMark, final String mark)
174 {
175     SwingUtilities.invokeLater(
176         new Runnable()
177         {
178             public void run()
179             {
180                 squareToMark.setMark(mark); // Ustaw znak w polu
181             }
182         }
183     );
184 }
185
186 // Wyślij komunikat do serwera z informacją o klikniętym polu
187 public void sendClickedSquare(int location)
188 {
189     // Jeśli to moja kolej
190     if (myTurn)
191     {
192         output.format("%d\n", location); // Wyślij położenie na serwer
193         output.flush();
194         myTurn = false; // To już nie jest moja kolej
195     }
196 }
197
198 // Ustaw aktualne pole
199 public void setCurrentSquare(Square square)
200 {
201     currentSquare = square; // Ustaw aktualne pole na argument
202 }
203
204 // Prywatna klasa wewnętrzna dla pól tablicy
205 private class Square extends JPanel
206 {
207     private String mark; // Znak do narysowania w polu
208     private int location; // Położenie pola
209
210     public Square(String squareMark, int squareLocation)
211     {
212         mark = squareMark; // Ustaw znak pola
213         location = squareLocation; // Ustaw położenie pola
214
215         addMouseListener(
216             new MouseAdapter()
217             {
218                 public void mouseReleased(MouseEvent e)
219                 {
220                     setCurrentSquare(Square.this); // Ustaw aktualne pole
221
222                     // Ustaw położenie tego pola
223                     sendClickedSquare(getSquareLocation());

```

Rysunek 26.13. Część kliencka gry w kółko i krzyżyk — ciąg dalszy

```

224         }
225     }
226 );
227 }
228
229 // Zwróć preferowany rozmiar pola
230 public Dimension getPreferredSize()
231 {
232     return new Dimension(30, 30); // Zwróć preferowany rozmiar
233 }
234
235 // Zwróć minimalny rozmiar pola
236 public Dimension getMinimumSize()
237 {
238     return getPreferredSize(); // Zwróć preferowany rozmiar
239 }
240
241 // Ustaw znak w polu
242 public void setMark(String newMark)
243 {
244     mark = newMark; // Ustaw znak w polu
245     repaint(); // Przerysuj pole
246 }
247
248 // Zwróć położenie pola
249 public int getSquareLocation()
250 {
251     return location; // Zwróć położenie pola
252 }
253
254 // Rysuj pole
255 public void paintComponent(Graphics g)
256 {
257     super.paintComponent(g);
258
259     g.drawRect(0, 0, 29, 29); // Rysuj pole
260     g.drawString(mark, 11, 20); // Rysuj znak
261 }
262 }
263 }

```

Rysunek 26.13. Część klienta gry w kółko i krzyżyk — ciąg dalszy

```

1 // Rysunek 26.14. TicTacToeClientTest.java
2 // Klasa testująca klient gry w kółko i krzyżyk
3 import javax.swing.JFrame;
4
5 public class TicTacToeClientTest
6 {
7     public static void main(String[] args)
8     {
9         TicTacToeClient application; // Deklaracja aplikacji klienckiej
10
11         // Jeśli nie podano argumentów w wierszu polecenia

```

Rysunek 26.14. Klasa testująca klient gry w kółko i krzyżyk

```

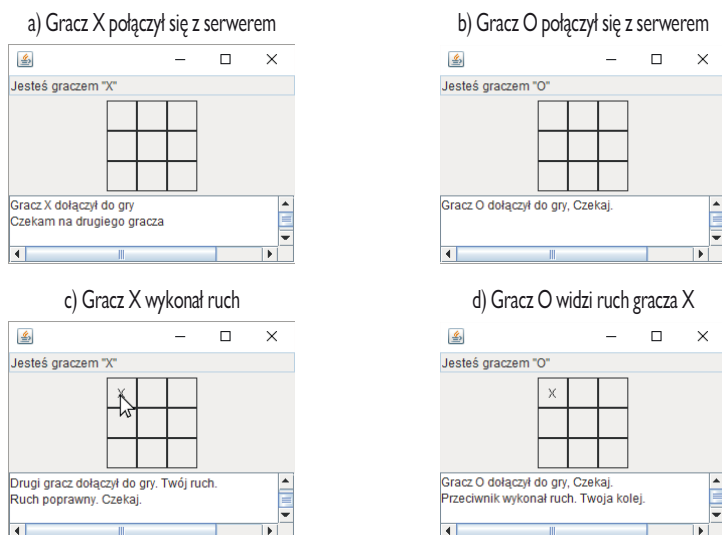
12     if (args.length == 0)
13         application = new TicTacToeClient("127.0.0.1"); // Komputer lokalny
14     else
15         application = new TicTacToeClient(args[0]); // Użycie wartości
                                                    ↗ z argumentu
16
17     application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18 }
19 }

```

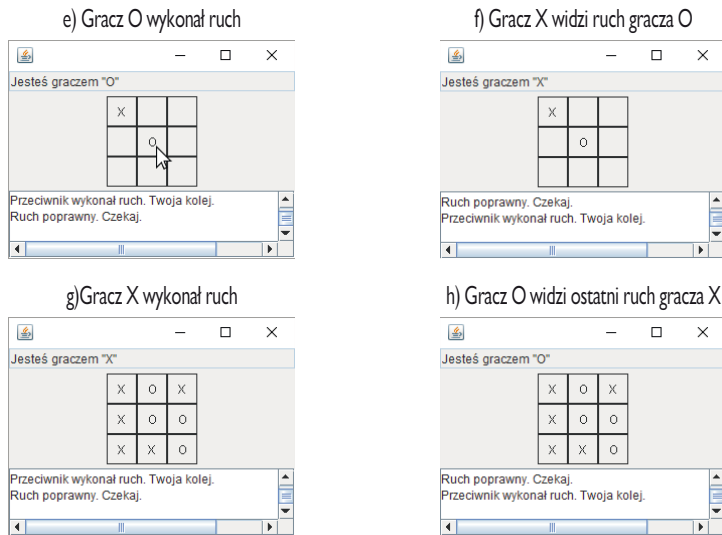
Rysunek 26.14. Klasa testująca klient gry w kółko i krzyżyk

do 125.), odczytuje komunikaty z serwera (wiersz 124.). Każdy komunikat jest przesyłany do metody `processMessage` (wiersze od 129. do 156.) w celu przetworzenia.

Jeśli otrzymany komunikat to Ruch poprawny., wiersze 134. i 135. wyświetlają komunikat Ruch poprawny. Czekaj. i wywołują metodę `setMark` (wiersze od 173. do 184.), aby ustawić znak klienta w aktualnym polu (tym, które użytkownik kliknął), ale czynią to poprzez metodę `invokeLater` ze `SwingUtilities`, aby aktualizacje interfejsu użytkownika zaszły w wątku obsługi zdarzeń. Jeśli otrzymany komunikat to Ruch niepoprawny. Spróbuj ponownie., wiersz 139. wyświetla komunikat, aby użytkownik mógł kliknąć inne pole. Jeśli komunikat to Przeciwnik wykonał ruch., wiersz 144. odczytuje liczbę z serwera wskazującą na miejsce umieszczenia znaku przez przeciwnika, a wiersze 149. i 150. umieszczają ten znak na tablicy (także tutaj czynią to poprzez metodę `invokeLater` ze `SwingUtilities`, aby aktualizacje interfejsu użytkownika zaszły w wątku obsługi zdarzeń). Jeśli pojawi się jakikolwiek inny komunikat, wiersz 155. po prostu go wyświetla.



Rysunek 26.15. Przykładowe zrzuty ekranu z przebiegu gry w kółko i krzyżyk



Rysunek 26.15. Przykładowe zrzuty ekranu z przebiegu gry w kółko i krzyżyk

26.8. Opcjonalne studium przypadku — aplikacja DeitelMessenger¹

To studium przypadku jest dostępne na stronie <http://www.deitel.com/books/jhnp11>. Pokoje rozmów (chat) zapewniają miejsce, w którym użytkownicy mogą się ze sobą komunikować za pomocą krótkich wiadomości tekstowych. Każda osoba widzi wszystkie wiadomości wysyłane przez innych, a także sama może wysyłać wiadomości. To studium przypadku zawiera wiele elementów dotyczących obsługi sieci, wielowątkowości i frameworku Swing. Aplikacja wykorzystuje też rozgłaszanie, co pozwala wysyłać pakiety DatagramPacket do grup klientów.

Studium przypadku DeitelMessenger to pokaznych rozmiarów aplikacja wykorzystująca wiele różnorodnych funkcjonalności Javy, w tym klasy Socket, DatagramPacket i MulticastSocket, wielowątkowość i interfejs graficzny Swing. Studium pokazuje też dobre praktyki programistyczne, oddzielając interfejs od implementacji, co umożliwi programistom obsługę różnych protokołów sieciowych lub innych rodzajów interfejsu. Po analizie tego studium przypadku z pewnością będziesz w stanie tworzyć bardziej rozbudowane aplikacje sieciowe.

26.9. Podsumowanie

W tym rozdziale omówiliśmy podstawy programowania sieciowego w Javie. Wykorzystaliśmy dwa różne sposoby przesyłania danych poprzez sieć, a mianowicie komunikację strumieniową za pomocą protokołu TCP/IP oraz komunikację datagramową za pomocą protokołu UDP. Pokazaliśmy, jak zbudować prosty czat typu klient – serwer wykorzystujący zarówno komunikację strumieniową, jak

¹ Wspomniane studium przypadku pochodzi z jednego z poprzednich wydań książki i jest dostarczane bez dodatkowego wsparcia ze strony autorów.

i datagramową. Następnie napisaliśmy sieciową grę w kółko i krzyżyk, która pozwala dwóm klientom na wspólną grę poprzez wielowątkowy serwer przechowujący stan gry i dodatkową logikę gry. Z następnego rozdziału dowiesz się, jak tworzyć aplikacje internetowe wykorzystujące JavaServer Faces.

Streszczenie

Podrozdział 26.1. Wprowadzenie

- Java umożliwia tworzenie gniazd strumieniowych i datagramowych. W przypadku gniazd strumieniowych proces nawiązuje stałe połączenie z innym procesem. Gdy trwa połączenie, przepływ danych między procesami odbywa się za pomocą strumieni. Gniazda strumieniowe zapewniają usługę połączeniową. Protokołem używanym w tej komunikacji jest popularny TCP (ang. *Transmission Control Protocol*).
- W przypadku gniazd datagramowych przesyła się pojedyncze pakiety informacji. Protokół UDP (ang. *User Datagram Protocol*) to bezpołączeniowa usługa, która nie gwarantuje, że pakiet nie zostanie utracony lub powielony albo że dotrze do adresata w kolejności wysyłania.

Podrozdział 26.2. Odczyt pliku z serwera WWW

- Metoda `setPage` komponentu `JEditorPane` powoduje pobranie i wyświetlenie dokumentu wskazanego jako argument.
- Najczęściej dokument HTML zawiera hiperłącza, które łączą się z innymi dokumentami na stronach WWW. Jeśli dokument HTML jest wyświetlany w komponencie `JEditorPane` z wyłączoną edycją i użytkownik kliknie łącze, dojdzie do zdarzenia `HyperlinkEvent` zgłaszanego obiektom `HyperlinkListener`.
- Metoda `getEventType` z `HyperlinkEvent` określa typ zdarzenia. Klasa `HyperlinkEvent` zawiera zagnieżdżoną klasę `EventType`, która deklaruje trzy typy zdarzeń: `ACTIVATED`, `ENTERED` i `EXITED`. Metoda `getURL` z `HyperlinkEvent` pobiera adres URL reprezentowany przez hiperłącze.

Podrozdział 26.3. Wykonanie prostego serwera przy użyciu gniazd strumieniowych

- Za połączenia strumieniowe odpowiadają obiekty `Socket`.
- Obiekt `ServerSocket` pozwala określić port, na którym serwer będzie oczekiwał połączeń od klientów. Metoda `accept` obiektu oczekuje w nieskończoność na połączenia od klienta i zwraca obiekt `Socket`, jeśli dojdzie do nawiązania połączenia.
- Metody `getOutputStream` i `getInputStream` obiektu `Socket` zwracają referencje odpowiednio do obiektu `OutputStream` i do obiektu `InputStream`. Metoda `close` kończy połączenie.

Podrozdział 26.4. Wykonanie prostego klienta przy użyciu gniazd strumieniowych

- Nazwa serwera i numer portu służą do utworzenia obiektu `Socket`, który pozwoli klientowi połączyć się z serwerem. Nieudana próba połączenia powoduje zgłoszenie wyjątku `IOException`.

- Metoda `getByName` klasy `InetAddress` zwraca obiekt `InetAddress` zawierający adres IP wskazanego komputera. Metoda `getLocalHost` klasy `InetAddress` zwraca obiekt `InetAddress` zawierający adres IP lokalnego komputera.

Podrozdział 26.6. Datagramy — bezpołączeniowa interakcja między klientem i serwerem

- Transmisja połączeniowa przypomina w działaniu system telefoniczny: dzwonic, a po drugiej stronie odbiera osoba, z którą chcemy rozmawiać. Połączenie na czas tej rozmowy jest trwałe, nawet w sytuacji, gdy przez pewien czas nikt nic nie mówi.
- Transmisja bezpołączeniowa z użyciem datagramów przypomina bardziej komunikowanie się listami za pośrednictwem poczty. Jeśli dłuższa wiadomość nie zmieści się do jednej koperty, dzieli się ją na mniejsze i umieszcza w odpowiednio ponumerowanych kopertach. Wszystkie listy wysyłamy jednocześnie. Listy mogą przyjść we właściwej kolejności, w innej kolejności lub w ogóle nie dotrzeć do adresata.
- Obiekty `DatagramPacket` przechowują pakiety danych, które mają być wysłane lub są otrzymywane przez aplikację. Obiekty `DatagramSocket` wysyłają i odbierają obiekty `DatagramPacket`.
- Konstruktor bezargumentowy `DatagramSocket` dowiazuje obiekt do portu wybranego przez komputer wykonujący program. Konstruktor przyjmujący numer portu jako argument będący liczbą całkowitą dowiazuje obiekt `DatagramSocket` do wskazanego portu. Jeśli konstruktorowi nie uda się dowiązać do portu, dochodzi do wyjątku `SocketException`. Metoda `receive` z `DatagramSocket` blokuje wątek (czeka) do momentu otrzymania pakietu, a następnie umieszcza pakiet w przekazanym argumencie.
- Metoda `getAddress` z `DatagramPacket` zwraca obiekt `InetAddress` zawierający informacje na temat komputera, z którego lub do którego jest wysyłany pakiet. Metoda `getPort` zwraca liczbę całkowitą będącą numerem portu, przez który wysyłano lub pobierano pakiet. Metoda `getLength` zwraca liczbę bajtów danych znajdujących się w `DatagramPacket`. Metoda `getData` zwraca tablicę bajtów zawierającą dane.
- Konstruktor `DatagramPacket` w przypadku wysyłania pakietu przyjmuje cztery argumenty: tablicę bajtów do wysłania, liczbę bajtów do wysłania, adres klienta, do którego będzie wysyłany pakiet, i numer portu, na którym klient czeka na otrzymanie pakietu.
- Metoda `send` z `DatagramSocket` wysyła datagram poprzez sieć.
- Jeśli w trakcie odbioru lub wysyłania pakietu pojawi się błąd, zostanie zgłoszony wyjątek `IOException`.

Ćwiczenia do samooceny

26.1. Wypełnij puste miejsca w następujących zdaniach:

- Wyjątek _____ ma miejsce, jeśli pojawi się błąd wejścia – wyjścia w momencie zamykania gniazda sieciowego.
- Wyjątek _____ zostanie zgłoszony, jeśli podanej nazwy hosta nie uda się zamienić na adres IP.
- Jeśli konstruktorowi `DatagramSocket` nie uda się poprawnie utworzyć obiektu, zgłosi wyjątek _____.
- Wiele spośród klas Javy dotyczących obsługi sieci znajduje się w pakiecie _____.
- Klasa _____ dowiazuje aplikację do portu w celu umożliwienia komunikacji datagramowej.
- Obiekt klasy _____ zawiera adres IP.
- Dwa rodzaje gniazd omawiane w tym rozdziale to _____ i _____.

- h) Metoda `getLocalHost` zwraca obiekt _____ zawierający lokalny adres IP komputera, na którym jest wykonywany program.
 - i) Konstruktor `URL` sprawdza, czy przekazany argument tekstowy jest poprawnym adresem `URL`. Jeśli tak, inicjalizuje obiekt. Jeśli nie, zgłasza wyjątek _____.
- 26.2.** Określ, czy poniższe zdania są **prawdziwe** czy **falsywe**. Jeśli są **falsywe**, wyjaśnij dlaczego.
- a) UDP to protokół wykorzystujący trwałe połączenia.
 - b) Dzięki gniazdom strumieniowym proces ustanawia połączenie z innym procesem.
 - c) Serwer czeka na porcie na połączenie od klienta.
 - d) Transmisja datagramowa poprzez sieć jest pewna — gwarantuje otrzymanie pakietów przez odbiorcę we właściwej kolejności.

Odpowiedzi do samooceny

- 26.1.** a) `IOException`, `UnknownHostException`; c) `SocketException`; d) `java.net`; e) `DatagramSocket`; f) `InetAddress`; g) gniazda strumieniowe, gniazda datagramowe; h) `InetAddress`; i) `MalformedURLException`.
- 26.2.**
- a) Falsz. UDP jest protokołem bezpołączeniowym. Trwałe połączenia to protokół TCP.
 - b) Prawda.
 - c) Prawda.
 - d) Falsz. Pakiety mogą zostać utracone lub zduplikowane albo przyjść poza kolejnością.

Ćwiczenia

- 26.3.** Podaj różnice między usługami połączeniowymi i bezpołączeniowymi.
- 26.4.** W jaki sposób klient określa nazwę hosta drugiego komputera?
- 26.5.** W jakich okolicznościach może zostać zgłoszony wyjątek `SocketException`?
- 26.6.** W jaki sposób klient może pobrać wiersz tekstu z serwera?
- 26.7.** Opisz, w jaki sposób klient łączy się z serwerem.
- 26.8.** Opisz, w jaki sposób serwer wysyła dane do klienta.
- 26.9.** Opisz, jak przygotować serwer na otrzymywanie połączeń strumieniowych od jednego klienta.
- 26.10.** W jaki sposób serwer nasłuchuje połączeń strumieniowych na konkretnym porcie?
- 26.11.** Co określa, ile połączeń od klientów może czekać w kolejce na połączenie z serwerem?
- 26.12.** Podaj, co wskazano w tekście jako możliwe powody odmówienia przez serwer połączenia z klientem.
- 26.13.** Użyj gniazda strumieniowego, które umożliwi klientowi wskazanie nazwy pliku tekstowego. Serwer ma przesłać do klienta zawartość pliku lub informację, że plik nie istnieje.
- 26.14.** Zmodyfikuj ćwiczenie 26.13 w taki sposób, aby klient mógł zmodyfikować plik i przesłać jego zawartość z powrotem na serwer. Użytkownik dokonuje edycji pliku w komponencie `JTextArea` i klika przycisk *Zapisz zmiany*, aby przesłać plik na serwer.
- 26.15. (Serwer wielowątkowy)** Serwery wielowątkowe są obecnie bardzo popularne, w szczególności po pojawieniu się serwerów wielordzeniowych. Zmodyfikuj aplikację prostego serwera z podrzdziału 26.5 tak, aby była serwerem wielowątkowym. Następnie użyj kilku aplikacji klienckich tak, aby wszystkie jednocześnie połączyły się z serwerem. Użyj klasy `ArrayList` do przechowywania wątków klienckich.

Klasa ta zapewnia kilka metod pomocnych w tym ćwiczeniu. Metoda `size` zwraca informację o liczbie przechowywanych elementów. Metoda `get` zwraca element znajdujący się na konkretnej pozycji. Metoda `add` umieszcza argument na końcu obiektu `ArrayList`. Metoda `remove` usuwa przekazany argument z listy.

- 26.16. (Gra w warcaby)** W tym rozdziale przedstawiliśmy program do gry w kółko i krzyżyk wykorzystujący serwer wielowątkowy. Napisz na tej podstawie program do gry w warcaby. Użytkownicy części klienckiej będą wykonywali swoje ruchy naprzemiennie. Część serwerowa będzie pośredniczyć w komunikacji, decydować, czyj jest teraz ruch, a także sprawdzać, czy przekazany ruch jest poprawny. Gracze sami będą decydować, kiedy ma nastąpić koniec gry.
- 26.17. (Gra w szachy)** Napisz program do gry w szachy oparty na ćwiczeniu 26.16.
- 26.18. (Gra w oczko)** Napisz program do gry w oczko, w którym serwer rozdaje karty każdemu klientowi. Serwer powinien na żądanie przekazać graczowi dodatkowe karty (zgodnie z zasadami gry).
- 26.19. (Gra w pokera)** Wykonaj program do gry w pokera, w którym serwer rozdaje karty każdemu z graczy. Serwer powinien na żądanie przekazać graczowi dodatkowe karty (zgodnie z zasadami gry).
- 26.20. (Zmiany w wielowątkowej grze w kółko i krzyżyk)** Programy z rysunków 26.11 i 26.13 implementują wielowątkową grę w kółko i krzyżyk w wersji klient – serwer. Naszym celem było pokazanie zasad tworzenia serwera wielowątkowego, który potrafi obsłużyć jednocześnie połączenia od wielu klientów. Serwer w przykładzie jest w zasadzie jedynie pośrednikiem między dwoma klientami — upewnia się, że ruch jest poprawny, a gracze wykonują swoje ruchy naprzemiennie. Serwer nie sprawdza, kto wygrał lub czy jest remis. Nie można też zacząć nowej gry lub przerwać istniejącej.

Oto lista kilku proponowanych modyfikacji kodu z rysunków 26.11 i 26.13:

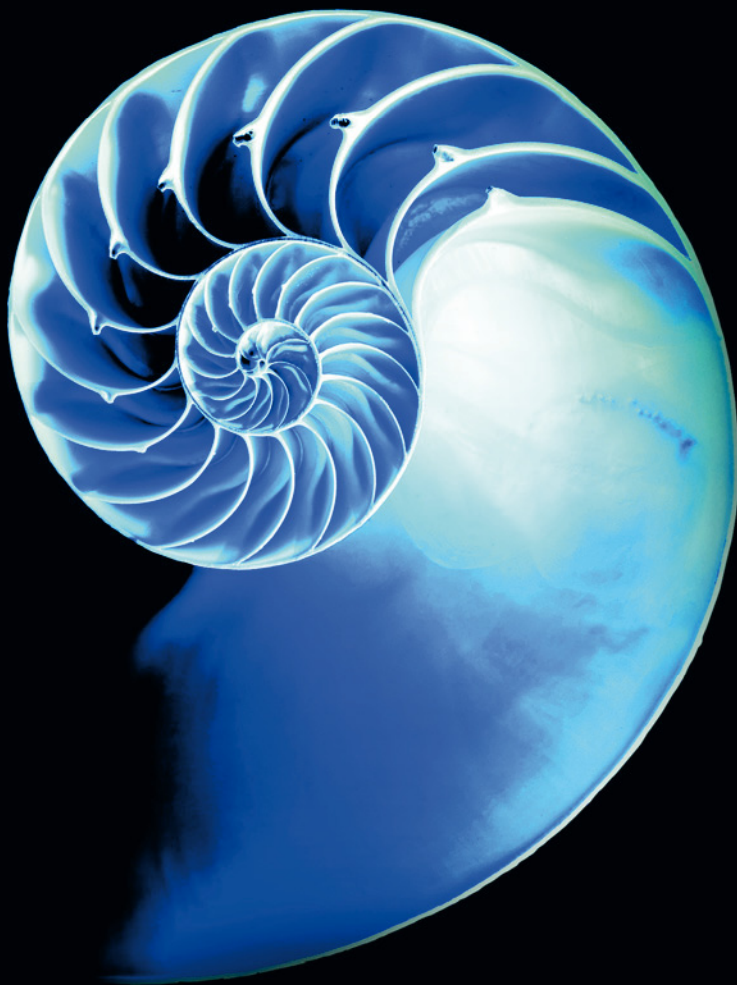
- Zmodyfikuj klasę `TicTacToeServer` tak, aby sprawdzała po każdym ruchu wygraną, przegraną lub remis. Wyślij do każdego klienta komunikat wskazujący zakończenie gry i jej wynik.
 - Zmodyfikuj klasę `TicTacToeClient`, dodając przycisk pozwalający graczowi rozpocząć nową grę. Przycisk powinien stać się aktywny dopiero po zakończeniu rozgrywki. Wymaga to zmian w klasach `TicTacToeServer` i `TicTacToeClient` związanych z wyczyszczeniem tablicy i informacji o stanie gry. Dodatkowo drugi klient powinien zostać poinformowany o rozpoczynaniu nowej gry i konieczności resetu.
 - Zmodyfikuj klasę `TicTacToeClient`, dodając przycisk pozwalający graczowi przerwać grę w dowolnym momencie. Po kliknięciu należy powiadomić serwer i drugiego klienta. Serwer powinien czekać na połączenie od innego klienta, aby zacząć nową grę.
 - Zmodyfikuj klasy `TicTacToeServer` i `TicTacToeClient`, aby wygrywający grę mógł wybrać, czy w następnej rozgrywce będzie używał znaku X czy O. Przypomnijmy, że X zawsze wykonuje pierwszy ruch.
 - Jeśli jesteś ambitny, umożliw klientowi grę przeciwko serwerowi w trakcie oczekiwania na połączenie z drugim klientem.
- 26.21. (Trójwymiarowa gra w kółko i krzyżyk)** Zmodyfikuj wielowątkową wersję gry w kółko i krzyżyk tak, aby zaimplementować trójwymiarową wersję gry na planszy 4 na 4 na 4. Zaimplementuj serwer będący pośrednikiem między dwoma graczami. Wyświetl trójwymiarową planszę jako cztery tablice o rozmiarach 4 na 4. Jeśli jesteś ambitny, wprowadź następujące modyfikacje:
- Narysuj planszę w sposób trójwymiarowy.
 - Pozwól serwerowi sprawdzać wygraną, porażkę lub remis. Uwaga! Istnieje wiele możliwych sposobów na wygranę gry na planszy 4 na 4 na 4.

26.22. (Sieciowy kod Morse’a) Chyba najbardziej popularnym sposobem kodowania jest kod Morse’a, wymyślony przez Samuela Morse’a w 1832 roku na potrzeby telegrafu. Kod przypisuje ciągi kropek i kresek poszczególnym literom alfabetu, cyfrom i kilku znakom specjalnym (np. kropce, przecinkowi, średnikowi i dwukropkowi). W systemach opartych na dźwięku kropka odpowiada krótkiemu dźwiękowi, a kreska długiemu. Inne rodzaje reprezentacji kropek i kresek stosowane są w sygnalizacji świetlnej lub flagowej. Poszczególne słowa rozdziela się znakiem spacji lub po prostu brakiem kresek i kropek. W systemie dźwiękowym spacja to krótka przerwa bez transmisji jakiegokolwiek dźwięku. Międzynarodowy kod Morse’a przedstawiony jest na rysunku 26.16.

Znak	Kod	Znak	Kod	Znak	Kod	Znak	Kod
A	.-	J	.---	S	...	1	.-.-.-
B	-...	K	-.-	T	-	2	..---
C	-.-.	L	.-..	U	..-	3	...--
D	-..	M	--	V	...-	4-
E	.	N	-.	W	.-	5
F	O	---	X	-.--	6	-....
G	--.	P	.-.-	Y	-.--	7	--...
H	Q	---.	Z	--..	8	----.
I	..	R	.-.			9	----.
						0	-----

Rysunek 26.16. Litery i cyfry w międzynarodowym kodzie Morse’a

Napisz aplikację typu klient – serwer, w której dwa klienty wysyłają do siebie wiadomości w kodzie Morse’a, używając serwera jako pośrednika. Aplikacja kliencka pozwala wpisać tekst w języku polskim w komponencie JTextArea. W momencie wysyłki program zamienia tekst na kod Morse’a i wysyła go poprzez serwer do drugiego klienta. Użyj jednej spacji do rozdzielania liter i trzech spacji do rozdzielania wyrazów. Gdy zakodowany tekst dotrze do drugiego klienta, powinien zostać rozkodowany i wyświetlony jako zwykły tekst i tekst w kodzie Morse’a. Klient powinien mieć jedno pole JTextField do wpisywania wiadomości i jeden komponent JTextArea do wyświetlania otrzymanych wiadomości.



Cele

W tym rozdziale:

- Powody powstania modułowości
- Materiały JEP i JSR związane z modułowością
- Tworzenie deklaracji modułu określających zależności modułu za pomocą `requires` i wskazujących pakiety udostępniane innym modułom za pomocą `exports`
- Umożliwienie działania mechanizmu refleksji typów w trakcie działania aplikacji za pomocą konstrukcji `open` i `opens`
- Wykorzystanie usług do luźnego powiązania komponentów systemowych w celu uproszczenia tworzenia i utrzymania dużych systemów
- Zastosowanie przez moduł konstrukcji `uses` do użycia usługi i konstrukcji `provides...with` do zapewnienia usługi
- Użycie polecenia `jdeps` do określenia zależności modułu
- Migracja kodu bezmodułowego do Javy 9 za pomocą modułów nienazwanych i automatycznych
- Wykorzystanie IDE NetBeans do utworzenia grafu modułów
- Określanie zależności przez system wykonawczy dzięki mechanizmowi rozwiązywania modułów
- Użycie polecenia `link` do tworzenia mniejszych wersji wykonawczych odpowiednich dla urządzeń o ograniczonych zasobach

- 27.1.** Wprowadzenie
- 27.2.** Deklaracja modułu
 - 27.2.1. Dyrektywa `requires`
 - 27.2.2. Dyrektywa `requires transitive` — niejawne czytanie
 - 27.2.3. Dyrektywy `exports` i `exports...to`
 - 27.2.4. Dyrektywa `uses`
 - 27.2.5. Dyrektywa `provides...with`
 - 27.2.6. Modyfikator `open` oraz dyrektywy `opens` i `opens...to`
 - 27.2.7. Ograniczone słowa kluczowe
- 27.3.** Modułowa wersja aplikacji powitalnej
 - 27.3.1. Struktura aplikacji
 - 27.3.2. Klasa `Welcome`
 - 27.3.3. Plik `module-info.java`
 - 27.3.4. Graf zależności modułu
 - 27.3.5. Kompilacja modułu
 - 27.3.6. Uruchamianie aplikacji z poziomu rozbitego folderu aplikacji
 - 27.3.7. Spakowanie modułu do pliku JAR
 - 27.3.8. Uruchamianie aplikacji z modułowego pliku JAR
 - 27.3.9. Dodatkowa uwaga — ścieżka klas a ścieżka modułów
- 27.4.** Tworzenie i użycie własnych modułów
 - 27.4.1. Eksport pakietu w celu użycia w innych modułach
 - 27.4.2. Wykorzystanie klasy pakietu w innym module
 - 27.4.3. Kompilacja i uruchomienie przykładu
 - 27.4.4. Zapakowanie aplikacji do modułowych plików JAR
 - 27.4.5. Silna enkapsulacja i dostępność
- 27.5.** Graf zależności modułu — dokładniejsze spojrzenie
 - 27.5.1. Moduł `java.sql`
 - 27.5.2. Moduł `java.se`
 - 27.5.3. Wyświetlenie grafu zależności modułów JDK
 - 27.5.4. Błąd — graf modułu z cyklem
- 27.6.** Migracja kodu do Javy 9
 - 27.6.1. Moduł nienazwany
 - 27.6.2. Moduły automatyczne
 - 27.6.3. Narzędzie `jdeps` — analiza zależności
- 27.7.** Zasoby w modułach — wykorzystanie modułu automatyczne
 - 27.7.1. Moduły automatyczne
 - 27.7.2. Wymaganie kilku modułów
 - 27.7.3. Otwarcie modułu na potrzeby mechanizmu refleksji
 - 27.7.4. Graf zależności modułu
 - 27.7.5. Kompilacja modułu
 - 27.7.6. Uruchomienie aplikacji po modularyzacji
- 27.8.** Tworzenie własnych systemów wykonawczych narzędziem `jlink`
 - 27.8.1. Wyświetlenie listy modułów JRE
 - 27.8.2. Własny system wykonawczy zawierający tylko moduł `java.base`
 - 27.8.3. Tworzenie własnego systemu wykonawczego dla aplikacji powitalnej
 - 27.8.4. Wykonywanie aplikacji powitalnej we własnym systemie wykonawczym
 - 27.8.5. Użycie mechanizmu rozwiązywania modułów z własnym systemem wykonawczym
- 27.9.** Usługi i klasa `ServiceLoader`
 - 27.9.1. Interfejs dostawcy usług
 - 27.9.2. Wczytywanie i użycie dostawców usług
 - 27.9.3. Dyrektywa `uses` modułu i konsumpcja usług
 - 27.9.4. Uruchomienie aplikacji bez dostawców usług
 - 27.9.5. Implementacja dostawcy usług
 - 27.9.6. Dyrektywa `provides...with` modułu i deklaracja dostawcy usług
 - 27.9.7. Uruchomienie aplikacji z jednym dostawcą usług
 - 27.9.8. Implementacja drugiego dostawcy usług
 - 27.9.9. Uruchomienie aplikacji z dwoma dostawcami usług
- 27.10.** Podsumowanie

27.1. Wprowadzenie¹

W tym rozdziale wprowadzamy **system modułów platformy Java** (JPMS — ang. *Java Platform Module System*), czyli najważniejszą nową technologię Javy 9. Modułowość, będąca rezultatem projektu **Jigsaw**², pomoże programistom na każdym poziomie zaawansowania stać się bardziej produktywnymi w budowaniu i konserwacji systemów oraz zarządzaniu nimi, w szczególności jeśli chodzi o większe systemy. Studenci uczestniczący w zajęciach dla bardziej zaawansowanych powinni bardzo dobrze opanować system modułów, aby w ten sposób przygotować się do pracy zawodowej.

Wymagane oprogramowanie

Przed przystąpieniem do czytania tego rozdziału zainstaluj JDK 9 i pobierz przykładowe kody źródłowe opisywane po wstępie do drukowanej wersji książki. Przedstawimy kilka grafów zależności modułów wykonanych we wstępnej wersji IDE NetBeans, która zawiera obsługę JDK 9:

<http://wiki.netbeans.org/JDK9Support>

Twórcy innych IDE najprawdopodobniej zapewnią podobne narzędzia.

Czym jest moduł?

Modułowość to wyższy poziom łączenia elementów niż pakiety. Nowym, kluczowym elementem języka jest moduł — unikatowo nazwana grupa pakietów wielokrotnego stosowania, a także zasoby (takie jak obrazy lub pliki XML) i **deskryptor modułu** zawierający:

- nazwę modułu;
- zależności modułu (czyli wskazanie innych modułów, od których zależy ten moduł);
- pakiety, które **jawnie** udostępnia moduł innym modułom (domyślnie wszystkie inne pakiety są **niejawnie** niedostępne dla innych modułów);
- oferowane usługi;
- wykorzystywane usługi;
- wskazanie, jakie inne moduły mogą korzystać z **mechanizmu refleksji**.

Historia

Platforma Java SE istnieje od 1995 roku. Obecnie około 10 milionów programistów pisze różnego rodzaju aplikacje — od małych aplikacji działających na urządzeniach o ograniczonych zasobach (np. urządzenia osadzone lub stanowiące część internetu rzeczy) po ogromne aplikacje biznesowe o krytycznym znaczeniu dla firm. Istnieje ogromnie dużo starego kodu, ale do tej pory platforma Java była w dużej części systemem monolitycznym działającym na zasadzie „jeden rozmiar dla wszystkich”. Pojawiało się we wcześniejszych latach wiele prób modularyzacji Javy, ale żaden z pomysłów się nie przyjął.

¹ Chcielibyśmy podziękować następującym osobom za odpowiedzi na nasze pytania i za cenne uwagi: Brian Goetz, Alex Buckley, Alan Bateman, Lance Anderson, Mandy Chung i Paul Bakker.

² <http://openjdk.java.net/projects/jigsaw/>

Modularyzacja platformy Java SE stanowi poważne wyzwanie, więc powstanie odpowiedniej wersji zajęło wiele lat. Propozycja **JSR 277: Java Module System**³ została zgłoszona już w 2005 roku dla Javy 7. Ten JSR został później wycofany i zastąpiony przez **JSR 376: Java Platform Module System**⁴ nakierowany na użycie w Javie 8. Platforma Java SE została zmodularyzowana dopiero w Javie 9, co spowodowało opóźnienie wydania Javy 9 do września 2017 roku.

Cele

Zgodnie z JSR 376 kluczowymi celami modularyzacji platformy Java SE są:

- Trwała i pewna konfiguracja — modułowość dostarcza mechanizmy jawnego deklarowania zależności między modułami w sposób, który jest rozpoznawany zarówno na etapie kompilacji, jak i wykonywania kodu. System może przejść przez wszystkie zależności i określić podzbiór wszystkich modułów niezbędnych do obsługi aplikacji.
- Silna enkapsulacja — pakiety modułu są widoczne dla innych modułów tylko wtedy, gdy moduł jawnie je **wyeksportuje**. Nawet wówczas inny moduł nie może z nich skorzystać, dopóki sam jawnie nie wskaże, że **wymaga** innych modułów. Poprawia to bezpieczeństwo platformy, bo potencjalny atakujący ma dostęp do mniejszej liczby modułów. Mając na uwadze modułowość, najczęściej tworzy się czystsze i bardziej logiczne projekty aplikacji.
- Skalowalność — wcześniej platforma Java była systemem monolitycznym składającym się z ogromnej liczby pakietów, co powodowało, że jej tworzenie, utrzymanie i rozwijanie nie były łatwe. Nie można było w łatwy sposób tworzyć podzbiorów platformy. Obecnie platforma składa się z 95 modułów (liczba ta będzie się zmieniać wraz z rozwojem Javy). Można teraz tworzyć własne systemy wykonawcze składające się tylko z modułów, których potrzebuje aplikacja lub docelowe urządzenie. Jeśli docelowe urządzenie nie obsługuje GUI, można utworzyć system wykonawczy pozbawiony modułów związanych z GUI, co znacząco zmniejszy rozmiar wersji wykonawczej.
- Większa integralność — przed Javą 9 możliwe było wykorzystywanie wielu klas, które nie były przewidziane do stosowania przez klasy aplikacji końcowych. Dzięki silnej enkapsulacji te wewnętrzne API teraz są faktycznie zamknięte hermetycznie i ukryte przez aplikacjami wykorzystującymi platformę. Wadą tego rozwiązania jest trudniejsza migracja starszego kodu do wersji działającej w Javie 9.
- Poprawiona wydajność — maszyna wirtualna Javy korzysta z różnych optymalizacji w celu poprawy wydajności aplikacji. JSR 376⁵ wskazuje, że techniki te są bardziej efektywne, jeśli z wyprzedzeniem wiadomo, które moduły będą wykorzystywane przez aplikację.

³ <https://jcp.org/en/jsr/detail?id=277>

⁴ <https://jcp.org/en/jsr/detail?id=376>

⁵ Mark Reinhold, „JSR 376: Java Platform Module System”, <https://jcp.org/en/jsr/detail?id=376>

Wyświetlenie listy modułów JDK

Niezwyczajnie ważnym aspektem Javy 9 jest podział JDK na moduły, które obsługują różne konfiguracje (JEP 200⁶). Użycie polecenia `java` z JDK z poziomu folderu `bin` z opcją `--list-modules`:

```
java --list-modules
```

spowoduje wyświetlenie listy modułów JDK (rysunek 27.1), która zawiera **moduły standardowe** implementujące specyfikację Javy SE (nazwy zaczynają się od `java`), moduły JavaFX (nazwy zaczynają się od `javafx`), moduły specyficzne dla JDK (nazwy zaczynają się od `jdk`) i moduły specyficzne dla Oracle (nazwy zaczynają się od `oracle`). Nazwa modułu zawiera **informację o wersji**. Ponieważ wykorzystywaliśmy wersję JDK 9 o wczesnym dostępie, każdy moduł zawiera informację o wersji w postaci `"@9-ea"`, wskazującą na wersję o wczesnym dostępie (`ea`). W finalnym wydaniu Javy 9 zniknie przyrostek `"-ea"`.

Dokumenty JEP i JSR dotyczące modułowości Javy

We wstępie omówiliśmy, czym są dokumenty JEP i JSR. Dokumenty JEP i JSR dotyczące modułowości Javy przedstawia rysunek 27.2. Będziemy odnosić się do nich w treści tego rozdziału.

Krótkie podsumowanie rozdziału

W tym rozdziale przedstawimy najbardziej podstawowe pojęcia związane z modułowością, z którymi zetknie się każda osoba tworząca duże systemy. Oto kluczowe tematy poruszane w tym rozdziale:

- Deklaracje modułów — utworzymy deklaracje modułów, które zawierają zależności między modułami (dyrektywa `requires`), pakiety modułu udostępniane innym modułom (dyrektywa `exports`), oferowane usługi (dyrektywa `provides...with`), konsumowane usługi (dyrektywa `uses`), a także dostęp do mechanizmu refleksji (modyfikator `open` oraz dyrektywy `opens` i `opens...to`).
- Graf zależności modułu — wykorzystamy IDE NetBeans wspierającą JDK 9, aby utworzyć graf modułów pozwalający zwizualizować zależności między modułami.
- Rozwiązywanie modułów — pokażemy kroki, które wykonuje mechanizm rozwiązywania modułów, aby zapewnić spełnienie wszystkich zależności.
- Narzędzie `jlink` (linker Javy) — omówimy nowe narzędzie JDK 9, które pozwala tworzyć mniejsze systemy wykonawcze, a następnie używać ich do wykonywania aplikacji. W praktyce większość aplikacji wiersza poleceń prezentowanych w tej książce można uruchomić w systemie wykonawczym składającym się tylko z najbardziej podstawowego modułu JDK — `java.base` — zawierającego takie pakiety jak `java.lang`, `java.io` i `java.util`. Jak się wkrótce przekonasz, wszystkie moduły **niejawnie** zależą od modułu `java.base`.

⁶ Mark Reinhold, „JEP 200: The Modular JDK”, <http://openjdk.java.net/jeps/200>

java.activation@9-ea	jdk.httpserver@9-ea
java.base@9-ea	jdk.incubator.httpclient@9-ea
java.compiler@9-ea	jdk.internal.ed@9-ea
java.corba@9-ea	jdk.internal.jvmstat@9-ea
java.datatransfer@9-ea	jdk.internal.le@9-ea
java.desktop@9-ea	jdk.internal.opt@9-ea
java.instrument@9-ea	jdk.internal.vm.ci@9-ea
java.jnlp@9-ea	jdk.jartool@9-ea
java.logging@9-ea	jdk.javadoc@9-ea
java.management@9-ea	jdk.javaws@9-ea
java.management.rmi@9-ea	jdk.jcmd@9-ea
java.naming@9-ea	jdk.jconsole@9-ea
java.prefs@9-ea	jdk.jdeps@9-ea
java.rmi@9-ea	jdk.jdi@9-ea
java.scripting@9-ea	jdk.jdwp.agent@9-ea
java.se@9-ea	jdk.jfr@9-ea
java.se.ee@9-ea	jdk.jlink@9-ea
java.security.jgss@9-ea	jdk.jshell@9-ea
java.security.sasl@9-ea	jdk.jsobject@9-ea
java.smartcardio@9-ea	jdk.jstatd@9-ea
java.sql@9-ea	jdk.localedata@9-ea
java.sql.rowset@9-ea	jdk.management@9-ea
java.transaction@9-ea	jdk.management.agent@9-ea
java.xml@9-ea	jdk.naming.dns@9-ea
java.xml.bind@9-ea	jdk.naming.rmi@9-ea
java.xml.crypto@9-ea	jdk.net@9-ea
java.xml.ws@9-ea	jdk.pack@9-ea
java.xml.ws.annotation@9-ea	jdk.packager@9-ea
javafx.base@9-ea	jdk.packager.services@9-ea
javafx.controls@9-ea	jdk.plugin@9-ea
javafx.deploy@9-ea	jdk.plugin.dom@9-ea
javafx.fxml@9-ea	jdk.plugin.server@9-ea
javafx.graphics@9-ea	jdk.policytool@9-ea
javafx.media@9-ea	jdk.rmic@9-ea
javafx.swing@9-ea	jdk.scripting.nashorn@9-ea
javafx.web@9-ea	jdk.scripting.nashorn.shell@9-ea
jdk.accessibility@9-ea	jdk.sctp@9-ea
jdk.attach@9-ea	jdk.security.auth@9-ea
jdk.charsets@9-ea	jdk.security.jgss@9-ea
jdk.compiler@9-ea	jdk.snmp@9-ea
jdk.crypto.cryptoki@9-ea	jdk.unsupported@9-ea
jdk.crypto.ec@9-ea	jdk.xml.bind@9-ea
jdk.crypto.mscapi@9-ea	jdk.xml.dom@9-ea
jdk.deploy@9-ea	jdk.xml.ws@9-ea
jdk.deploy.controlpanel@9-ea	jdk.zipfs@9-ea
jdk.dynalink@9-ea	oracle.desktop@9-ea
jdk.editpad@9-ea	oracle.net@9-ea
jdk.hotspot.agent@9-ea	

Rysunek 27.1. Wykonanie polecenia `java --list-modules` powoduje wyświetlenie nazw 95 modułów JDK

- Mechanizm refleksji — **refleksja** to mechanizm umożliwiający programom Javy dynamiczne wczytywanie typów, a następnie tworzenie i używanie obiektów tych typów⁷. Można korzystać z tej funkcjonalności pomimo silnej enkapsulacji wprowadzanej przez Javę 9, ale tylko w odniesieniu

⁷ Strona „Trail: The Reflection API” działu *The Java Tutorials*, <https://docs.oracle.com/javase/tutorial/reflect/>.

Dokumenty JEP i JSR dotyczące modułowości Javy

„JEP 200: The Modular JDK” (<http://openjdk.java.net/jeps/200>)
 „JEP 201: Modular Source Code” (<http://openjdk.java.net/jeps/201>)
 „JEP 220: Modular Run-Time Images” (<http://openjdk.java.net/jeps/220>)
 „JEP 260: Encapsulate Most Internal APIs” (<http://openjdk.java.net/jeps/260>)
 „JEP 261: Module System” (<http://openjdk.java.net/jeps/261>)
 „JEP 275: Modular Java Application Packaging” (<http://openjdk.java.net/jeps/275>)
 „JEP 282: jlink: The Java Linker” (<http://openjdk.java.net/jeps/282>)
 „JSR 376: Java Platform Module System” (<https://www.jcp.org/en/jsr/detail?id=376>)
 „JSR 379: Java SE 9” (<https://www.jcp.org/en/jsr/detail?id=379>)

Rysunek 27.2. Dokumenty JEP i JSR dotyczące modułowości Javy

do modułów, które **jawnie** na to pozwalają. Pokażemy, w jaki sposób wskazać, że moduł dopuszcza mechanizm refleksji do działania, stosując modyfikator `open` oraz dyrektywy `opens` i `opens...to`.

- Migracje — platforma Java jest używana od ponad 20 lat, więc ogromne ilości niemodułowego, starszego kodu trzeba zmigrować do modułowego świata Javy 9. Choć istnieją pewne pułapki związane z silną enkapsulacją Javy 9, pokażemy, że nienazwane lub automatyczne moduły mogą znacząco ułatwić proces migracji. Wykorzystamy narzędzie `jdeps`, aby określić zależności między modułami, a także wskazać użycia wewnętrznych API istniejących przed Javą 9 (większość tych API jest obecnie ukryta dzięki enkapsulacji). Większość kodu sprzed Javy 9 będzie działała bez modyfikacji, ale mogą pojawić się pewne problemy, opisywane dokładniej w podrozdziale 27.6.
- Usługi i dostawcy usług — gdy tworzy się duże systemy spełniające istotne potrzeby, mogą one istnieć i pozostawać w użyciu przez wiele dekad. W trakcie tego okresu zmiana to raczej reguła niż wyjątek. W podrozdziale 10.13 opisaliśmy **powiązanie ścisłe** i **powiązanie luźne**. Dowiedziono, że **ściśle powiązanie** utrudnia modyfikację systemów. Pokażemy więc, jak wykonać luźno powiązane komponenty systemu, używając interfejsów i implementacji dostawców usług oraz klasy `ServiceLoader`. Pokażemy również, jak stosować dyrektywy `uses` i `provides...with` w deklaracjach modułów, aby wskazać, że moduł używa usługi lub dostarcza jej implementację.

Wymienione elementy zilustrujemy kilkoma obszerniejszymi przykładami kodu z sensownymi wynikami, kilkoma krótkimi fragmentami kodu, grafami modułów wykonanymi dzięki widokowi Graph z IDE NetBeans, a także przykładami nowych poleceń (np. `jlink`) lub dodatkowymi opcjami istniejących poleceń, takich jak `javac`, `java` i `jar`. Dodatkowe źródła informacji bogate w przykłady to:

- strona *Project Jigsaw: Module System Quick-Start Guide* dostępna pod adresem <http://openjdk.java.net/projects/jigsaw/quick-start>;
- książka Maka Sandera i Paula Bakker’a *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications* wydana w 2017 roku przez wydawnictwo O’Reilly Media.

Uwaga dotycząca terminologii

System wykonawczy Javy (nazywany w skrócie JRE) zawiera maszynę wirtualną Javy i inne oprogramowanie związane z uruchamianiem programów Javy. Od Javy 9 JRE stanowi poprawny podzbiór JDK, które składa się z wszystkich API Javy i narzędzi związanych zarówno z tworzeniem, jak i wykonywaniem programów Javy. W tym rozdziale terminy „platforma Java” i „platforma Java SE” są synonimami JDK.

27.2. Deklaracja modułu

Jak wspomnieliśmy wcześniej, moduł musi zapewnić deskryptor modułu — metadane określające zależności modułu, pakiety udostępniane przez moduł innym modułom itp. Deskryptor modułu to skompilowana wersja **deklaracji modułu** zdefiniowana w pliku o nazwie *module-info.java*. Deklaracja modułu zaczyna się od słowa kluczowego **module**, po której następuje unikatowa **nazwa modułu** i **treść modułu** zawarta w nawiasach klamrowych:

```
module nazwaModułu {
}
```

Treść deklaracji modułu może być pusta lub zawierać różne **dyrektywy**, takie jak `requires`, `exports`, `provides...with`, `uses` lub `opens` (wkrótce opiszemy każdą z nich). Jak przekonasz się w punkcie 27.3.5, kompilacja deklaracji modułu tworzy deskryptor modułu, który trafia do pliku o nazwie *module-info.class* w głównym folderze modułu. Teraz pokrótce przedstawimy każdą z dyrektyw modułu. Faktyczną deklaracją modułu zajmiemy się dopiero w punkcie 27.3.3.

27.2.1. Dyrektywa `requires`

Dyrektywa **`requires`** wskazuje, że ten moduł zależy od innego modułu — związek ten nazywamy **zależnością modułu**. Każdy moduł **musi** jawnie wskazać swoje zależności. Gdy moduł A wymaga (`requires`) modułu B, mówimy, że moduł A **czyta** moduł B, czyli moduł B **jest czytany przez** moduł A. Aby określić zależność od innego modułu, użyj zapisu `requires` w postaci:

```
requires nazwaModułu;
```

Punkt 27.3.3 ilustruje dyrektywę `requires`⁸.

27.2.2. Dyrektywa `requires transitive` — niejawne czytanie

Aby wskazać zależność od innego modułu i jednocześnie pokazać, że inne moduły czytające ten moduł także powinny czytać tę zależność — czyli zapewnić im **niejawne czytanie** — zastosuj dyrektywę `requires transitive`:

```
requires transitive nazwaModułu;
```

Przyjrzyjmy się następującej dyrektywie z deklaracji modułu `java.desktop`:

```
requires transitive java.xml;
```

⁸ Istnieje również dyrektywa `requires static`, która wskazuje, że moduł jest wymagany na etapie kompilacji, ale jest **opcjonalny** na etapie działania. To tak zwana **zależność opcjonalna**, której opis wykracza poza ramy tego rozdziału.

W tym przypadku każdy moduł, który czyta `java.desktop`, dodatkowo w sposób **niejawny** czyta `java.xml`. Przykładowo jeśli metoda z modułu `java.desktop` zwróci typ z modułu `java.xml`, kod z modułów czytających `java.desktop` staje się zależny od `java.xml`. Gdyby nie deklaracja `requires transitive` w `java.desktop`, taki moduł mógłby korzystać z `java.xml` dopiero po jego **jawnym** odczytaniu.

Zgodnie z JSR 379⁹ standardowe moduły Javy SE **muszą** zapewnić niejawną czytelność we wszystkich przypadkach podobnych do tu opisywanego. Choć standardowy moduł Javy SE może zależeć od modułów niestandardowych, **nie** może zapewniać im niejawnego czytania.



27.1. Wskazówka poprawiająca przenośność kodu

Ponieważ standardowe moduły Javy SE nie mogą gwarantować niejawnego czytania niestandardowych modułów, kod zależny tylko od standardowych modułów Javy jest przenośny między różnymi implementacjami Javy SE.

27.2.3. Dyrektywy `exports` i `exports...to`

Dyrektywa `exports` określa jeden z pakietów modułu, którego publiczne typy (a także zagnieżdżone typy publiczne i chronione) powinny być dostępne dla kodu we wszystkich innych modułach. Dyrektywa `exports...to` umożliwia określenie za pomocą listy, którego modułu lub których modułów kod ma mieć dostęp do eksportowanego pakietu — jest to tak zwany **eksport kwalifikowany**. Podrozdział 27.4 ilustruje użycie dyrektywy `exports`.

27.2.4. Dyrektywa `uses`

Dyrektywa `uses` określa usługę używaną przez ten moduł — czyni to moduł **konsumentem usługi**. Usługa to obiekt klasy implementującej interfejs lub rozszerzającej klasę abstrakcyjną wskazaną w dyrektywie `uses`. Punkt 27.9.3 ilustruje użycie dyrektywy `uses`.

27.2.5. Dyrektywa `provides...with`

Dyrektywa `provides...with` określa moduł zapewniający implementację usługi — czyni to moduł **dostawcą usługi**. Część `provides` dyrektywy określa interfejs lub klasę abstrakcyjną wymienianą w dyrektywie `uses`, a część `with` określa nazwę klasy, która implementuje interfejs lub rozszerza klasę abstrakcyjną. Punkt 27.9.6 ilustruje użycie dyrektywy `provides...with`.

27.2.6. Modyfikator `open` oraz dyrektywy `opens` i `opens...to`^{10,11}

Przed Javą 9 mechanizm refleksji mógł posłużyć do tego, by poznać wszystkie typy pakietu i wszystkie składowe typu — nawet składowe prywatne — niezależnie od tego, czy taka funkcjonalność była pożądana, czy nie. Tak naprawdę nie istniała prawdziwa enkapsulacja.

⁹ Iris Clark i Mark Reinhold, „Java SE 9 (JSR 379)”, 6 marca 2017, <http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html#s7>

¹⁰ Alex Buckley, „JPMS: Modules in the Java Language and JVM”, 23 lutego 2017, <http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html>

¹¹ James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley i Dan Smith, *The Java® Language Specification Java SE 9 Edition*, punkt 7.7.2, 22 lutego 2017, <http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-02/java-se-9-jls-pr-diffs.pdf>

Jednym z kluczowych powodów powstania systemu modułów jest dążenie do zapewnienia **silnej enkapsulacji**. Domyślnie typ modułu nie jest dostępny dla innych modułów, chyba że jest to typ publiczny **oraz** dokonano jego eksportu. Uwidacznia się tylko te pakiety, które chce się uwidocznić. W przypadku Javy 9 dotyczy to również mechanizmu refleksji.

Umożliwienie dostępu do pakietu tylko w trybie wykonywania

Dyrektywa **opens** w postaci:

```
opens pakiet
```

wskazuje, że typy publiczne z *pakiet* (a także zagnieżdżone typy publiczne i chronione) są dostępne dla kodu z innych modułów tylko w trakcie działania programu. Dodatkowo wszystkie typy wskazanego pakietu (wraz ze składowymi typów) są dostępne również dla mechanizmu refleksji.

Umożliwienie dostępu do pakietu tylko w trybie wykonywania i dla wybranych modułów

Dyrektywa **opens...to** w postaci:

```
opens pakiet to lista-oddzielonych-przecinkami-modułów
```

wskazuje, że typy publiczne z *pakiet* (a także zagnieżdżone typy publiczne i chronione) są dostępne tylko dla kodu z wymienionych modułów i tylko w trakcie działania programu. Wszystkie typy wskazanego pakietu (wraz ze składowymi typów) są dostępne również dla mechanizmu refleksji działającego z poziomu wymienionych modułów.

Umożliwienie dostępu do wszystkich pakietów modułu tylko w trybie wykonywania

Jeśli wszystkie pakiety modułu powinny być dostępne tylko w trybie wykonywania lub dla mechanizmu refleksji dla wszystkich innych modułów, można **otworzyć** cały moduł konstrukcją:

```
open module nazwaModułu {  
    // Dyrektywy modułu  
}
```

Domyślne działanie mechanizmu refleksji

Domyślnie moduł z dostępem do mechanizmu refleksji dla pakietu może widzieć tylko typy publiczne pakietu (a także ich zagnieżdżone typy publiczne i chronione). Jednakże kod innego modułu **ma** dostęp do **wszystkich** typów eksponowanych przez pakiet i **wszystkich** składowych tych typów, również prywatnych. Aby dowiedzieć się więcej na temat zastosowania mechanizmu refleksji do dostępu do wszystkich składowych typów, odwiedź stronę:

<https://docs.oracle.com/javase/tutorial/reflect/>

Wstrzykiwanie zależności

Mechanizm refleksji jest często używany w połączeniu z **wstrzykiwaniem zależności**. Przykładem są tu aplikacje JavaFX używające FXML, takie jak te z rozdziałów 12., 13. i 22. Gdy aplikacja wczytuje dane FXML, obiekt kontrolera i komponenty GUI, których **używa**, powstają dynamicznie w opisany poniżej sposób:

- Ponieważ aplikacja **zależy** od obiektu kontrolera, który obsługuje interakcje GUI, FXMLLoader **wstrzykuje** obiekt kontrolera do działającej aplikacji — klasa FXMLLoader używa mechanizmu refleksji do znalezienia i wczytania klasy kontrolera do pamięci, a następnie utworzenia z niej obiektu.
- Ponieważ kontroler **zależy** od komponentów GUI zadeklarowanych w FXML, klasa FXMLLoader tworzy obiekty komponentów GUI zadeklarowane w FXML i **wstrzykuje** je do obiektu kontrolera, przypisując każdy z nich do odpowiedniej zmiennej instancji z adnotacją @FXML.

Gdy opisany proces się zakończy, kontroler może wejść w interakcję z GUI i reagować na zdarzenia. Dyrektywy `opens...` to użyjemy w punkcie 27.7.2, aby umożliwić klasie FXMLLoader użycie mechanizmu refleksji w aplikacji JavaFX z własnym modulem.

27.2.7. Ograniczone słowa kluczowe

Słowa kluczowe `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` i `with` są **ograniczonymi słowami kluczowymi**. Są to słowa kluczowe używane tylko w deklaracji modułu. Można je bez przeszkód stosować jako identyfikatory w pozostałej części kodu.

W przypisie dolnym numer 8 wspomnieliśmy również o dyrektywie `requires static`. Oczywiście `static` jest standardowym słowem kluczowym.

27.3. Modułowa wersja aplikacji powitalnej

W tym podrozdziale utworzymy prostą aplikację powitalną, aby pokazać podstawy działania modułów. Zajmiemy się:

- utworzeniem klasy znajdującej się w module;
- zapewnieniem deklaracji modułu;
- skompilowaniem deklaracji modułu i klasy `Welcome` jako modułu;
- uruchomieniem klasy zawierającej metodę `main` w tym module.

Po omówieniu tych podstaw pokażemy również:

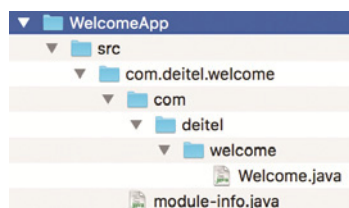
- spakowanie aplikacji `Welcome` do modułowego pliku JAR;
- uruchomienie aplikacji z poziomu pliku JAR.

27.3.1. Struktura aplikacji

Aplikacja prezentowana w tym podrozdziale składa się z dwóch plików `.java`: pliku `Welcome.java`, który zawiera klasę `Welcome`, i pliku `module-info.java`, który zawiera deklarację modułu. Zgodnie z konwencją modułowe aplikacje stosują następującą strukturę katalogów:

```
FolderAplikacji
src
  FolderZNazwqModułu
    FolderyPakietów
      PlikiŹródłoweKoduJavy
        module-info.java
```


Aplikacja powitalna zostanie zdefiniowana w pakiecie `com.deitel.welcome`. Strukturę folderów przedstawia rysunek 27.3.



Rysunek 27.3. Struktura folderów aplikacji powitalnej

Folder `src` zawiera cały kod źródłowy aplikacji. Znajduje się w nim **folder będący korzeniem** modułu, który składa się z nazwy modułu — `com.deitel.welcome` (wkrótce dokładniej przyjrzymy się nazewnictwu modułów). Główny folder modułu zawiera zagnieżdżone foldery reprezentujące strukturę pakietów — `com/deitel/welcome` — co odpowiada pakietowi `com.deitel.welcome`. Folder najbardziej zagnieżdżony zawiera plik `Welcome.java`. W głównym folderze modułu znajduje się wymagany plik deklaracji modułu o nazwie `module-info.java`.

Konwencje nazewnictwa modułów

Podobnie jak nazwy pakietów, także nazwy modułów muszą być **unikatowe**. Aby zapewnić taką unikatowość, warto rozpocząć nazwę modułu od odwróconej nazwy domeny internetowej stosowanej przez firmę. Ponieważ naszą domeną jest `deitel.com`, zaczęliśmy nazwę pakietu od `com.deitel`. Zgodnie z konwencją nazwy modułów wykorzystują podobne zasady.

Jeśli w trakcie kompilacji okaże się, że kilka modułów ma taką samą nazwę, pojawi się błąd kompilacji. Jeśli taka sama sytuacja wystąpi w trakcie działania aplikacji, system wykonawczy zgłosi wyjątek.

Przedstawiony przykład używa takiej samej nazwy dla modułu i pakietu, ponieważ w module znajduje się tylko jeden pakiet. Nie jest to wymagane, ale do podobnych sytuacji dochodzi dość często. W modułowej aplikacji Java przechowuje nazwy modułów niezależnie od nazw pakietów, więc **dopuszcza się** duplikację nazwy modułu i nazwy pakietu.

Moduły najczęściej grupują powiązane ze sobą pakiety. Oznacza to, że w nazwach pakietów będą pojawiały się na początku części wspólne. Na przykład jeśli moduł zawiera pakiety:

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

jako nazwy modułu najlepiej będzie użyć części wspólnej nazw pakietów, czyli `com.deitel.sample`. Jeżeli taka część wspólna nie istnieje, można wybrać nazwę reprezentującą ogólny cel powstania modułu. Na przykład moduł `java.base` zawiera główne pakiety, które uważa się za kluczowe dla większości aplikacji Javy (między m.in. `java.lang`, `java.io`, `java.time` i `java.util`). Moduł `java.sql` zawiera pakiety związane z interakcją z bazami danych poprzez JDBC (pakiety `java.sql` i `javax.sql`). To jedynie dwa z wielu standardowych modułów przedstawionych

na rysunku 27.1. Dostępna online dokumentacja zawiera pełną listę pakietów eksportowanych przez każdy z modułów — w przypadku modułu `java.base` odwiedź stronę:

<http://download.java.net/java/jdk9/docs/api/java.base-summary.html>

Wyświetlenie zawartości modułu `java.base`

Użyj opcji `--describe-module` polecenia `java`, aby wyświetlić informacje na temat modułu `java.base` wydobywane z jego deskryptora. Informacje te będą zawierały między innymi listę eksportowanych pakietów:

```
java --describe-module java.base
```

Rysunek 27.4. przedstawia **fragment** wyniku działania polecenia zawierający listę pakietów modułu, do których może mieć dostęp **dowolny** inny moduł. Z wielu spośród tych pakietów, np. `java.io`, `java.lang`, `java.math`, `java.nio`, `java.time` i `java.util`, korzystaliśmy w poprzednich przykładach prezentowanych w książce.

<pre> exports java.io exports java.lang exports java.lang.annotation exports java.lang.invoke exports java.lang.module exports java.lang.ref exports java.lang.reflect exports java.math exports java.net exports java.net.spi exports java.nio exports java.nio.channels exports java.nio.channels.spi exports java.nio.charset exports java.nio.charset.spi exports java.nio.file exports java.nio.file.attribute exports java.nio.file.spi exports java.security exports java.util.stream exports java.util.zip exports javax.crypto exports javax.crypto.interfaces exports javax.crypto.spec exports javax.net exports javax.net.ssl </pre>	<pre> exports java.security.acl exports java.security.cert exports java.security.interfaces exports java.security.spec exports java.text exports java.text.spi exports java.time exports java.time.chrono exports java.time.format exports java.time.temporal exports java.time.zone exports java.util exports java.util.concurrent exports java.util.concurrent.atomic exports java.util.concurrent.locks exports java.util.function exports java.util.jar exports java.util.regex exports java.util.spi exports javax.security.auth exports javax.security.auth.callback exports javax.security.auth.login exports javax.security.auth.spi exports javax.security.auth.x500 exports javax.security.cert </pre>
--	--

Rysunek 27.4. Część wyników zwracanych przez polecenie `java --describe-module java.base`

Pełny zestaw wyników polecenia przedstawia wiele dodatkowych informacji na temat modułu `java.base`. Na rysunku 27.5 widoczne są niektóre rodzaje dodatkowych informacji z podziałem na kategorie.

<pre> ... uses java.util.spi.CurrencyNameProvider uses java.util.spi.ResourceBundleControlProvider uses java.util.spi.LocaleNameProvider </pre>

Rysunek 27.5. Fragment wyników działania polecenia `java --describe-module java.base` prezentujący pozostałe kategorie informacji

```

...
  provides java.nio.file.spi.FileSystemProvider
    with jdk.internal.jrtfs.JrtFileSystemProvider
...
  exports sun.net.sdp to oracle.net
  exports jdk.internal.jimage to jdk.jlink
  exports sun.net.www.protocol.http.ntlm to jdk.deploy
...
  contains com.sun.crypto.provider
  contains com.sun.java.util.jar.pack
  contains com.sun.net.ssl
...

```

Rysunek 27.5. Fragment wyników działania polecenia `java --describe-module java.base` prezentujący pozostałe kategorie informacji — *ciąg dalszy*

Wiersze `uses` w postaci:

```
uses java.util.spi.CurrencyNameProvider
```

wskazują, że istnieją typy w pakietach modułu `java.base`, które używają obiektów implementujących różne interfejsy dostawców usług. Zapis `provides...with`:

```
provides java.nio.file.spi.FileSystemProvider
  with jdk.internal.jrtfs.JrtFileSystemProvider
```

wskazuje, że pakiet `jdk.internal.jrtfs` modułu zawiera implementację klasy dostawcy usługi o nazwie `JrtFileSystemProvider`, która implementuje interfejs dostawcy usług o nazwie `FileSystemProvider` z pakietu `java.nio.file.spi`. Podrozdział 27.9 pokazuje większy przykład ilustrujący zastosowanie interfejsów i dostawców usługi do utworzenia **luźno powiązanych** komponentów systemu, co upraszcza tworzenie, konserwację i rozwój systemu w porównaniu z wersją ściśle powiązaną.

Zapis `exports...to` typu:

```
exports sun.net.sdp to oracle.net
```

wskazuje, że moduł `java.base` eksportuje pakiet `sun.net.sdp` jedynie do konkretnego modułu (w tym przypadku `oracle.net`). Moduł `java.base` ma wiele tego rodzaju kwalifikowanych eksportów. Pakiety wymienione w takich eksportach mogą być odczytywane tylko przez jeden lub więcej modułów wskazanych na liście po słowie `to`. W JDK kwalifikowane eksporty wykorzystuje się do udostępniania innym częściom JDK pakietów (np. `sun.net.sdp`), które stanowią implementację wewnętrzną i nie powinny być używane przez programistów.

Wiersze `contains` typu:

```
contains com.sun.crypto.provider
```

wskazują, że moduł zawiera pakiety, które nie są eksportowane w celu ich użycia przez inne moduły. Zauważ, że `contains` nie jest dyrektywą taką jak `requires` lub `exports` i nie można jej stosować w deklaracji modułu. To jedynie wstawiana przez kompilator informacja wskazująca, że dany pakiet w ogóle istnieje — pakiet nie jest eksportowany i nie jest dostępny poza modułem. JVM używa tych informacji do poprawy wydajności w trakcie wczytywania klas z pakietów¹².

¹² Brian Goetz poinformował o tym autorów książki w e-mailu przesłanym 16 marca 2017 roku.

27.3.2. Klasa Welcome

Rysunek 27.6 przedstawia klasę `Welcome`, która wyświetla tekst w konsoli. Definiując typy, które znajdują się w module, każdy z nich **trzeba** umieścić w pakiecie (wiersz 3.).

```

1 // Rysunek 27.6. Welcome.java
2 // Klasa Welcome, która zostanie umieszczona w module
3 package com.deitel.welcome; // Wszystkie klasy modułów muszą się znajdować
                               ↪ w pakietach
4
5 public class Welcome {
6     public static void main(String[] args) {
7         // Klasa System znajduje się w pakiecie java.lang modułu java.base
8         System.out.println("Witamy w systemie modułów platformy Java!");
9     }
10 }
```

Rysunek 27.6. Klasa `Welcome`, która zostanie umieszczona w module

27.3.3. Plik `module-info.java`

Rysunek 27.7 przedstawia deklarację modułu o nazwie `com.deitel.welcome`. Moduły, które tworzymy na swój własny użytek, nazywamy **modułami aplikacyjnymi**.

```

1 // Rysunek 27.7. module-info.java
2 // Deklaracja modułu com.deitel.welcome
3 module com.deitel.welcome {
4     requires java.base; // Niejawne dla wszystkich modułów, więc można pominąć
5 }
```

Rysunek 27.7. Deklaracja modułu `com.deitel.welcome`

Deklaracja modułu rozpoczyna się od słowa kluczowego `module`, po którym następuje nazwa modułu i nawiasy klamrowe otaczające treść modułu. Deklaracja zawiera tylko dyrektywę `requires` wskazującą, że kod aplikacji zależy od modułu `java.base`. Tak naprawdę wszystkie moduły zależą od `java.base`, więc dyrektywa z wiersza 4. właściwie jest stosowana **niejawnie** w każdej deklaracji modułu i można ją pominąć:

```

module com.deitel.welcome {
}
```

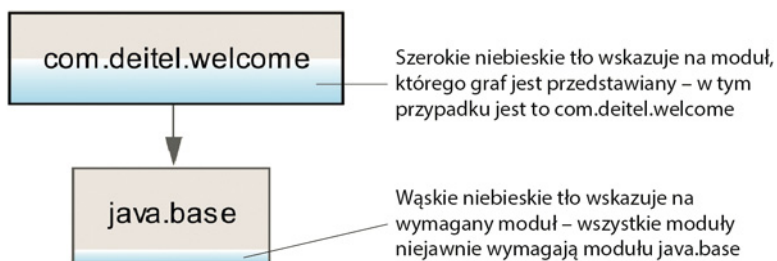


27.1. Obserwacja z poziomu inżynierii oprogramowania

Każdy moduł **niejawnie** zależy od `java.base`. Pisanie `requires java.base;` w deklaracji modułu jest więc całkowicie zbędne.

27.3.4. Graf zależności modułu

Rysunek 27.8 przedstawia **graf zależności modułu** `com.deitel.welcome`, który wyraźnie pokazuje, że moduł ten zależy tylko od modułu standardowego `java.base`. Wskazuje na to strzałka idąca od `com.deitel.welcome` do `java.base`. Graf będzie wyglądał identycznie niezależnie od tego, czy umieścimy w deklaracji wiersz 4., czy tego nie zrobimy.



Rysunek 27.8. Graf zależności modułu `com.deitel.welcome`

Graf zależności modułów przedstawia zależności między **modułami obserwowalnymi**¹³, czyli między wbudowanymi modułami standardowymi i dowolnymi modułami dodatkowymi wymaganymi przez aplikację lub moduł biblioteczny. **Węzły** grafu to moduły, a **zależności** to krawędzie skierowane (strzałki) łączące poszczególne węzły. Niektóre krawędzie reprezentują **jawne zależności** modułów wskazane w deklaracjach w dyrektywach `requires` (rysunek 27.14). Niektóre krawędzie są **zależnościami niejawnymi**, które wynikają z faktu, iż jeden wymagany moduł zależy od innego modułu (rysunek 27.22). Na rysunku 27.8 moduł `java.base` przedstawiany jest jako zależność jawna, bo zależą od niego wszystkie moduły.

Przedstawiony graf powstał na podstawie wczesnej wersji IDE NetBeans obsługującej JDK 9 — aby pobrać tę wersję lub uzyskać dodatkowe informacje na jej temat, skorzystaj z poniższego adresu WWW:

<http://wiki.netbeans.org/JDK9Support>

W projekcie NetBeans, gdy otworzysz plik `module-info.java`, możesz wybrać widok *Source* lub *Graph*. W widoku *Graph* IDE tworzy graf zależności modułu na podstawie deklaracji modułu. Gdy NetBeans rysuje graf, uwzględnia wszystkie zależności, w tym niejawną zależność od `java.base`. Rysunek 27.8 wskazuje, że moduł `java.base` nie ma innych zależności.

Aby utworzyć graf w NetBeans, wykonaliśmy następujące kroki:

1. Utworzyliśmy projekt `WelcomeApp` zawierający pakiet `com.deitel.welcome`.
2. Następnie dodaliśmy plik `module-info.java` modułu `com.deitel.welcome`, klikając prawym klawiszem myszy projekt i wybierając z menu kontekstowego *New/Other...*, a potem wybierając w oknie dialogowym *Java Module Info* z kategorii *Java*. Później kliknęliśmy przyciski *Next >* i *Finish*. Plik zostanie dodany automatycznie do domyślnego pakietu projektu.
3. Na końcu otwarliśmy plik `module-info.java`, zmieniliśmy domyślną nazwę modułu przypisaną przez NetBeans (nazwa projektu) na `com.deitel.welcome` i zmieniliśmy widok na *Graph*.

Możemy umieścić węzły w NetBeans przez ich przeciągnięcie lub kliknięcie grafu prawym klawiszem myszy i wybranie którejś spośród opcji z *Layouts* — wybrali-

¹³ Alan Bateman, Alex Buckley, Jonathan Gibbons i Mark Reinhold, „JEP 261: Module System”, <http://openjdk.java.net/jeps/261>

śmy wersję *Hierarchical*, która powoduje, że moduł pojawia się na górze, a strzałki dotyczące zależności wskazują w dół. Użyj polecenia *Zoom To Fit*, aby graf zajął całą dostępną przestrzeń okna, i polecenia *Export As Image*, aby zapisać obraz zawierający graf.

27.3.5. Kompilacja modułu

Aby skompilować moduł aplikacji, otwórz okno wiersza poleceń, użyj polecenia `cd`, aby przejść do folderu z przykładami *WelcomeApp*, i wpisz:

```
javac -d mods/com.deitel.welcome ^
src/com.deitel.welcome/module-info.java ^
src/com.deitel.welcome/com/deitel/welcome/Welcome.java
```

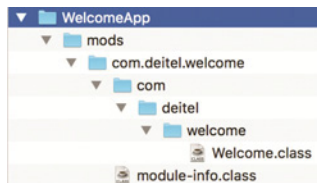
Opcja `-d` wskazuje, że kompilator `javac` powinien umieścić skompilowany kod w określonym folderze — w tym przypadku w folderze *mods*, który będzie zawierał podkatalog o nazwie *com.deitel.welcome* reprezentujący skompilowany moduł. Nazwę *mods* stosuje się standardowo przez konwencję, aby wskazać folder zawierający moduły.

Uwaga dotycząca długich poleceń stosowanych w tym rozdziale

Aby poprawić czytelność poleceń, dzielimy każde z nich na wiele wierszy, korzystając ze specjalnych znaków kontynuacji wiersza. Wiele poleceń z tego rozdziału jest długich. Prezentujemy tu polecenia w formacie dla systemu Windows, więc korzystamy ze znaku karety (^). W systemach Linux i macOS zastąp znak karety w poleceniach znakiem lewego ukośnika (\), aby wskazać kontynuację wiersza. Oczywiście możesz też wpisać polecenie jako jeden długi wiersz bez znaków kontynuacji.

Struktura folderów aplikacji po skompilowaniu

Jeśli uda się poprawnie skompilować kod, struktura podkatalogów w *mods* będzie zawierała skompilowany kod (rysunek 27.9). To tak zwany **rozbity folder modułu**, ponieważ katalogi i pliki *.class* nie znajdują się w pliku JAR (archiwum Javy), czyli w zbiorze folderów i plików skompresowanym do jednego folderu. Struktura folderów odpowiada strukturze folderów z katalogu *src*. Wkrótce spakujemy aplikację do pliku JAR. Rozbite foldery modułu i modułowe pliki JAR (punkt 27.3.7) to tak zwane **artefakty modułu**. Można je umieścić na ścieżce modułów — liście lokalizacji modułów — w trakcie kompilowania lub wykonywania modułowego kodu^{14,15}.



Rysunek 27.9. Struktura katalogów w folderze *mods* aplikacji

¹⁴ Mark Reinhold, „The State of the Module System”, 8 marca 2016, <http://openjdk.java.net/projects/jigsaw/spec/sotms/#module-artifacts>

¹⁵ Alan Bateman, Alex Buckley, Jonathan Gibbons i Mark Reinhold, „JEP 261: Module System”, <http://openjdk.java.net/jeps/261>

Wyświetlenie zawartości modułu com.deitel.welcome

Użyj opcji `--describe-module` polecenia `java`, aby wyświetlić informacje znajdujące się w deskryptorze modułu `com.deitel.welcome`. Oto przykład:

```
java --module-path mods --describe-module com.deitel.welcome
```

Wynik działania polecenia jest następujący:

```
com.deitel.welcome
file:///C:/przyklady/rozdziel27/WelcomeApp/mods/com.deitel.welcome/
requires java.base
contains com.deitel.welcome
```

Wynik pokazuje, że moduł wymaga standardowego modułu `java.base` i zawiera pakiet `com.deitel.welcome` (foldery bez klas nie są uwzględniane jako pakiety). Choć moduł zawiera (contains) pakiet, **nie** jest on eksportowany. Oznacza to, że inne moduły **nie mają** dostępu do pakietu. Deklaracja modułu dla przykładu **jawnie** wskazywała, że wymaga on `java.base`, więc wynik zawiera fragment:

```
requires java.base
```

Gdyby deklaracja modułu w sposób **niejawny** wymagała `java.base`, wynik zawierałby wpis:

```
requires mandated java.base
```

Nie istnieje dyrektywa `requires mandated` — to tekst wyświetlany jedynie w przypadku użycia opcji `--describe-module`.

27.3.6. Uruchamianie aplikacji z poziomu rozbitego folderu aplikacji

Aby uruchomić aplikację z poziomu rozbitego folderu aplikacji, użyj następującego polecenia (wykonaj je w folderze `WelcomeApp`):

```
java --module-path mods ^
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

Opcja `--module-path` określa ścieżki do modułów — w tym przypadku do folderu `mods`. Opcja `--module` określa nazwę modułu i w pełni kwalifikowaną nazwę punktu wejścia do aplikacji, czyli klasę zawierającą metodę `main`. Program po uruchomieniu wyświetli tekst:

```
Witamy w systemie modułów platformy Java!
```

W przedstawionym przykładzie opcję `--module-path` można zapisać skrótowo jako `-p`, natomiast `--module` jako `-m`.

27.3.7. Spakowanie modułu do pliku JAR

Użyj polecenia `jar` do spakowania rozbitego folderu modułu jako **modułowego pliku JAR**¹⁶, który zawiera wszystkie pliki modułu, w tym plik `module-info.class` umieszczany w głównym folderze archiwum JAR. W trakcie uruchamiania aplikacji to plik JAR podaje się w ścieżce modułu. Folder, w którym chce się umieścić plik JAR, musi istnieć przed uruchomieniem polecenia `jar`.

¹⁶ Alan Bateman, Alex Buckley, Jonathan Gibbons i Mark Reinhold, „JEP 261: Module System”, <http://openjdk.java.net/jeps/261>

Jeśli moduł zawiera punkt wejścia do aplikacji, można wskazać odpowiednią klasę w poleceniu `jar` opcją `--main-class`. Oto przykład:

```
jar --create -f jars/com.deitel.welcome.jar ^
--main-class com.deitel.welcome.Welcome ^
-C mods/com.deitel.welcome .
```

Działanie poszczególnych opcji jest następujące:

- opcja `--create` wskazuje zamiar utworzenia nowego pliku JAR;
- opcja `-f` określa nazwę pliku JAR — nazwę wraz z ewentualnym folderem wskazuje się tuż po opcji (w tym przypadku będzie to plik *com.deitel.welcome.jar* w folderze *jars*);
- opcja `--main-class` wskazuje w pełni kwalifikowaną nazwę klasy będącej punktem wejścia do aplikacji, czyli klasą zawierającą metodę `main`;
- opcja `-C` określa folder, który zawiera pliki do umieszczenia w pliku JAR, a dodatkowo wskazuje znakiem kropki (`.`), że należy umieścić w archiwum wszystkie pliki z tego folderu.

Opcje `-create`, `-f` i `--main-class` można zapisać skrótowo jako `-cfe`, a następnie wskazać nazwę pliku JAR i główną klasę.

```
jar -cfe jars/com.deitel.welcome.jar ^
com.deitel.welcome.Welcome ^
-C mods/com.deitel.welcome .
```

27.3.8. Uruchamianie aplikacji z modułowego pliku JAR

Gdy już umieściliśmy aplikację w modułowym pliku JAR ze wskazanym punktem wejścia do aplikacji, wystarczy użyć zapisu:

```
java --module-path jars -m com.deitel.welcome
```

lub:

```
java -p jars -m com.deitel.welcome
```

Program po uruchomieniu wyświetli tekst:

```
Witamy w systemie modułów platformy Java!
```

Jeśli nie wskazaliśmy punktu wejścia do aplikacji przy tworzeniu pliku JAR, nadal możemy uruchomić aplikację, ale trzeba wskazać nazwę modułu i w pełni kwalifikowaną nazwę klasy:

```
java --module-path jars ^
-m com.deitel.welcome/com.deitel.welcome.Welcome
```

lub:

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

27.3.9. Dodatkowa uwaga — ścieżka klas a ścieżka modułów

Przed Javą 9 kompilator i system wykonawczy lokalizowały typy za pomocą **ścieżki klas** — listy folderów i plików archiwów zawierających skompilowane klasy Javy. We wcześniejszych wersjach Javy ścieżka klas była definiowana jako zawartość zmiennej środowiskowej `CLASSPATH`, rozszerzenia umieszczane w specjalnym folderze `JRE` lub opcje przekazywane do poleceń `java` i `javac`.

Ponieważ typy mogą być wczytywane z kilku różnych lokalizacji, kolejność tych lokalizacji mogła negatywnie wpływać na stabilność aplikacji. Przykładowo wiele lat temu jeden z autorów książki zainstalował aplikację Javy od niezależnego dostawcy. Instalator aplikacji umieścił starą wersję biblioteki Javy w folderze rozszerzeń JRE. Kilka aplikacji Javy na komputerze wymagało nowszej wersji, bo korzystało z dodatkowych typów. Ponieważ klasy z folderu rozszerzeń JRE są stosowane **przed** innymi klasami ścieżki klas¹⁷, aplikacje zależne od nowszej biblioteki przestały działać, zgłaszając wyjątki `NoClassDefFoundErrors` i `NoSuchMethod` `↳Errors` — co gorsza, czasem dopiero po jakimś czasie działania aplikacji.

Pewność konfiguracji zapewniana przez moduły i ich deskryptory pomaga wyeliminować wiele problemów związanych ze ścieżkami klas. Każdy moduł jawnie wskazuje swoje zależności, a te są rozwiązywane **w momencie uruchamiania aplikacji**. W punkcie 27.8.5 wymienimy kroki, które **mechanizm rozwiązywania modułów** wykonuje w trakcie uruchamiania aplikacji.

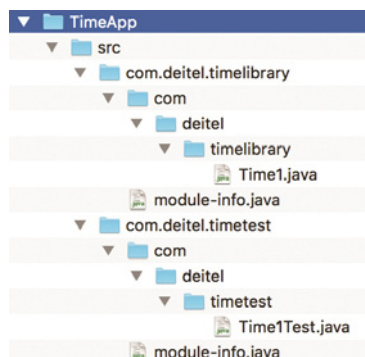


27.1. Typowy błąd programistyczny

Ścieżka modułów może zawierać konkretny moduł tylko raz, a pakiet może być zdefiniowany tylko w jednym module. Jeśli kilka modułów ma taką samą nazwę lub kilka modułów eksportuje ten sam pakiet, system wykonawczy kończy działanie jeszcze przed uruchomieniem aplikacji.

27.4. Tworzenie i użycie własnych modułów

Aby pokazać moduł, który zależy od innego, napisanego przez nas modułu, a nie tylko od modułów standardowych, wykorzystajmy jeden z wcześniejszych przykładów, który nie korzystał w ogóle z modułów. Zadeklarujemy klasy `Time1` i `Time1Test` z podrozdziału 8.2 w osobnych modułach, a następnie użyjemy klasy `Time1` z poziomu modułu zawierającego `Time1Test`. Przekonasz się, że **wyeksportujemy** klasę `Time1` z jednego modułu i będziemy jej **wymagać** w innym module (z klasą `Time1Test`), wskazując nazwę tego modułu. Rysunek 27.10 przedstawia strukturę folderów dla obu modułów aplikacji.



Rysunek 27.10. Struktura folderów w katalogu `src` przykładu `TimeApp`

¹⁷ „Understanding Extension Class Loading”, <https://docs.oracle.com/javase/tutorial/ext/basics/load.html>

27.4.1. Eksport pakietu w celu użycia w innych modułach

Jak wcześniej wspomnieliśmy, każda klasa, która ma zostać umieszczona w module, **musi** być zadeklarowana w pakiecie. Z tego powodu dodaliśmy instrukcję `package` w wierszu 3. (rysunek 27.11) klasy `Time1` (klasa była pierwotnie zadeklarowana na rysunku 8.1).

```

1 // Rysunek 27.11. Time1.java
2 // Klasa Time1, która zostanie umieszczona w module
3 package com.deitel.timelibrary;
4
5 public class Time1 {
6     private int hour; // 0–23
7     private int minute; // 0–59
8     private int second; // 0–59
9
10    // Ustaw nowy czas, używając czasu uniwersalnego
11    // Zgłoś wyjątek, jeśli godzina, minuta lub sekunda znajduje się poza zakresem
12    public void setTime(int hour, int minute, int second) {
13        // Sprawdź poprawność parametrów hour, minute i second
14        if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
15            second < 0 || second >= 60) {
16            throw new IllegalArgumentException(
17                "Wartość parametru hour, minute lub (i) second poza
18                ↪zakresem");
19        }
20        this.hour = hour;
21        this.minute = minute;
22        this.second = second;
23    }
24
25    // Zamień na tekst w formacie uniwersalnym (HH:MM:SS)
26    public String toUniversalString() {
27        return String.format("%02d:%02d:%02d", hour, minute, second);
28    }
29
30    // Zamień na tekst w formacie 12-godzinnym (H:MM:SS AM lub PM)
31    public String toString() {
32        return String.format("%d:%02d:%02d %s",
33            ((hour == 0 || hour == 12) ? 12 : hour % 12),
34            minute, second, (hour < 12 ? "AM" : "PM"));
35    }
36 }
```

Rysunek 27.11. Klasa `Time1`, która zostanie umieszczona w module

Deklaracja modułu `com.deitel.timelibrary`

Po umieszczeniu klasy `Time1` w pakiecie musimy zadeklarować moduł (rysunek 27.12). Wiersz 4. wskazuje, że moduł `com.deitel.timelibrary` eksportuje pakiet `com.deitel.timelibrary`. Klasy publiczne pakietu (w tym przypadku tylko klasa `Time1`) będą mogły być używane przez **dowolny** moduł odczytujący moduł `com.deitel.timelibrary`, o ile oczywiście znajdzie się on w ścieżce modułów w sposób opisany w punkcie 27.4.3.

```

1 // Rysunek 27.12. module-info.java
2 // Deklaracja modułu com.deitel.timelibrary
3 module com.deitel.timelibrary {
4     exports com.deitel.timelibrary; // Pakiet dostępny dla innych modułów
5 }

```

Rysunek 27.12. Deklaracja modułu `com.deitel.timelibrary`

27.4.2. Wykorzystanie klasy pakietu w innym module

Punkt wejścia do aplikacji — klasa `Time1Test` (pierwotnie zadeklarowana na rysunku 8.2) — również musi się znaleźć w pakiecie, aby mógł trafić do modułu (wiersz 3. z rysunku 27.13). Klasa `Time1Test` korzysta z obiektu klasy `Time1` zadeklarowanego w pakiecie innego modułu. Z tego powodu importujemy w wierszu 5. klasę `Time1`.

```

1 // Rysunek 27.13. Time1Test.java
2 // Obiekt Time1 użyty w aplikacji
3 package com.deitel.timetest;
4
5 import com.deitel.timelibrary.Time1;
6
7 public class Time1Test {
8     public static void main(String[] args) {
9         // Utworzenie i inicjalizacja obiektu Time1
10         Time1 time = new Time1(); // Wywołanie konstruktora Time1
11
12         // Wyświetlenie tekstowych reprezentacji czasu
13         displayTime("Po utworzeniu obiektu", time);
14         System.out.println();
15
16         // Zmień czas i wyświetl wartości ponownie
17         time.setTime(13, 27, 6);
18         displayTime("Po wywołaniu setTime", time);
19         System.out.println();
20
21         // Spróbuj ustawić niepoprawny czas
22         try {
23             time.setTime(99, 99, 99); // Wszystkie wartości są poza zakresem
24         }
25         catch (IllegalArgumentException e) {
26             System.out.printf("Wyjątek: %s%n%n", e.getMessage());
27         }
28
29         // Wyświetl czas po próbie użycia wartości spoza dopuszczalnego zakresu
30         displayTime("Po wywołaniu setTime z nieprawidłowymi wartościami",
31             ↪time);
32     }
33
34     // Wyświetla obiekt Time1 w formatach 12- i 24- godzinny
35     private static void displayTime(String header, Time1 t) {
36         System.out.printf("%s%nFormat uniwersalny: %s%nFormat 12-godzinny:
37             ↪%s%n",
38             header, t.toUniversalString(), t.toString());
39     }
40 }

```

Rysunek 27.13. Obiekt `Time1` użyty w aplikacji

Deklaracja modułu *com.deitel.timetest*

Ponieważ klasa `Time1` znajduje się w pakiecie wewnątrz modułu `com.deitel.timelibrary`, moduł zawierający klasę `Time1Test` (`com.deitel.timetest`) musi zadeklarować ten drugi moduł jako zależność. Deklaracja modułu (rysunek 27.14) wskazuje tę zależność za pomocą dyrektywy `requires` (wiersz 4.). Bez tego oraz dyrektywy `exports` z rysunku 27.12 klasa `Time1Test` nie mogłaby zaimportować i użyć klasy `Time1`.

```

1 // Rysunek 27.14. module-info.java
2 // Deklaracja modułu com.deitel.timetest
3 module com.deitel.timetest {
4     requires com.deitel.timelibrary;
5 }

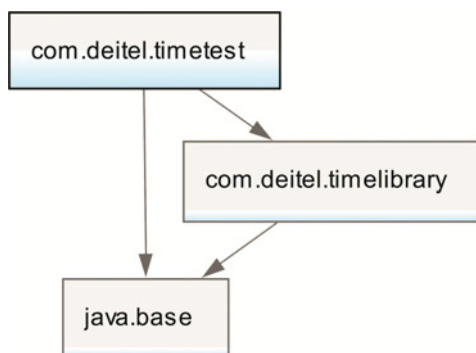
```

Rysunek 27.14. Deklaracja modułu `com.deitel.timetest`

Graf zależności modułu *com.deitel.timetest*

Rysunek 27.15. przedstawia graf zależności modułu `Time1Test` wskazujący, że:

- moduł o nazwie `com.deitel.timetest` odczytuje moduł `com.deitel.timelibrary` i standardowy moduł `java.base`;
- moduł o nazwie `com.deitel.timelibrary` odczytuje moduł `java.base`.



Rysunek 27.15. Graf zależności modułu `com.deitel.timetest`

Aby utworzyć graf w IDE NetBeans, wykonaliśmy następujące kroki:

1. Utworzyliśmy projekt `TimeLibrary` zawierający pakiet `com.deitel.timelibrary` oraz plik `module-info.java` modułu `com.deitel.timelibrary`.
2. Utworzyliśmy projekt `TimeApp` zawierający pakiet `com.deitel.timetest` oraz plik `module-info.java` modułu `com.deitel.timetest`.
3. Kliknęliśmy prawym klawiszem myszy węzeł *Libraries* projektu `TimeApp` i wybraliśmy polecenie *Add Project...*, wybraliśmy projekt `TimeLibrary` i kliknęliśmy *Add Project JAR Files* — spowoduje to dodanie pliku JAR modułu projektu `TimeLibrary` do projektu `TimeApp`.
4. Otworzyliśmy plik `module-info.java` projektu `TimeApp` w widoku *Graph*.

27.4.3. Kompilacja i uruchomienie przykładu

Przed uruchomieniem aplikacji trzeba skompilować oba moduły. Moduł `com.deitel.timelibrary` musi być kompilowany jako pierwszy, ponieważ moduł `com.deitel.timetest` jest od niego zależny. Narzędzia IDE lub inne narzędzia budowania (takie jak Ant, Gradle lub Maven) automatycznie wykrywają zależności i kompilują kod w odpowiednim porządku.

Kompilacja modułu `com.deitel.timelibrary`

Aby skompilować moduł `com.deitel.timelibrary`, należy otworzyć okno polecenia, użyć polecenia `cd` do zmiany aktualnego folderu na `TimeApp` i wykonać następujące polecenie:

```
javac -d mods/com.deitel.timelibrary ^
    src/com.deitel.timelibrary/module-info.java ^
    src/com.deitel.timelibrary/com/deitel/timelibrary/Time1.java
```

Kompilacja modułu `com.deitel.timetest`

Następnie należy wpisać następujące polecenie, aby skompilować moduł `com.deitel.timetest`:

```
javac --module-path mods -d mods/com.deitel.timetest ^
    src/com.deitel.timetest/module-info.java ^
    src/com.deitel.timetest/com/deitel/timetest/Time1Test.java
```

Dodaliśmy tutaj opcję `--module-path`, aby wskazać, że folder `mods` zawiera moduły, od których zależy moduł `com.deitel.timetest` — w tym przypadku jest to skompilowany wcześniej moduł `com.deitel.timelibrary`.

Uruchomienie przykładu

Aby uruchomić przykład, wpisz polecenie:

```
java --module-path mods ^
    -m com.deitel.timetest/com.deitel.timetest.Time1Test
```

W tym poleceniu:

- opcja `--module-path` wskazuje miejsce umieszczenia modułów aplikacji;
- opcja `-m` określa, która klasa powinna być stosowana jako punkt wejścia do aplikacji — innymi słowy jest to klasa zawierająca metodę `main`, którą ma uruchomić maszyna wirtualna Javy.

Zauważ, że w przypadku metody `main` trzeba wskazać nazwę modułu, a po ukośniku pełną nazwę klasy, ponieważ klasa znajduje się w pakiecie zawartym w module. Oto wynik działania programu:

```
Po utworzeniu obiektu
Format uniwersalny: 00:00:00
Format 12-godzinny: 12:00:00 AM
```

```
Po wywołaniu setTime
Format uniwersalny: 13:27:06
Format 12-godzinny: 1:27:06 PM
```

Wyjątek: Wartość parametru `hour`, `minute` lub `(i) second` poza zakresem

Po wywołaniu `setTime` z nieprawidłowymi wartościami

Format uniwersalny: 13:27:06

Format 12-godzinny: 1:27:06 PM

27.4.4. Zapakowanie aplikacji do modułowych plików JAR

W tym punkcie umieścimy każdy z modułów we własnym pliku JAR, gdyż wówczas łatwiej będzie uruchomić aplikację. Aby umieścić `com.deitel.timelibrary` w modułowym pliku JAR, użyj polecenia:

```
jar --create -f jars/com.deitel.timelibrary.jar ^
-C mods/com.deitel.timelibrary .
```

Aby umieścić `com.deitel.timetest` w modułowym pliku JAR, użyj polecenia:

```
jar --create -f jars/com.deitel.timetest.jar ^
--main-class com.deitel.timetest.TimeTest ^
-C mods/com.deitel.timetest .
```

Uruchomienie aplikacji z modułowego pliku JAR

Gdy moduły znajdują się w swoich plikach JAR i wskazaliśmy klasę zawierającą metodę `main`, możemy uruchomić aplikację bardzo prostym poleceniem:

```
java --module-path jars -m com.deitel.timetest
```

Program wykona się i wyświetli taki sam wynik jak w punkcie 27.4.3.

27.4.5. Silna enkapsulacja i dostępność

Przed Javą 9 można było użyć dowolnej klasy publicznej zaimportowanej przez kod. To, czy mieliśmy dostęp do składowych klasy, zależało od sposobu ich zadeklarowania: `public`, `protected`, `private` lub dostęp na poziomie pakietu (zobacz opisy w rozdziałach od 3. do 8.). Z powodu **silnej enkapsulacji** modułowej wprowadzonej w Javie 9 typy publiczne nie są już domyślnie **dostępne** dla kodu — `public` nie oznacza więc już dostępności dla wszystkich.

- Jeśli moduł eksportuje pakiet, typy publiczne tego pakietu są dostępne dla **każdego** modułu, który wskaże, że odczytuje ten pakiet.
- Jeśli moduł eksportuje pakiet na potrzeby konkretnego modułu (dyrektywa `exports...to`), typy publiczne tego pakietu są dostępne **jedynie** dla wskazanych modułów i tylko jeśli moduł ten **odczytuje** pakiet modułu.
- Jeśli moduł nie eksportuje pakietu, typy publiczne tego pakietu są dostępne **jedynie** dla innych pakietów wewnątrz tego samego modułu.

Jeżeli mamy dostęp do typu w innym module, nadal mają zastosowanie standardowe zasady dotyczące `public`, `protected`, `private` lub dostępu na poziomie pakietu.

Błąd kompilacji przy próbie użycia niedostępnego typu

Projekt `TimeAppMissingExports` w folderze *ExamplesShowingErrors* pokazuje, że **jawnie nazwane moduły** stosują silną enkapsulację i nie eksportują pakietów, które nie są jawnie wymienione za pomocą dyrektyw `exports`. W tym projekcie usunęliśmy dyrektywę `exports` z deklaracji modułu `com.deitel.timelibrary`, po czym ponownie skompilowaliśmy moduł. Następnie spróbaliśmy znów

skompilować moduł `com.deitel.timetest`. Kompilator zgłosił poniższy błąd, który wskazuje, że pakiet `com.deitel.timelibrary` nie jest eksportowany, a tym samym nie jest dostępny:

```
src\com.deitel.timetest\com\deitel\timetest\Time1Test.java:5:
error: package com.deitel.timelibrary is not visible
import com.deitel.timelibrary.Time1;
            ^
(package com.deitel.timelibrary is declared in module
com.deitel.timelibrary, which is not in the module graph)
1 error
```



27.2. Typowy błąd programistyczny

Błąd kompilacji pojawi się, jeśli zależność `requires` nie odpowiada po drugiej stronie dyrektywa `exports` pozwalająca na dostęp do pakietu.

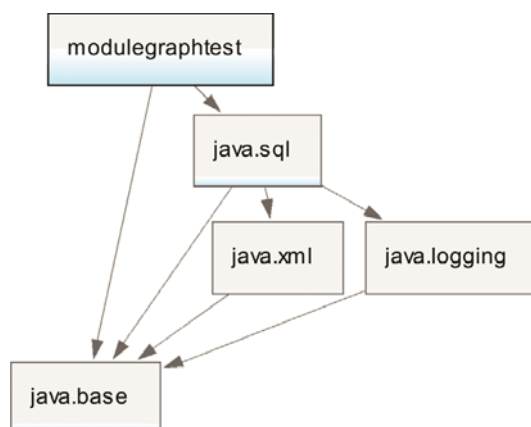
27.5. Graf zależności modułu — dokładniejsze spojrzenie

Przedstawiliśmy wcześniej dwa grafy zależności modułu. Teraz przedstawimy dodatkowe informacje na ich temat oraz pokażemy błędy, które występują, jeśli moduł bezpośrednio lub pośrednio wymaga sam siebie, czyli tworzy tak zwaną zależność cykliczną.

27.5.1. Moduł `java.sql`

Rysunek 27.16 przedstawia graf zależności modułu `modulegraphptest`, który jako swoją zależność ma moduł `java.sql` ze względu na użycie następującej deklaracji modułu:

```
module modulegraphptest {
    requires java.sql;
}
```



Rysunek 27.16. Graf zależności modułu, który zależy od modułu `java.sql`

NetBeans oznacza moduł zadeklarowany przez deklarację modułu (`modulegraphptest`) szerokim niebieskim tłem w dolnej części. Wąskim niebieskim tłem podświetla

moduł `java.sql`, bo jest on **jawnie** wymieniony jako dyrektywa `requires`, a także moduł `java.base`, który jest niejawnie wymagany przez wszystkie moduły. Pozostałe moduły (`java.xml` i `java.logging`) pojawiają się w grafie, bo zależy od nich moduł `java.sql`.

27.5.2. Moduł `java.se`

Rysunek 27.17 przedstawia zdecydowanie bardziej złożony graf zależności modułu **`java.se`** — to **moduł agregujący**, który określa poprzez `requires transitive` wszystkie moduły niezbędne do obsługi aplikacji Javy SE 9. Aby wykonać ten graf, pobraliśmy najpierw kod źródłowy JDK 9 w sposób opisany na stronie:

<http://hg.openjdk.java.net/jdk9/jdk9/raw-file/tip/common/doc/building.html>

Otworzyliśmy następnie deklarację modułu `java.se` (znajduje się w folderze `jdk/src/java.se/share/classes`) w trybie widoku *Graph* w IDE NetBeans. Dla poprawienia czytelności graf został obrócony o 90 stopni. Istnieje również moduł agregujący **`java.se.ee`**, który zawiera wszystko z modułu `java.se`, a także dodatkowe moduły Javy SE wspólne z platformą Java EE (Enterprise Edition).

27.5.3. Wyświetlenie grafu zależności modułów JDK

Warto przyjrzeć się pełnemu grafowi zależności modułów całego JDK. To największy graf, który pokazujemy. Jest on dostępny na stronie WWW pod adresem:

<http://deitel.com/bookresources/jhttp11/ModularJDKGraph.png>

Gdy otworzysz go w przeglądarce, będzie pomniejszony, bo jest naprawdę duży. Kliknij ikony powiększenia, a następnie przesuwaj zawartość okna w pionie lub poziomie. Graf wykonaliśmy przy użyciu narzędzia Graphviz dostępnego pod adresem:

<http://www.graphviz.org/>

27.5.4. Błąd — graf modułu z cyklem

Moduł nie może bezpośrednio lub pośrednio odnosić się do samego siebie. Doprowadziłoby to do powstania **cyklu** w trakcie wyliczania grafu zależności modułu.



27.3. Typowy błąd programistyczny

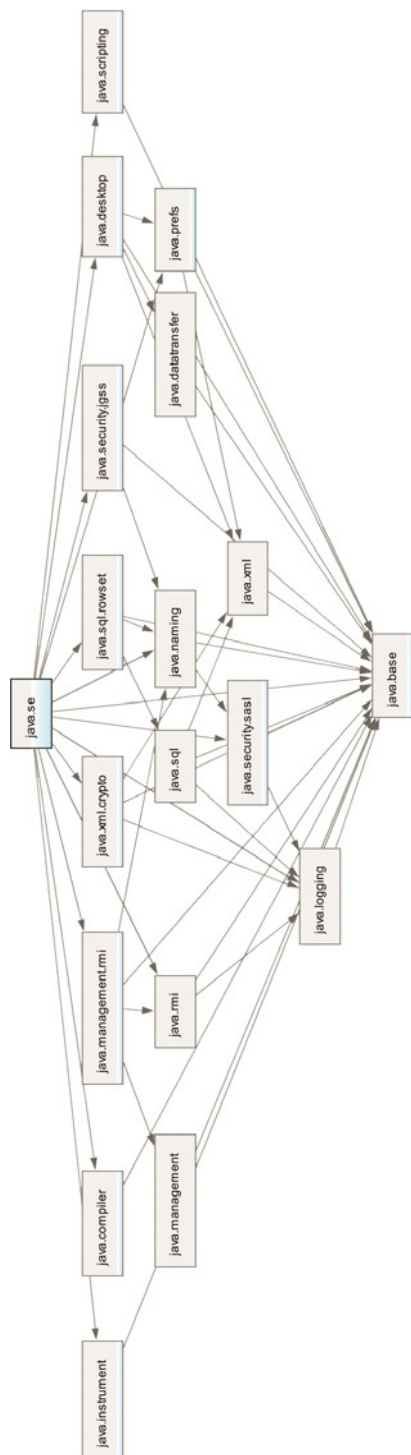
Błąd kompilacji wystąpi, jeśli graf modułu zawiera cykl.

Moduł, który niepoprawnie odnosi się do samego siebie

Przyjrzyjmy się następującej deklaracji modułu, który odnosi się sam do siebie:

```
module mymodule {
    requires mymodule;
}
```

Gdy skompilujemy taką deklarację, wystąpi błąd wskazujący na cykl w zależnościach modułu:

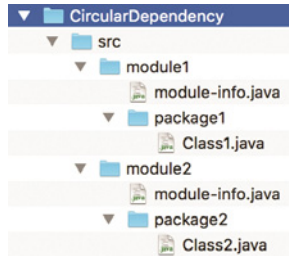


Rysunek 27.17. Graf zależności modułu java.se


```
module-info.java:2: error: cyclic dependence involving mymodule
    requires mymodule;
    ^
1 error
```

Dwa moduły, które niepoprawnie wymagają siebie nawzajem

Przjrzyjmy się teraz projektowi o nazwie `CircularDependency` składającemu się z dwóch modułów — `module1` i `module2` — o strukturze przedstawionej na rysunku 27.18.



Rysunek 27.18. Struktura folderu `src` projektu `CircularDependency`

Jeśli deklaracje modułu dla tych dwóch modułów wskazują, że każdy z nich wymaga drugiego, czyli istnieją zapisy:

```
module module1 {
    exports package1;
    requires module2;
}
```

i

```
module module2 {
    exports package2;
    requires module1;
}
```

to w momencie próby kompilacji tych modułów poleceniem:

```
javac --module-source-path src ^
      --module-path mods -d mods ^
      src/module1/module-info.java ^
      src/module1/package1/Class1.java ^
      src/module2/module-info.java ^
      src/module2/package2/Class2.java
```

kompilator zgłosi błąd wskazujący istnienie cyklu w zależnościach modułu:

```
src\module1\module-info.java:9: error: cyclic dependence involving
↳ module2
    requires module2;
    ^
1 error
```

Moduły w cyklu są tak naprawdę „jedną rzeczą”

W zasadzie wszystkie moduły tworzące cykl są tak naprawdę jednym modulem, a nie osobnymi modułami¹⁸. Gdy pisaliśmy ten rozdział, nasz znajomy, który pracuje dla dużej organizacji, powiedział nam, że przygotowują się oni na modularność Javy 9. Wskazał, że posiadają wiele dużych plików JAR sprzed Javy 9. Początkowo wydawało im się, że z każdego pliku JAR uczynią osobny moduł, ale okazało się, że istnieje między tymi plikami tyle zależności, iż wszystko musi się znaleźć w jednym module. Takie wzajemne zależności prowadzą do powstawania cykli. W idealnej sytuacji, gdy dochodzi do modularyzacji systemu, który wcześniej był monolityczny, chcemy uzyskać wiele niezależnych od siebie modułów, co pozwala na łatwiejszą konserwację i większe bezpieczeństwo. Stanowi to jednak poważne wyzwanie refaktoryzacyjne, jeśli baza kodu jest naprawdę duża.

27.6. Migracja kodu do Javy 9

Wiele aplikacji napisanych przed Javą 9 będzie bez przeszkód działało w Javie 9. Przygotowując tę książkę, sprawdzaliśmy każdą aplikację za pomocą JDK 9 i wszystkie działały bez zarzutu. W Javie 9 wszystkie programy są kompilowane i wykonywane z użyciem systemu modułów. Java 9 silnie enkapsuluje typy, które nie są eksportowane przez moduły, więc może się zdarzyć, że niektórych aplikacji nie uda się skompilować, bo wcześniej dostępne pakiety nie są już dostępne oficjalnie w Javie 9. Istnieje wiele **wewnętrznych API** sprzed Javy 9, które nie miały być stosowane poza JDK, ale tak naprawdę były poza nim stosowane — wiele z nich w Javie 9 nie jest eksportowanych, więc przestaje być dostępnych dla ogółu aplikacji¹⁹. Jeśli Twój kod używa wewnętrznych API pośrednio lub bezpośrednio, nie uda się go skompilować.

Niektóre wewnętrzne API uznane za szczególnie krytyczne są nadal dostępne w Javie 9. Wiele dokumentów JEP, do których odnośniki znajdują się w JSR 379²⁰, definiuje nowe, publiczne API, które zastępują stare. Wszystkie wewnętrzne API zostaną za jakiś czas usunięte.

**27.2. Obserwacja z poziomu inżynierii oprogramowania**

Modularność umożliwia silną enkapsulację. Kod, który nie został jawnie wyeksportowany, nie jest dostępny dla pozostałych modułów.

**27.4. Typowy błąd programistyczny**

JDK 9 ukrywa większość wewnętrznego API istniejącego przed Javą 9, więc kod, który go używa, nie będzie się kompilował lub uruchamiał w Javie 9.

¹⁸ Alex Buckley wskazał to w e-mailu do autorów książki przesłanym 24 marca 2017 roku.

¹⁹ Mark Reinhold, „JEP 260: Encapsulate Most Internal APIs”, <http://openjdk.java.net/jeps/260>

²⁰ Iris Clark i Mark Reinhold, „Java SE 9 (JSR 379)”, 6 marca 2017, <http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>



27.1. Wskazówka zapobiegająca błędom

Można użyć narzędzia `jdeps` (punkt 27.6.3) wydanego wraz z Javą 8, aby znaleźć zależności typu lub zależności wszystkich typów znajdujących się w pliku JAR. W Javie 9 narzędzie to obsługuje również moduły. Opcja `--jdk-internals` pozwala zidentyfikować kod, który korzysta z wewnętrznych API Javy. Niektóre wewnętrzne API sprzed Javy 9 zostały umieszczone w modułach, ale część jest obecnie w pełni ukryta. Dla każdego wewnętrznego API, które zlokalizuje `jdeps`, możesz przejrzeć JEP 260, aby zobaczyć, jak należy uaktualnić kod.

Java ma już ponad 20 lat, więc istnieje naprawdę dużo starego kodu Javy, który trzeba zmigrować do Javy 9. System modułów ma mechanizmy pozwalające na automatyczne umieszczenie kodu w modułach w celu wspomaganie migracji.

27.6.1. Moduł nienazwany

W Javie 9 wymaga się, aby cały kod znajdował się w modułach. Gdy wykonujemy kod, który nie znajduje się w module, kod jest wczytywany ze ścieżki klas i umieszczany w **module nienazwanym**. Dzięki temu możemy uruchamiać niemodułowy kod w modułowym JDK, ale niestety nie uzyskujemy w ten sposób zalet modularyzacji.

Moduł nienazwany:

- **niejawnie eksportuje** wszystkie swoje pakiety;
- **niejawnie odczytuje** wszystkie inne moduły.

Ponieważ moduł jest **nienazwany**, nie możemy się do niego odnieść w dyrektywie `requires` nazwanego modułu i dlatego nazwany moduł nie może zależeć od modułu nienazwanego.

27.6.2. Moduły automatyczne

Istnieje ogromna liczba bibliotek, z których korzysta wiele aplikacji. Sporo spośród tych bibliotek nie jest modułowych. Aby wspomóc proces migracji, możemy dodać **dowolny** plik JAR biblioteki do ścieżki modułów aplikacji, a następnie korzystać z pakietów zawartych w tym pliku JAR. W takiej sytuacji plik JAR staje się **niejawnie modulem automatycznym** i może być wskazany w dyrektywie `requires` deklaracji modułu. Nazwa pliku JAR (po odjęciu rozszerzenia `.jar`) staje się nazwą modułu, więc **musi być poprawnym identyfikatorem** Javy w celu jej użycia w dyrektywie `requires`. Dodatkowo moduł automatyczny:

- **niejawnie eksportuje** wszystkie pakiety — dowolny moduł czytający moduł automatyczny (włącznie z modulem nienazwanym) ma dostęp do typów publicznych pakietów modułu automatycznego;
- **niejawnie czyta** (`requires`) wszystkie inne moduły, w tym inne moduły automatyczne i moduł nienazwany — moduł automatyczny ma więc dostęp do wszystkich publicznych typów udostępnianych przez inne moduły systemu.

Przykład modułu automatycznego przedstawimy w podrozdziale 27.7.

27.6.3. Narzędzie jdeps — analiza zależności

8 Innym narzędziem wspomagającym proces migracji kodu do Javy 9 jest polecenie **jdeps**, które wprowadzono w Javie 8 w celu wspomaganie określania zależności typów **klas** i **pakietów**. Jednym z najważniejszych zastosowań narzędzia jest znajdowanie w kodzie sprzed Javy 9 zależności od wewnętrznego API, które w Javie 9 zostało hermetycznie zamknięte. Aby sprawdzić, czy klasa ma tego rodzaju zależności, użyj następującego polecenia dla kodu skompilowanego przed pojawieniem się Javy 9:

```
jdeps --jdk-internals NazwaKlasy.class
```

A jeśli posiadasz plik JAR z wieloma klasami, użyj polecenia:

```
jdeps --jdk-internals NazwaPlikuJAR.jar
```

Jeżeli polecenie nie zwróci żadnych wyników, klasa lub zbiór klas nie ma żadnych zależności dotyczących wewnętrznego API JDK, które w Javie 9 przestało być dostępne.



27.2. Wskazówka zapobiegająca błędom

Sprawdź każdą klasę lub plik JAR skompilowany przed Javą 9 poleceniem jdeps, aby upewnić się, że kod nie zależy od wewnętrznych API Javy.

Określanie potrzebnych modułów

Java 9 dodaje możliwość odkrywania zależności **modułu** w jej kodzie. Gdy przygotowujemy się do utworzenia **własnej** wersji wykonawczej, narzędzie jdeps może posłużyć do określenia zależności aplikacji, co pozwala poznać wymagane do dołączenia moduły. Przykładowo aplikacja powitalna zależy tylko od modułu java.base. Aby to potwierdzić, możemy użyć z poziomu folderu *WelcomeApp* poniższego polecenia, które sprawdza zależności modułu com.deitel.welcome:

```
jdeps --module-path jars -m com.deitel.welcome
```

Otrzymujemy informację o pakietach i modułach używanych przez aplikację:

```
com.deitel.welcome
[file:///C:/przyklady/rozdzial27/WelcomeApp/jars/com.deitel.welcome.jar]
requires java.base (@9-ea)
com.deitel.welcome -> java.base
com.deitel.welcome -> java.io      java.base
com.deitel.welcome -> java.lang    java.base
```

Wynik pokazuje, że moduł com.deitel.welcome zależy od modułu java.base, a konkretniej używa tylko dwóch pakietów tego modułu: java.io i java.lang.

Powyższe polecenie można też zapisać jako:

```
jdeps jars/com.deitel.welcome.jar
```

Można także zastosować polecenie jdeps dla konkretnego pliku *.class*:

```
jdeps mods/com.deitel.welcome/com/deitel/welcome/Welcome.class
```

Spowoduje ono zwrócenie następującego wyniku:

```
Welcome.class -> java.base
com.deitel.welcome -> java.io      java.base
com.deitel.welcome -> java.lang    java.base
```

Rozbudowane wyniki narzędzia *jdeps*

Jeśli chcesz poznać więcej szczegółów, zastosuj opcję `-v` jak w tym oto przykładzie:

```
jdeps -v jars/com.deitel.welcome.jar
```

Spowoduje to zwrócenie następującego wyniku:

```
com.deitel.welcome
[file:///C:/przyklady/rozdzial27/WelcomeApp/jars/com.deitel.welcome.jar]
requires java.base (@9-ea)
com.deitel.welcome -> java.base
com.deitel.welcome.Welcome -> java.io.PrintStream    java.base
com.deitel.welcome.Welcome -> java.lang.Object        java.base
com.deitel.welcome.Welcome -> java.lang.String        java.base
com.deitel.welcome.Welcome -> java.lang.System        java.base
```

W tej wersji widać wyraźnie, których pakietów, typów i modułów używa aplikacja. Wiedząc, że aplikacja wymaga tylko modułu `java.base`, możemy użyć `link` do utworzenia własnej wersji wykonawczej zawierającej tylko ten moduł (podrozdział 27.8).

Użycie narzędzia *jdeps* do wykonania plików DOT na potrzeby narzędzi do wizualizacji

Można skorzystać z narzędzi do wizualizacji grafów — np. Graphviz (www.graphviz.org) i jego internetowej wersji (www.webgraphviz.com) — aby utworzyć graf zależności modułów na podstawie języka opisu grafów DOT²¹, który definiuje węzły i krawędzie grafu. Narzędzie *jdeps* potrafi tworzyć pliki DOT (rozszerzenie `.dot`) po zastosowaniu opcji `--dot-output`:

```
jdeps --dot-output . jars/com.deitel.welcome.jar
```

Uruchomienie polecenia spowoduje utworzenie w aktualnym folderze (`.`) dwóch plików `.dot`:

- *summary.dot* — opis zależności modułu `com.deitel.welcome`;
- *com.deitel.welcome.dot* — opis konkretnych zależności pakietowych modułu `com.deitel.welcome`.

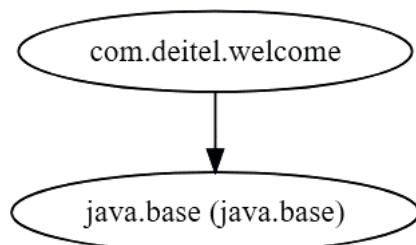
Rysunek 27.19 przedstawia graf, który uzyskaliśmy po otwarciu pliku *summary.dot* w edytorze tekstu, a następnie skopiowaniu i wklejeniu jego zawartości:

```
digraph "summary" {
    "com.deitel.welcome" -> "java.base (java.base)";
}
```

do obszaru tekstowego na stronie *webgraphviz.com* oraz kliknięciu przycisku *Generate Graph*²².

²¹ [https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

²² Rozszerzenie `.dot` jest stosowane również dla szablonów programu Microsoft Word. W systemie Windows z zainstalowanym programem Word otwieraj pliki `.dot` bezpośrednio w poziomie edytora tekstu.



Rysunek 27.19. Graf wykonany na stronie webgraphviz.com na podstawie pliku summary.dot

27.7. Zasoby w modułach — wykorzystanie modułu automatycznego

Gdy typy w module wymagają zasobów — np. obrazów, filmów, dokumentów XML itp. — zasoby te powinno się umieścić w module, aby były dostępne dla typów modułu w trakcie działania aplikacji. To tak zwana **enkapsulacja zasobów**²³. W tym podrozdziale zmigrujemy niemodułową aplikację JavaFX Video Player wykonaną w podrozdziale 22.6 do modułu, który zawiera również zasoby aplikacji — plik FXML opisujący GUI aplikacji oraz film odtwarzany po uruchomieniu aplikacji. Zgodnie z konwencją zasoby umieszcza się w folderze o nazwie *res*.

Przypomnijmy, że pierwotna wersja aplikacji z rozdziału 22. składała się z następujących plików:

- *VideoPlayer.fxml* — plik FXML opisujący GUI aplikacji;
- *VideoPlayer.java* — podklasa klasy *Application* rozpoczynająca wykonywanie aplikacji;
- *VideoPlayerController.java* — klasa kontrolera, która reaguje na zdarzenia GUI i wczytuje fil;
- *sts117.mp4* — odtwarzany film pochodzący od NASA²⁴;
- *controlsfx-8.40.12.jar* — biblioteka *ControlsFX* zawierająca klasę okna dialogowego *ExceptionHandler* (używamy jej do wyświetlenia ewentualnych błędów w trakcie odtwarzania filmu).

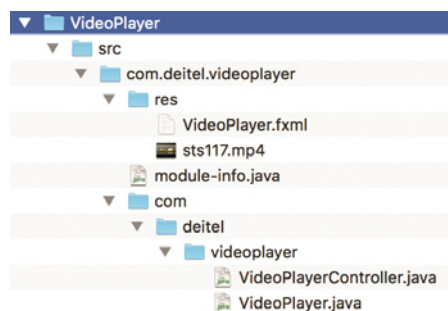
Na potrzeby tego przykładu zreorganizowaliśmy pliki w strukturę folderów przedstawioną na rysunku 27.20, aby móc poprawnie obsłużyć moduły. Zwróć uwagę na następujące elementy struktury:

- Pliki *VideoPlayer.fxml* i *sts117.mp4*, które nie są kodem źródłowym Javy, znajdują się w folderze *res*. Pliki te będą w trakcie działania aplikacji odczytywane z folderu *res*.

²³ Java Platform Module System Requirements, <http://openjdk.java.net/projects/jigsaw/spec/reqs/#resource-encapsulation>

²⁴ Aby poznać warunki wykorzystywania materiału, odwiedź stronę <http://www.nasa.gov/multimedia/guidelines/>.

- Na potrzeby modularyzacji musieliśmy umieścić klasy `VideoPlayer` i `VideoPlayerController` w pakiecie — struktura folderów `com/deitel/videooplayer` odpowiada pakietowi `com.deitel.videooplayer`.
- Zgodnie z wymaganiami utworzyliśmy w głównym folderze modułu plik `module-info.java`.



Rysunek 27.20. Struktura folderów odtwarzacza wideo wymagana przez modularyzację

Dodatkowo zmieniliśmy nazwę pliku `controlsfx-8.40.12.jar` na `controlsfx.jar`, a sam plik przenieśliśmy do podfolderu `mods` folderu `VideoPlayer`.

27.7.1. Moduły automatyczne

Biblioteka ControlsFX wykorzystywana w podrozdziale 22.6 nie była projektowana do użycia jako moduł Javy. Możemy jednak dodać **dowolny** plik biblioteki JAR do ścieżki modułów, a następnie korzystać bez przeszkód z pakietów zawartych w pliku JAR. W takiej sytuacji plik JAR staje się niejawnie **modułem automatycznym** i może być wskazany w dyrektywie `requires`. Nazwa pliku JAR — po odjęciu rozszerzenia `.jar` — staje się nazwą modułu, więc **musi być poprawnym identyfikatorem Javy**, aby mogła się znaleźć w dyrektywie `requires`. Właśnie z tego powodu usunęliśmy z nazwy pliku fragment `-8.40.12`. Dodatkowo moduł automatyczny:

- **niejawnie eksportuje** wszystkie swoje pakiety — dowolny moduł czytający moduł automatyczny ma dostęp do typów publicznych pakietów modułu automatycznego;
- **niejawnie czyta** wszystkie inne moduły, w tym inne moduły automatyczne i moduł nienazwany — moduł automatyczny ma więc dostęp do wszystkich publicznych typów udostępnianych przez inne moduły systemu.

Zmiany w kodzie związane z modularyzacją

Dokonaaliśmy następujących zmian w kodzie:

- Plik `VideoPlayer.fxml` — zmieniliśmy nazwę klasy kontrolera tak, aby zawierała pełną nazwę kwalifikowaną (`com.deitel.videooplayer.VideoPlayerController`), dzięki czemu klasa `FXMLLoader` będzie potrafiła bez przeszkód odnaleźć klasę kontrolera.
- Plik `VideoPlayer.java` — zmieniliśmy nazwę wczytywanego pliku FXML z `"VideoPlayer.fxml"` na `"/res/VideoPlayer.fxml"`, co wskazuje, że plik

FXML znajduje się teraz w folderze *res*. Dodaliśmy również instrukcje package w postaci:

```
package com.deitel.videoplayer;
```

- Plik *VideoPlayerController.java* — zmieniliśmy nazwę wczytywanego pliku *video* z "sts117.mp4" na "/res/sts117.mp4", co wskazuje, że plik znajduje się teraz w folderze *res*. Dodaliśmy również instrukcje package w postaci:

```
package com.deitel.videoplayer;
```

Pozostała część kodu jest dokładnie taka sama jak w podrozdziale 22.6.

27.7.2. Wymaganie kilku modułów

Deklaracja modułu *com.deitel.videoplayer* (rysunek 27.21) wskazuje, że moduł wymaga modułów *javafx.controls*, *javafx.fxml*, *javafx.media* i *controlsfx* (moduł automatyczny omawiany w punkcie 27.7.1). Moduł eksportuje pakiet *com.deitel.videoplayer* (wiersz 9.), ponieważ klasa *VideoPlayerController* jest używana przez klasę *FXMLLoader* (moduł *javafx.fxml*), gdy tworzy obiekt kontrolera i interfejs graficzny aplikacji.

```
1 // Rysunek 27.21. module-info.java
2 // Deklaracja modułu com.deitel.videoplayer
3 module com.deitel.videoplayer {
4     requires javafx.controls;
5     requires javafx.fxml;
6     requires javafx.media;
7     requires controlsfx; // Moduł automatyczny dla ControlsFX
8
9     exports com.deitel.videoplayer;
10    opens com.deitel.videoplayer to javafx.fxml;
11 }
```

Rysunek 27.21. Deklaracja modułu *com.deitel.videoplayer*

27.7.3. Otwarcie modułu na potrzeby mechanizmu refleksji

Na rysunku 27.21 dyrektywa *opens...to* (wiersz 10.) wskazuje, że istniejące w pakiecie *com.deitel.videoplayer* typy powinny być dostępne dla mechanizmu refleksji wykorzystywanego przez typy znajdujące się w module *javafx.fxml*. Jak wcześniej wspomnieliśmy, pozwoli to klasie *FXMLLoader* znaleźć i wczytać klasę *VideoPlayerController*. Klasa *FXMLLoader* tworzy obiekt *VideoPlayerController* i **wstrzykuje** do niego referencje do komponentów GUI zdefiniowanych w pliku FXML. Aby jeden moduł mógł **otworzyć** pakiet dla innego modułu, pakiet ten musi zostać najpierw wyeksportowany (może to być eksport kwalifikowany wykorzystujący *exports...to*).

27.7.4. Graf zależności modułu

Rysunek 27.22 przedstawia graf zależności modułu *com.deitel.videoplayer*. Elementy z wąskim niebieskim tłem na dole są modułami wymaganymi jawnie przez dyrektywy *requires* (poza modulem *java.base*, który jest wymagany jawnie przez wszystkie moduły). Pozostałe wyświetlone moduły to zależności modułów wskazanych przez dyrektywy *requires*.



Rysunek 27.22. Graf zależności modułu `com.deitel.videoplayer`

27.7.5. Kompilacja modułu

Aby skompilować moduł `com.deitel.videoplayer`, wpisz:

```
javac --module-path mods -d mods/com.deitel.videoplayer ^
src/com.deitel.videoplayer/module-info.java ^
src/com.deitel.videoplayer/com/deitel/videoplayer/*.java
```

Zauważ, że dodaliśmy opcję `--module-path`, ponieważ folder `mods` zawiera plik `controlsfx.jar` — moduł automatyczny wymagany do skompilowania aplikacji.

Skopiowanie plików zasobów do modułu

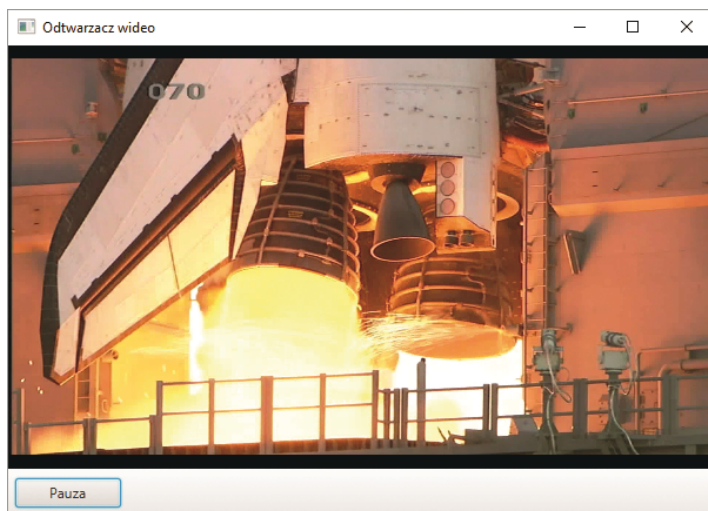
Niektóre IDE i narzędzia do budowania automatycznie umieszczają zasoby modułu w skompilowanym module. Polecenie `javac` tego nie czyni. Po skompilowaniu modułu skopiuj folder `res` z folderu `src/com.deitel.videoplayer` do folderu `mods/com.deitel.videoplayer`.

27.7.6. Uruchomienie aplikacji po modularyzacji

Aby uruchomić klasę `VideoPlayer` z modułu `com.deitel.videoplayer`, wpisz:

```
java --module-path mods ^
-m com.deitel.videoplayer/com.deitel.videoplayer.VideoPlayer
```

Rysunek 27.23 przedstawia aplikację działającą w systemie Windows.



Rysunek 27.23. Działający odtwarzacz wideo po modularyzacji

27.8. Tworzenie własnych systemów wykonawczych narzędziem `jlink`

Nowe narzędzie JDK 9 w postaci polecenia `jlink` pozwala tworzyć własne obrazy systemów wykonawczych²⁵. We własnym systemie wykonawczym można osadzić tylko te elementy, które są faktycznie niezbędne do prawidłowego działania aplikacji. Jeśli tworzymy system wykonawczy dla urządzenia, które nie obsługuje interfejsu graficznego, nie musimy dołączać modułów związanych z JavaFX i Swingiem. W zasadzie większość przykładów z tej książki działających na poziomie konsoli wymaga do prawidłowego działania tylko i wyłącznie modułu `java.base`.

27.8.1. Wyświetlenie listy modułów JRE

Zmodularyzowane JRE to podzbiór modułów JDK²⁶. Wykonanie polecenia:

```
java --list-modules
```

z poziomu folderu `bin` z JRE spowoduje wyświetlenie tylko 73 modułów (rysunek 27.24) zamiast pełnej listy 95 modułów JDK. Oczywiście liczba modułów będzie się zmieniała wraz z rozwojem Javy. W punkcie 27.8.3 budujemy poleceniem `jlink` własny system wykonawczy — osadzimy w nim tylko jeden moduł podstawowy.

²⁵ Jean-Francois Denise, „JEP 282: `jlink`: The Java Linker”, <http://openjdk.java.net/jeps/282>

²⁶ Brian Goetz poinformował o tym autorów książki w e-mailu przesłanym 20 marca 2017 roku.

java.activation@9-ea java.base@9-ea java.compiler@9-ea java.corba@9-ea java.datatransfer@9-ea java.desktop@9-ea java.instrument@9-ea java.jnlp@9-ea java.logging@9-ea java.management@9-ea java.management.rmi@9-ea java.naming@9-ea java.prefs@9-ea java.rmi@9-ea java.scripting@9-ea java.se@9-ea java.se.ee@9-ea java.security.jgss@9-ea java.security.sasl@9-ea java.smartcardio@9-ea java.sql@9-ea java.sql.rowset@9-ea java.transaction@9-ea java.xml@9-ea java.xml.bind@9-ea java.xml.crypto@9-ea java.xml.ws@9-ea java.xml.ws.annotation@9-ea javafx.base@9-ea javafx.controls@9-ea javafx.deploy@9-ea javafx.fxml@9-ea javafx.graphics@9-ea javafx.media@9-ea javafx.swing@9-ea javafx.web@9-ea jdk.accessibility@9-ea	jdk.charsets@9-ea jdk.crypto.cryptoki@9-ea jdk.crypto.ec@9-ea jdk.crypto.mscapi@9-ea jdk.deploy@9-ea jdk.deploy.controlpanel@9-ea jdk.dynalink@9-ea jdk.httpserver@9-ea jdk.incubator.httpclient@9-ea jdk.internal.le@9-ea jdk.internal.vm.ci@9-ea jdk.javaws@9-ea jdk.jdwp.agent@9-ea jdk.jfr@9-ea jdk.jsobject@9-ea jdk.localedata@9-ea jdk.management@9-ea jdk.management.agent@9-ea jdk.naming.dns@9-ea jdk.naming.rmi@9-ea jdk.net@9-ea jdk.pack@9-ea jdk.plugin@9-ea jdk.plugin.dom@9-ea jdk.plugin.server@9-ea jdk.scripting.nashorn@9-ea jdk.scripting.nashorn.shell@9-ea jdk.sctp@9-ea jdk.security.auth@9-ea jdk.security.jgss@9-ea jdk.snmp@9-ea jdk.unsupported@9-ea jdk.xml.dom@9-ea jdk.zipfs@9-ea oracle.desktop@9-ea oracle.net@9-ea
---	---

Rysunek 27.24. Wynik działania polecenia `java --list-modules` na poziomie JRE



27.3. Obserwacja z poziomu inżynierii oprogramowania

Można użyć modułowej platformy Java do wygodnego budowania własnych systemów wykonawczych dla urządzeń o mniejszych możliwościach lub mniejszej pojemności.

27.8.2. Własny system wykonawczy zawierający tylko moduł `java.base`

Na potrzeby tego punktu przejdź do folderu *WelcomeApp* — po utworzeniu własnego systemu wykonawczego wykonamy w nim aplikację powitalną. Poniższe polecenie tworzy system wykonawczy składający się tylko z modułu `java.base`:

```
jlink --module-path "%JAVA_HOME%/jmods --add-modules java.base ^
--output javabaseruntime
```

Opcje polecenia działają następująco:

- `--module-path` określa co najmniej jeden folder z modułami, które mają się znaleźć w systemie wykonawczym — w tym przypadku jest to folder *jmods* zawierający modułowe pliki JAR dla wszystkich modułów JDK;
- `--add-modules` wskazuje moduły, które należy dołączyć do systemu wykonawczego — w tym przypadku jest to tylko moduł *java.base*;
- `--output` określa folder, w którym znajdzie się system wykonawczy — w tym przypadku jest to folder *javabaseruntime*; folder ten znajdzie się wewnątrz folderu, z poziomu którego było uruchamiane polecenie (o ile nie wskażemy innej ścieżki); jeśli folder już istnieje, narzędzie zgłosi błąd.

Ten system wykonawczy może uruchamiać tylko aplikacje wymagające do poprawnej pracy typów z pakietów zawartych w module *java.base*. Wiele aplikacji wiersza poleceń prezentowanych w książce wymaga tylko i wyłącznie tego modułu.

Uwaga dotycząca zmiennej środowiskowej JAVA_HOME

Zmienna środowiskowa *JAVA_HOME* musi odnosić się do folderu instalacyjnego JDK 9 w systemie operacyjnym. Zatrzyj do informacji znajdujących się na początku książki, aby prawidłowo ustawić zmienną. W systemie Windows użycie wartości zmiennej *JAVA_HOME* odbywa się za pomocą składni `%JAVA_HOME%`. W systemach Linux i macOS użytkownicy powinni zastąpić `%JAVA_HOME%` wersją `$JAVA_HOME`. W systemach tych polecenie będzie miało postać:

```
jlink --module-path "$JAVA_HOME"/jmods --add-modules java.base \
--output javabaseruntime
```

We wszystkich systemach użyj cudzysłowów ("`"`), jeśli ścieżka zawiera znaki spacji.

Uruchomienie aplikacji powitalnej we własnym systemie wykonawczym

Aby uruchomić aplikację powitalną we własnym systemie wykonawczym w Windowsie, użyj polecenia:

```
javabaseruntime\bin\java --module-path mods ^
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

W systemach Linux i macOS użyj polecenia:

```
javabaseruntime/bin/java --module-path mods \
--module com.deitel.welcome/com.deitel.welcome.Welcome
```

Program wykona się i wyświetli tekst:

```
Witamy w systemie modułów platformy Java!
```

Wyświetlenie listy modułów własnego systemu wykonawczego

Wcześniej użyliśmy polecenia:

```
java --list-modules
```

do wyświetlenia listy wszystkich modułów JDK. Gdy już mamy własny system wykonawczy, korzystamy z polecenia *java* znajdującego się w folderze *bin*, aby potwierdzić dostępną listę modułów:

```
javabaseruntime\bin\java --list-modules
```

Przy uruchamianiu polecenia w systemie Windows rozdziel foldery lewym ukośnikiem (\). W przypadku systemów macOS i Linux oddziel foldery prawym ukośnikiem (/). Powyższe polecenie spowoduje wyświetlenie następującego wyniku:

```
java.base@9-ea
```

Poniższe polecenie utworzy własny system wykonawczy zawierający tylko moduł `java.desktop` i wszystkie inne moduły, od których on zależy:

```
jlink --module-path "%JAVA_HOME%/jmods ^  
--add-modules java.desktop --output javadesktopruntime
```

Tym razem wykonanie polecenia:

```
javadesktopruntime\bin\java --list-modules
```

(zastosuj prawe ukośniki w systemach macOS i Linux) spowoduje zwrócenie następujących modułów:

```
java.base@9-ea  
java.datatransfer@9-ea  
java.desktop@9-ea  
java.prefs@9-ea  
java.xml@9-ea
```

27.8.3. Tworzenie własnego systemu wykonawczego dla aplikacji powitalnej

Aby utworzyć własny system wykonawczy dla modułu `com.deitel.welcome` wraz ze wszystkimi jego zależnościami (w tym przypadku tylko moduł `java.base`), użyj polecenia:

```
jlink --module-path jars;"%JAVA_HOME%/jmods ^  
--add-modules com.deitel.welcome --output welcomeruntime
```

Spowoduje to utworzenie własnego systemu wykonawczego w folderze *welcomeruntime*. Powyższe polecenie wskazuje kilka folderów — `jars` i ścieżkę ze zmiennej środowiskowej `%JAVA_HOME%`. W systemie Windows do rozdzielania znaków stosuje się znak średnika (;). W systemach Linux i macOS zastąp średniki znakami dwukropka (:), czyli użyj polecenia:

```
jlink --module-path jars:"$JAVA_HOME"/jmods \  
--add-modules com.deitel.welcome --output welcomeruntime
```

Aby zobaczyć listę modułów dołączonych do takiej wersji systemu wykonawczego, użyj w systemie Windows polecenia:

```
welcomeruntime\bin\java --list-modules
```

(zastosuj prawe ukośniki w systemach macOS i Linux), które zwróci następującą listę modułów:

```
com.deitel.welcome  
java.base@9-ea
```

27.8.4. Wykonywanie aplikacji powitalnej we własnym systemie wykonawczym

Aby uruchomić aplikację we własnym systemie wykonawczym, użyj w systemie Windows polecenia:

```
welcomeruntime\bin\java -m com.deitel.welcome
```

(zastosuj prawe ukośniki w systemach macOS i Linux). Program wykona się i wyświetli tekst:

```
Witamy w systemie modułów platformy Java!
```

27.8.5. Użycie mechanizmu rozwiązywania modułów z własnym systemem wykonawczym

Gdy uruchamia się aplikację modułową, JVM używa **mechanizmu rozwiązywania modułów**, aby określić, które moduły są wymagane na etapie działania programu i czy wszystkie zależności są spełnione — operację tę nazywa się **domknięciem przejściowym**. Aby odnaleźć moduły, mechanizm korzysta z modułów obserwowalnych, czyli wbudowanych w system wykonawczy (np. `java.base`) i dostępnych w ścieżce. Jeśli wymaganego modułu nie uda się odnaleźć, system wykonawczy zgłosi wyjątek `java.lang.module.FindException`.

Dla dowolnej aplikacji i systemu wykonawczego można zobaczyć kroki, których używa mechanizm rozwiązywania modułów, aby określić zależności modułu i upewnić się, że wszystkie wymagane elementy są dostępne. W tym celu użyj opcji **-Xdiag:resolver**²⁷ polecenia `java`:

```
welcomeruntime\bin\java -Xdiag:resolver -m com.deitel.welcome
```

(zastosuj prawe ukośniki w systemach macOS i Linux). W przykładzie używamy własnego systemu wykonawczego, który wyświetli kroki mechanizmu rozwiązywania modułów, a następnie wynik działania programu:

```
[Resolver] Root module com.deitel.welcome located
[Resolver] (jrt:/com.deitel.welcome)
[Resolver] Module java.base located, required by com.deitel.welcome
[Resolver] (jrt:/java.base)
[Resolver] Result:
[Resolver] com.deitel.welcome
[Resolver] java.base
Witamy w systemie modułów platformy Java!
```

Proces rozwiązywania modułów w przypadku aplikacji powitalnej jest następujący:

1. Najpierw mechanizm znajduje **moduł początkowy** — `com.deitel.welcome` — zawierający punkt wejścia do aplikacji. Mechanizm nazywa ten moduł **korzeniem**. Jest to korzeń grafu zależności modułu.
2. Następnie mechanizm poszukuje modułu `java.base`, bo moduł `com.deitel.welcome` wskazuje go w deskrytorze jako swoją zależność.

²⁷ Alan Bateman, Alex Buckley, Jonathan Gibbons i Mark Reinhold, „JEP 261: Module System”, <http://openjdk.java.net/jeps/261>

3. Ponieważ `java.base` nie zależy od innych modułów, graf zależności kończy się, a mechanizm wyświetla listę modułów niezbędną do działania programu.

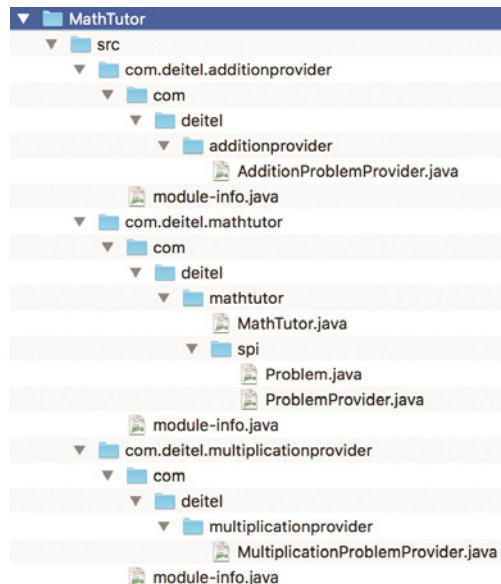
Następnie program uruchamia się i wyświetla wyniki. Jeśli w trakcie procesu nie znaleziono wymaganego modułu, zamiast wyników programu pojawi się wyjątek `java.lang.module.FindException`.

27.9. Usługi i klasa `ServiceLoader`

W podrozdziale 10.3 omawialiśmy „programowanie do interfejsu, a nie do implementacji” jako mechanizm zapewniania luźno powiązanych obiektów. Użyjemy tych koncepcji w tym podrozdziale, omawiając usługę klasy `ServiceLoader`, która pomaga w tworzeniu luźno powiązanych komponentów systemowych. W ten sposób znacznie łatwiej tworzy się i utrzymuje duże systemy.

Aplikacja MathTutor

Wykonamy aplikację `MathTutor` (składającą się z trzech modułów), która obsługuje różne rodzaje losowo generowanych zadań matematycznych. Zamiast umieszczać je na sztywno w aplikacji, będziemy wczytywać zadania matematyczne poprzez **interfejs dostawcy usług**, który opisuje, jak pobrać zadanie. Następnie zdefiniujemy dwóch **dostawców usług** — klasy implementujące ten interfejs. Jeden dostawca dostarczać będzie zadania związane z dodawaniem, a drugi z mnożeniem. W trakcie działania programu wczytamy implementacje dostawców i z nich skorzystamy. Pełna struktura aplikacji składa się z trzech modułów przedstawionych na rysunku 27.25.



Rysunek 27.25. Moduły aplikacji `MathTutor`

Moduły aplikacji MathTutor

Moduł `com.deitel.mathtutor` zawiera dwa powiązane pakiety:

- Pakiet `com.deitel.mathtutor`, który zawiera klasę `MathTutor` — aplikację wiersza poleceń, która wyświetla losowe zadanie matematyczne, prosi o wpisanie odpowiedzi i wyświetla informację o jej poprawności.
- Pakiet `com.deitel.mathtutor.spi`, który zawiera interfejs dostawcy usług `ProblemProvider` i pomocniczą klasę abstrakcyjną `Problem` reprezentującą zadanie matematyczne. Klasa `MathTutor` używa `ProblemProvider` do otrzymywania obiektów `Problem`.

Moduł `com.deitel.additionprovider` zawiera pakiet o takiej samej nazwie, w którym znajduje się klasa `AdditionProblemProvider`. Ta implementacja interfejsu dostawcy usług `ProblemProvider` generuje losowe obiekty `Problem` dotyczące dodawania.

Moduł `com.deitel.multiplicationprovider` zawiera pakiet o takiej samej nazwie, w którym znajduje się klasa `MultiplicationProblemProvider`. Ta implementacja interfejsu dostawcy usług `ProblemProvider` generuje losowe obiekty `Problem` dotyczące mnożenia.

W jaki sposób zademonstrujemy aplikację?

Początkowo uruchomimy aplikację `MathTutor` bez umieszczania modułów implementujących dostawców usług w ścieżce modułów, aby pokazać, co się stanie, jeśli **brakuje** dostawców usług w trakcie działania aplikacji. Następnie „wepniemy” moduł `com.deitel.additionprovider` do ścieżki klas i ponownie uruchomimy aplikację, aby pokazać, że potrafi zgłaszać do rozwiązania zadania generowane przez `AdditionProblemProvider`. Na końcu „wepniemy” oba moduły, `com.deitel.additionprovider` i `com.deitel.multiplicationprovider`, aby pokazać, że jesteśmy w stanie pobierać obiekty `Problem` generowane przez `AdditionProblemProvider` i `MultiplicationProblemProvider`.

Architektura wtyczek

Architektura wtyczek (pluginów) wykonana przy użyciu interfejsu dostawcy usług i implementacji tego interfejsu ułatwia rozszerzanie aplikacji. Wystarczy utworzyć moduł zawierający implementację `ProblemProvider` i dodać go do ścieżki modułów, gdy uruchamia się aplikację. Aplikacja jest też bardziej konfigurowalna, bo można wybrać, które moduły mają się znaleźć w ścieżce, gdy uruchamiamy program.

Pewność konfiguracji

Mechanizm tworzenia luźno powiązanych systemów takich jak aplikacja `MathTutor` był stosowany w zasadzie od momentu powstania języka Java. Nowym kluczowym pojęciem w Javie 9, które ma zastosowanie również do modułów, jest **pewność konfiguracji**. Aby prezentowana aplikacja mogła wyświetlać zadania matematyczne do rozwiązania, musi być w stanie zlokalizować i wczytać implementacje `ProblemProvider`. Deklaracje modułów pozwalają wskazać, których interfejsów dostawców usług używa moduł, a także czy moduł dostarcza jakieś implementacje tych interfejsów.

27.9.1. Interfejs dostawcy usług

Pakiet `com.deitel.mathtutor.spi` zawiera interfejs dostawcy usług `ProblemProvider` modułu `com.deitel.mathtutor` i pomocniczą klasę abstrakcyjną `Problem`. Ostatni komponent nazwy pakietu — `spi` — jest często stosowany w pakietach, które deklarują jeden lub kilka interfejsów dostawców usług. Interfejs `ProblemProvider` (rysunek 27.26) deklaruje metodę `getProblem` (wiersz 6.), która zwraca `Problem`.

```

1 // Rysunek 27.26. ProblemProvider.java
2 // Interfejs dostawcy usług pozwalający pobierać obiekty Problem
3 package com.deitel.mathtutor.spi;
4
5 public interface ProblemProvider {
6     public Problem getProblem();
7 }

```

Rysunek 27.26. Interfejs dostawcy usług pozwalający pobierać obiekty `Problem`

Abstrakcyjna klasa `Problem` (rysunek 27.27) zapewnia typowe funkcjonalności dotyczące zadań matematycznych. Każde zadanie wykorzystuje dwa operandy typu `int` i wynik typu `int`, a także tekstową reprezentację operacji — aplikacja wyświetla ten tekst przy każdej operacji. Metoda abstrakcyjna `getResult` jest przesłaniana w każdej konkretnej podklasie abstrakcyjnej klasy `Problem`.

```

1 // Rysunek 27.27. Problem.java
2 // Klasa nadrzędna Problem zawierająca informacje na temat zadania matematycznego
3 package com.deitel.mathtutor.spi;
4
5 public abstract class Problem {
6     private int leftOperand;
7     private int rightOperand;
8     private int result;
9     private String operation;
10
11     // Konstruktor
12     public Problem(int leftOperand, int rightOperand, String operation) {
13         this.leftOperand = leftOperand;
14         this.rightOperand = rightOperand;
15         this.operation = operation;
16     }
17
18     // Pobiera lewy operand
19     public int getLeftOperand() {return leftOperand;}
20
21     // Pobiera prawy operand
22     public int getRightOperand() {return rightOperand;}
23
24     // Pobiera operację
25     public String getOperation() {return operation;}
26
27     // Pobiera wynik
28     public abstract int getResult();
29 }

```

Rysunek 27.27. Klasa nadrzędna `Problem` zawierająca informacje na temat zadania matematycznego

27.9.2. Wczytywanie i użycie dostawców usług

Klasa MathTutor (rysunek 27.28) to punkt wejścia do aplikacji. Zawiera logikę związaną z lokalizacją i wczytaniem implementacji ProblemProvider, a następnie wykorzystaniem ich do przedstawienia zadań matematycznych użytkownikowi.

```

1 // Rysunek 27.28. MathTutor.java
2 // Aplikacja wykorzystująca interfejs ProblemProvider do wyświetlenia zadań
   ↪ matematycznych
3 package com.deitel.mathtutor;
4
5 import java.util.List;
6 import java.util.Random;
7 import java.util.Scanner;
8 import java.util.ServiceLoader;
9 import java.util.ServiceLoader.Provider;
10 import java.util.stream.Collectors;
11 import com.deitel.mathtutor.spi.Problem;
12 import com.deitel.mathtutor.spi.ProblemProvider;
13
14 public class MathTutor {
15     private static Scanner input = new Scanner(System.in);
16
17     public static void main(String[] args) {
18         // Pobierz mechanizm wczytywania usług dla ProblemProvider
19         ServiceLoader<ProblemProvider> serviceLoader =
20             ServiceLoader.load(ProblemProvider.class);
21
22         // Pobierz listę dostawców usług
23         List<Provider<ProblemProvider>> providersList =
24             serviceLoader.stream().collect(Collectors.toList());
25
26         // Sprawdź, czy istnieją jacyś dostawcy
27         if (providersList.isEmpty()) {
28             System.out.println(
29                 "Wyłączenie MathTutor - nie zaleziono dostawców zadań.");
30             return;
31         }
32
33         boolean shouldContinue = true;
34         Random random = new Random();
35
36         do {
37             // Losowo wybierz wersję ProblemProvider
38             ProblemProvider provider =
39                 providersList.get(random.nextInt(providersList.size())).get();
40
41             // Pobierz obiekt Problem
42             Problem problem = provider.getProblem();
43
44             // Wyświetl zadanie użytkownikowi
45             showProblem(problem);
46         } while (playAgain());
47     }
48
49     // Wyświetl użytkownikowi zadanie do rozwiązania

```

Rysunek 27.28. Aplikacja wykorzystująca interfejs ProblemProvider do wyświetlenia zadań matematycznych

```

50 private static void showProblem(Problem problem) {
51     String problemStatement = String.format("Ile wynosi %d %s %d? ",
52         problem.getLeftOperand(), problem.getOperation(),
53         problem.getRightOperand());
54
55     // Wyświetl zadanie i zaczekaj na odpowiedź
56     System.out.printf(problemStatement);
57     int answer = input.nextInt();
58
59     while (answer != problem.getResult()) {
60         System.out.println("Odpowiedź nieprawidłowa. Spróbuj ponownie: ");
61         System.out.printf(problemStatement);
62         answer = input.nextInt();
63     }
64
65     System.out.println("Dobrze!");
66 }
67
68 // Następne zadanie?
69 private static boolean playAgain() {
70     System.out.printf("Następne zadanie? t, by kontynuować, n, by
71     ↪przerwać: ");
72     String response = input.next();
73
74     return response.toLowerCase().startsWith("t");
75 }

```

Rysunek 27.28. Aplikacja wykorzystująca interfejs `ProblemProvider` do wyświetlenia zadań matematycznych — ciąg dalszy

Użycie klasy `ServiceLoader` do znalezienia dostawców usług

Wiersze 19. i 20.:

```

ServiceLoader<ProblemProvider> serviceLoader =
    ServiceLoader.load(ProblemProvider.class);

```

tworzą obiekt **ServiceLoader** (pakiet `java.util`), który wczytuje implementacje `ProblemProvider`. Metoda statyczna **load** klasy `ServiceLoader` otrzymuje jako argument obiekt `Class` reprezentujący typ interfejsu dostawcy usług — zapis `Problem ↪Provider.class` to **literal** klasy równoważny utworzeniu obiektu `Class<Problem ↪Provider>` w następujący sposób:

```
new Class<ProblemProvider>()
```

Metoda `load` zwraca `ServiceLoader<ProblemProvider>`, który wie, jak wczytywać tylko i wyłącznie implementacje `ProblemProvider`.

Istnieje kilka sposobów pobierania implementacji z obiektu `ServiceLoader`. W wierszach 23. i 24.:

```

List<Provider<ProblemProvider>> providersList =
    serviceLoader.stream().collect(Collectors.toList());

```

pobieramy listę dostępnych implementacji dostawców usług, używając metody **stream**. Zwrócony zostaje obiekt `Stream<Provider<ProblemProvider>>` reprezentujący wszystkie dostępne implementacje `ProblemProvider`, o ile istnieją. Interfejs `Provider` (importowany w wierszu 9.) to zagnieżdżony typ klasy `ServiceLoader`.

Dla każdej dostępnej implementacji `ProblemProvider` strumień zawiera jeden obiekt `Provider<ProblemProvider>`. Wiersz 24. używa metody `collect` ze `Stream` i predefiniowanego obiektu `Collector` definiowanego przez `Collectors.toList`, aby pobrać obiekt `List` zawierający wszystkie dostępne implementacje. Jeśli lista jest pusta (wiersz 27.), program wyświetla stosowny komunikat i kończy działanie.

Korzystanie z interfejsu dostawcy usług

Jeśli obiekt `List` zawiera choć jedną implementację dostawcy usług, wiersze od 36. do 46. używają jej do wyświetlania użytkownikowi po jednym zadaniu matematycznym do rozwiązywania. Wiersze 38. i 39.:

```
ProblemProvider provider =
    providersList.get(random.nextInt(providersList.size())).get();
```

losowo wybierają jeden obiekt `Provider<ProblemProvider>` z `providersList`, a następnie wywołują metodę `get`, aby pobrać obiekt `ProblemProvider`. Wiersz 42.:

```
Problem problem = provider.getProblem();
```

pobiera obiekt `Problem` z wybranego obiektu `ProblemProvider`.

Zwróć uwagę na **luźne powiązanie** aplikacji `MathTutor` i obiektów `Problem` ➔ `Provider`. Aplikacja w żaden sposób nie odnosi się do klas `AdditionProblem` ➔ `Provider` i `MultiplicationProblemProvider` odpowiedzialnych za generowanie zadań matematycznych.

27.9.3. Dyrektywa `uses` modułu i konsumpcja usług

Rysunek 27.29 przedstawia deklarację modułu `com.deitel.mathtutor`. Zwróć uwagę, że ten moduł eksportuje pakiet `com.deitel.mathtutor.spi` zawierający interfejs dostawcy usług `ProblemProvider` i pomocniczą klasę `Problem`. Dzięki temu moduły implementujące interfejs `ProblemProvider` mają dostęp do tych typów. Nową funkcjonalnością w przedstawionej deklaracji jest **dyrektywa `uses`** (wiersz 6.). Dyrektywa ta wskazuje, że istnieje w module `com.deitel.mathtutor` typ **używający** obiektów implementujących interfejs `ProblemProvider`. Taki moduł nazywamy **konsumentem usług**.

```
1 // Rysunek 27.29. module-info.java
2 // Deklaracja modułu com.deitel.mathtutor
3 module com.deitel.mathtutor {
4     exports com.deitel.mathtutor.spi; // Pakiet z interfejsem dostawcy
5
6     uses com.deitel.mathtutor.spi.ProblemProvider;
7 }
```

Rysunek 27.29. Deklaracja modułu `com.deitel.mathtutor`

Aby móc konsumować obiekty `ProblemProvider`, obiekt `ServiceLoader` musi być w stanie znaleźć i dynamicznie odczytać ich implementacje za pomocą mechanizmu **refleksji** Javy. Gdy uruchamiamy aplikację, mechanizm rozwiązywania modułów znajdzie w deskrytorze informację, że moduł używa implementacji `ProblemProvider`, a tym samym jest zależny od jej dostawców. Poszuka więc w ścieżce modułów jakichkolwiek modułów dostarczających implementacje tego interfejsu. Jeśli znajdzie taki moduł, doda go do grafu zależności.

27.9.4. Uruchomienie aplikacji bez dostawców usług

Aby skompilować moduł `com.deitel.mathtutor`, wpisz:

```
javac -d mods/com.deitel.mathtutor ^
src/com.deitel.mathtutor/module-info.java ^
src/com.deitel.mathtutor/com/deitel/mathtutor/MathTutor.java ^
src/com.deitel.mathtutor/com/deitel/mathtutor/spi/*.java
```

Następnie uruchom aplikację bez wskazywania implementacji `ProblemProvider`, wywołując poniższe polecenie `java`, które w ścieżce modułów ma tylko moduł `com.deitel.mathtutor`:

```
java --module-path mods/com.deitel.mathtutor ^
-m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

Wynikiem będzie komunikat:

```
Wyłączenie MathTutor - nie zaleziono dostawców zadań.
```

27.9.5. Implementacja dostawcy usług

Wykonajmy teraz klasę `AdditionProblemProvider` (rysunek 27.30), która implementuje interfejs dostawcy usług `ProblemProvider` (wiersz 10.). Pakiet `com.deitel.additionprovider` zawierający klasę umieścimy w module `com.deitel.additionprovider` (punkt 27.9.6). Importujemy interfejs `ProblemProvider` i klasę `Problem` z pakietu `com.deitel.mathtutor.spi` (wiersze 7. i 8.) modułu `com.deitel.mathtutor`. Gdy `MathTutor` wywoła metodę `getProblem` z `AdditionProblemProvider` (wiersze od 14. do 23.), metoda tworzy anonimową podklasę `Problem` (wiersze od 16. do 22.), przekazując konstruktorowi obiektów `Problem` dwie losowe wartości typu `int` jako operandy i tekst `+` jako operację. Wiersze od 18. do 21. przesłaniają metodę `getResult` klasy nadrzędnej, aby zwrócić sumę lewego i prawego operandu.

```
1 // Rysunek 27.30. AdditionProblemProvider.java
2 // Implementacja klasy AdditionProblemProvider interfejsu
3 // ProblemProvider dla aplikacji MathTutor
4 package com.deitel.additionprovider;
5
6 import java.util.Random;
7 import com.deitel.mathtutor.spi.Problem;
8 import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class AdditionProblemProvider implements ProblemProvider {
11     private static Random random = new Random();
12
13     // Zwraca nowe zadanie z dodawania
14     @Override
15     public Problem getProblem() {
16         return new Problem(random.nextInt(10), random.nextInt(10), "+") {
17             // Przesłania getResult, aby obsłużyć operację dodawania
18             @Override
19             public int getResult() {
20                 return getLeftOperand() + getRightOperand();
21             }
22         };
23     }
24 }
```

Rysunek 27.30. Implementacja klasy `AdditionProblemProvider` interfejsu `ProblemProvider` dla aplikacji `MathTutor`

27.9.6. Dyrektywa provides...with modułu i deklaracja dostawcy usług

Rysunek 27.31 przedstawia deklarację modułu `com.deitel.additionprovider`. Zwróć uwagę, że moduł ten wymaga modułu `com.deitel.mathtutor`. Przypomnijmy, że na rysunku 27.29 moduł ten eksportował pakiet `com.deitel.mathtutor.spi` zawierający typy wykorzystywane przez klasę `AdditionProblemProvider`. Nowym elementem w deklaracji jest **dyrektywa provides...with**. Wiersze 6. i 7. wskazują, że:

- moduł ten zapewnia implementację interfejsu `ProblemProvider` zadeklarowaną w pakiecie `com.deitel.mathtutor.spi`. `ProblemProvider` modułu `com.deitel.mathtutor`;
- implementację tę zapewnia klasa `AdditionProblemProvider` zadeklarowana w pakiecie `com.deitel.additionprovider` aktualnego modułu.

```

1 // Rysunek 27.31. module-info.java
2 // Deklaracja modułu com.deitel.additionprovider
3 module com.deitel.additionprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider
7         with com.deitel.additionprovider.AdditionProblemProvider;
8 }

```

Rysunek 27.31. Deklaracja modułu `com.deitel.additionprovider`

Tego rodzaju moduł nazywamy **dostawcą usług**. Po części `provides` pojawia się nazwa interfejsu lub klasy abstrakcyjnej, która została wskazana w dyrektywie `uses` w innym module. Część `with` określa nazwę klasy implementującej interfejs lub rozszerzającej klasę abstrakcyjną.

27.9.7. Uruchomienie aplikacji z jednym dostawcą usług

Zanim jednak uruchomimy aplikację, umieszczając `AdditionProblemProvider` w ścieżce klas, najpierw skompilujemy `com.deitel.additionprovider`, używając następującego polecenia:

```

javac --module-path mods -d mods/com.deitel.additionprovider ^
    src/com.deitel.additionprovider/module-info.java ^
    src/com.deitel.additionprovider/com/deitel/additionprovider/ ^
    AdditionProblemProvider.java

```

Uruchom aplikację za pomocą następującego polecenia `java`:

```

java --module-path mods ^
    -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor

```

Program po uruchomieniu przedstawia różne zadania dotyczące dodawania:

```

Ile wynosi 9 + 6? 15
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: t
Ile wynosi 2 + 6? 7
Odpowiedź nieprawidłowa. Spróbuj ponownie:
Ile wynosi 2 + 6? 8
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: n

```

27.9.8. Implementacja drugiego dostawcy usług

Klasa `MultiplicationProblemProvider` (rysunek 27.32) również implementuje interfejs `ProblemProvider` (wiersz 10.). Klasa ta jest umieszczona w pakiecie `com.deitel.multiplicationprovider` modułu `com.deitel.multiplicationprovider` (rysunek 27.33). Klasa `MultiplicationProblemProvider` jest w zasadzie prawie taka sama jak klasa `AdditionProblemProvider`, ale w wierszu 16. przekazuje jako operację tekst "*" i przesłania metodę `getResult`, wyliczając i zwracając iloczyn obu operandów.

```

1 // Rysunek 27.32. MultiplicationProblemProvider.java
2 // Implementacja klasy MultiplicationProblemProvider interfejsu
3 // ProblemProvider dla aplikacji MathTutor
4 package com.deitel.multiplicationprovider;
5
6 import java.util.Random;
7 import com.deitel.mathtutor.spi.Problem;
8 import com.deitel.mathtutor.spi.ProblemProvider;
9
10 public class MultiplicationProblemProvider implements ProblemProvider {
11     private static Random random = new Random();
12
13     // Zwraca nowe zadanie z mnożenia
14     @Override
15     public Problem getProblem() {
16         return new Problem(random.nextInt(10), random.nextInt(10), "*") {
17             // Przesłania getResult, aby obsłużyć operację mnożenia
18             @Override
19             public int getResult() {
20                 return getLeftOperand() * getRightOperand();
21             }
22         };
23     }
24 }

```

Rysunek 27.32. Implementacja klasy `MultiplicationProblemProvider` interfejsu `ProblemProvider` dla aplikacji `MathTutor`

Rysunek 27.33 przedstawia deklarację modułu `com.deitel.multiplicationprovider`. Ten moduł również wymaga modułu `com.deitel.mathtutor`. Wiersze 6. i 7. wskazują, że moduł zapewnia implementację interfejsu `ProblemProvider` za pomocą klasy `MultiplicationProblemProvider`.

```

1 // Rysunek 27.33. module-info.java
2 // Deklaracja modułu com.deitel.multiplicationprovider
3 module com.deitel.multiplicationprovider {
4     requires com.deitel.mathtutor;
5
6     provides com.deitel.mathtutor.spi.ProblemProvider with
7         com.deitel.multiplicationprovider.MultiplicationProblemProvider;
8 }

```

Rysunek 27.33. Deklaracja modułu `com.deitel.multiplicationprovider`

27.9.9. Uruchomienie aplikacji z dwoma dostawcami usług

Teraz uruchommy aplikację w taki sposób, aby w ścieżce modułów znalazły się klasy `AdditionProblemProvider` i `MultiplicationProblemProvider`. Najpierw jednak skompilujmy moduł `com.deitel.multiplicationprovider`:

```
javac --module-path mods ^
  -d mods/com.deitel.multiplicationprovider ^
  src/com.deitel.multiplicationprovider/module-info.java ^
  src/com.deitel.multiplicationprovider/com/deitel/
    multiplicationprovider/MultiplicationProblemProvider.java
```

Następnie uruchamiamy aplikację poleceniem `java`:

```
java --module-path mods ^
  -m com.deitel.mathtutor/com.deitel.mathtutor.MathTutor
```

Oto przykładowy rezultat działania programu dla zadań z dodawania i mnożenia:

```
Ile wynosi 4 * 8? 20
Odpowiedź nieprawidłowa. Spróbuj ponownie:
Ile wynosi 4 * 8? 32
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: t
Ile wynosi 3 * 6? 18
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: t
Ile wynosi 3 + 7? 10
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: t
Ile wynosi 9 + 3? 12
Dobrze!
Następne zadanie? t, aby kontynuować, n, aby przerwać: n
```

27.10. Podsumowanie

W tym rozdziale omówiliśmy całkowicie nowy element Javy 9, którym jest system modułów platformy Java. Przedstawiliśmy podstawowe koncepcje, z którymi spotka się każdy programista tworzący duży system.

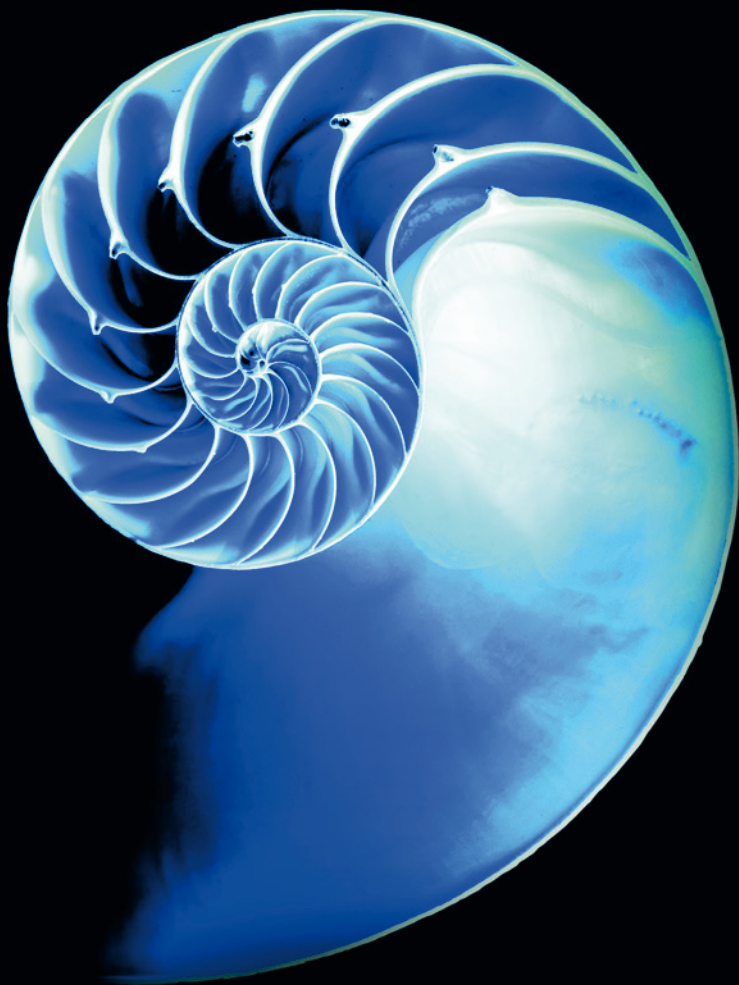
Dowiedziałeś się, że wszystkie moduły niejawnie zależą od `java.base`. Utworzyliśmy deklaracje modułów, które określają zależności modułu (dyrektywa `requires`), pakiety modułu udostępniane innym modułom (dyrektywa `exports`), oferowane usługi (dyrektywa `provides...with`), konsumowane usługi (dyrektywa `uses`), a także udostępnianie mechanizmu refleksji innym modułom (modyfikator `open` oraz dyrektywy `opens` i `opens...to`).

Aby pomóc w wizualizacji zależności między modułami, pokazaliśmy kilka grafów zależności modułów wykonanych w IDE NetBeans w wersji obsługującej JDK 9. Przedstawiliśmy kroki, jakie realizuje mechanizm rozwiązywania modułów, aby upewnić się, że są spełnione wszystkie zależności.

Użyliśmy nowego narzędzia `jlink` z JDK 9, aby wykonać mniejsze, własne systemy wykonawcze Javy, a następnie użyliśmy ich do uruchamiania aplikacji. Omówiliśmy silny mechanizm enkapsulacji stosowany przez moduły, a także kroki niezbędne do jawnego umożliwienia działania mechanizmu refleksji — modyfikator `open` oraz dyrektywy `opens` i `opens...to` z deklaracji modułu.

Pokazaliśmy też sposoby migracji ogromnej ilości starego kodu, który nie był tworzony w sposób modułowy, do modułowej wersji Javy 9. Wyjaśniliśmy, jak moduł nienazwany i moduły automatyczne upraszczają proces migracji. Użyliśmy narzędzia `jdeps`, aby poznać zależności kodu od modułów, a także sprawdzić, czy kod nie używa wewnętrznych API istniejących przed Javą 9 (w Javie 9 stosują one w większości silną enkapsulację).

Na końcu zajęliśmy się zastosowaniem usług i ich dostawców do budowania luźno powiązanych systemów, które korzystają z interfejsów dostawców usług, ich implementacji oraz klasy `ServiceLoader`. Przedstawiliśmy też zastosowanie w deklaracjach modułów dyrektywy `uses`, która pozwala wskazać, że moduł używa usługi, i dyrektywy `provides...with`, która umożliwia wskazanie, że moduł dostarcza jej implementację. W następnym rozdziale zajmiemy się pozostałymi tematami związanymi z Javą 9.



Cele

W tym rozdziale:

- Krótkie powtórzenie omówionych wcześniej nowości w Javie 9
- Nowy sposób numerowania wersji Javy
- Nowe metody wyrażeń regularnych: `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` i `results`
- Nowe metody `takeWhile` i `dropWhile` oraz przeciążenie `iterate` obiektów `Stream`
- Nowe elementy JavaFX i graficzne wprowadzone w Javie 9
- Wykorzystanie modułów w JShell
- Zmiany związane z bezpieczeństwem Javy 9 i inne powiązane tematy
- Elementy niedostępne w JDK 9 i Javie 9
- Pakiety, klasy i metody zaproponowane do usunięcia w następnym wydaniu Javy

28.1. Wprowadzenie	28.9.3. Obsługa protokołu DTLS (Datagram Transport Layer Security)
28.2. Przypomnienie — funkcjonalności Javy 9 omówione w poprzednich rozdziałach	28.9.4. Obsługa walidacji OCSP dla TLS
28.3. Nowa wersja formatu tekstowego	28.9.5. Rozszerzenie umożliwiające negocjacje protokołu warstwy aplikacyjnej w TLS
28.4. Wyrażenia regularne — nowe metody klasy <code>Matcher</code>	28.10. Inne tematy związane z Javą 9
28.4.1. Metody <code>appendReplacement</code> i <code>appendTail</code>	28.10.1. Usprawnienie łączenia tekstów
28.4.2. Metody <code>replaceFirst</code> i <code>replaceAll</code>	28.10.2. Obsługa usług i API logów na poziomie platformy
28.4.3. Metoda <code>results</code>	28.10.3. Aktualizacja API procesów
28.5. Nowe metody interfejsu <code>Stream</code>	28.10.4. Podpowiedzi dotyczące oczekiwania
28.5.1. Metody <code>takeWhile</code> i <code>dropWhile</code> strumienia	28.10.5. Obsługa paczek zasobów z kodowaniem UTF-8
28.5.2. Metoda <code>iterate</code> interfejsu <code>Stream</code>	28.10.6. Domyślne korzystanie z danych CLDR
28.5.3. Metoda <code>ofNullable</code> interfejsu <code>Stream</code>	28.10.7. Usunięcie ostrzeżeń o wycofaniu z instrukcji importu
28.6. Moduły w <code>JShell</code>	28.10.8. Wielowydaniowe pliki JAR
28.7. API skórek dostępne w <code>JavaFX 9</code>	28.10.9. Unicode 8
28.8. Inne usprawnienia związane z interfejsem graficznym i grafiką	28.10.10. Rozbudowa obsługi współbieżności
28.8.1. Obrazy o wielu rozdzielczościach	28.11. Elementy usunięte z <code>JDK</code> i Javy 9
28.8.2. Obsługa obrazów TIFF	28.12. Elementy zaproponowane do usunięcia w przyszłych wersjach Javy
28.8.3. Funkcjonalności pulpitu zależne od platformy	28.12.1. Ulepszone wycofywanie
28.9. Tematy związane z bezpieczeństwem Javy 9	28.12.2. Elementy, które prawdopodobnie zostaną usunięte w przyszłych wydaniach Javy
28.9.1. Filtrowanie nadchodzących danych serializowanych	28.12.3. Znajdowanie wycofywanych funkcjonalności
28.9.2. Domyślne tworzenie magazynów kluczy PKCS12	28.12.4. Aplety Javy
	28.13. Podsumowanie

28.1. Wprowadzenie

W momencie wydawania książki dokument „Java Specification Request (JSR) 379: Java SE 9” nie istniał jeszcze w wersji finalnej i był dostępny pod adresem:

<http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>

JSR 379 zawiera informacje na temat:

- funkcjonalności dodanych w Javie 9;
- funkcjonalności usuniętych w Javie 9;
- funkcjonalności zaproponowanych do usunięcia w następnych wersjach Javy.

Po zatwierdzeniu ostatecznej wersji JSR 379 będzie dostępny do pobrania pod adresem:

<https://www.jcp.org/en/jsr/detail?id=379>

JSR 379 to lektura obowiązkowa dla każdego programisty Javy 9. Dokument ten zapewnia zarówno ogólny opis, jak i szczegółowe omówienie wszystkich funkcjonalności Javy 9, a także łączy do kluczowych dokumentów JEP i JSR.

W każdej nowej wersji języka pojawiają się elementy, z których od razu skorzysta prawie każdy programista, elementy będące w obszarze zainteresowania niewielu programistów i pewne specyficzne elementy interesujące tylko nieliczne osoby. Niniejszy rozdział podzieliłiśmy zatem na kilka części:

- przypomnienie nowy funkcjonalności Javy 9 opisanych w poprzednich rozdziałach;
- przykłady użycia i opis funkcjonalności elementów, które mogą okazać się przydatne szerszemu gronu programistów;
- krótkie omówienie bardziej specjalistycznych funkcjonalności wraz z łączami do miejsc, gdzie można znaleźć więcej informacji na ich temat;
- lista funkcjonalności usuniętych z JDK 9 i Javy 9;
- lista funkcjonalności proponowanych do usunięcia w następnych wydaniach Javy.

Programiści powinni oczywiście unikać korzystania w nowym kodzie z funkcjonalności wymienionych w ostatnich dwóch grupach, a w starym kodzie powinni zastąpić te funkcjonalności w momencie migracji do Javy 9.

28.2. Przypomnienie — funkcjonalności Javy 9 omówione w poprzednich rozdziałach

Oto lista omówionych w książce funkcjonalności Javy 9 wraz z łączami zawierającymi informacje na ich temat:

- Podkreślenie () nie jest już poprawnym identyfikatorem (podrozdział 2.2). To jeden z kilku elementów opisywanych w dokumencie „JEP 213: Milling Project Coin” (<http://openjdk.java.net/jeps/213>).
- Wzmianka o rozbudowie możliwości SecureRandom (podrozdział 6.8) na podstawie JEP 273 (<http://openjdk.java.net/jeps/273>).
- Od Javy 9 kompilator zgłasza ostrzeżenie, jeśli próbujemy skorzystać ze statycznych składowych klasy poprzez instancję klasy (podrozdział 8.11).
- Wprowadzenie prywatnych interfejsów metod (podrozdział 10.11). To jeden z elementów opisywanych w dokumencie „JEP 213: Milling Project Coin”.
- Wzmianka o nowym API przejścia przez stos (podrozdział 11.7) z JEP 259 (<http://openjdk.java.net/jeps/259>).
- Wzmianka o tym, że efektywne finalne zmienne typu AutoCloseable mogą być teraz stosowane w instrukcjach try z zasobami (podrozdział 11.12). To jeden z elementów opisywanych w dokumencie „JEP 213: Milling Project Coin”.
- Omówienie nowych funkcjonalności JavaFX 9, a także innych usprawnień związanych z grafiką i GUI (podrozdział 13.8).
- Przedstawienie bardziej zwartej reprezentacji obiektów String w Javie 9 (podrozdział 14.3) na podstawie JEP 254 (<http://openjdk.java.net/jeps/254>).
- Przedstawienie nowych metod fabrycznych związanych z tworzeniem kolekcji tylko do odczytu (podrozdział 16.14) na podstawie JEP 269 (<http://openjdk.java.net/jeps/269>).

- W rozdziale 25. przedstawiliśmy szczegółowo na podstawie wielu przykładów nowe narzędzie JDK o nazwie `jshell`.
- W rozdziale 30. przedstawiliśmy szczegółowo i z przykładami nowy system modułów platformy Java 9.

28.3. Nowa wersja formatu tekstowego

Przed Javą 9 wersje JDK były numerowane według schematu `1.X.0_number` ↪ *Aktualizacji*, gdzie `X` to numer głównego wydania Javy. Na przykład:

- aktualne wydanie JDK Javy 8 na numer `jdk1.8.0_131`;
- finalne wydanie JDK Javy 7 ma numer `jdk1.7.0_80`.

Ten system numeracji uległ zmianie. JDK 9 będzie znane jako `jdk-9`. W przyszłości będą tworzone podwersje z dodanymi nowymi funkcjonalnościami, a wydania bezpieczeństwa będą jedynie poprawiały znalezione luki w zabezpieczeniach. Każda z takich aktualizacji będzie odpowiednio oznaczona w wersji JDK. W wersji 9.1.3 poszczególne elementy oznaczać będą:

- 9 — numer głównej wersji Javy,
- 1 — numer podwersji po aktualizacji,
- 3 — numer aktualizacji bezpieczeństwa.

Jeśli pojawi się wersja 9.2.5, oznaczać to będzie wydanie Javy 9, które jest drugim wydaniem po aktualizacjach z łącznie pięcioma wydaniem bezpieczeństwa (dotyczą one zarówno wersji głównych, jak i podwersji). Dodatkowe informacje na ten temat znajdziesz w dokumencie JEP 223 dostępnym pod adresem:

<http://openjdk.java.net/jeps/223>

28.4. Wyrażenia regularne — nowe metody klasy `Matcher`

Java SE 9 dodaje kilka nowych metod i przeciążeń: `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` i `results` (rysunek 28.1).

```

1 // Rysunek 28.1. MatcherMethods.java
2 // Nowe metody klasy Matcher w Javie 9
3 import java.util.regex.Matcher;
4 import java.util.regex.Pattern;
5
6 public class MatcherMethods {
7     public static void main(String[] args) {
8         String sentence = "to pan a to plan z palmami panamskimi";
9
10        System.out.printf("zdanie: %s\n", sentence);
11
12        // Wykorzystywany przez appendReplacement i appendTail z Matcher
13        Pattern pattern = Pattern.compile("an"); // Wyrażenie regularne do
                                                ↪ dopasowania
14
15        // Dopasowanie wyrażenia regularnego do tekstu
16        // i zastąpienie każdego dopasowania dużymi literami

```

Rysunek 28.1. Nowe metody klasy `Matcher` w Javie 9

```

17     Matcher matcher = pattern.matcher(sentence);
18
19     // Służy do przebudowania tekstu
20     StringBuilder builder = new StringBuilder();
21
22     // Dołącza tekst do builder i konwertuje każde dopasowanie na duże litery
23     while (matcher.find()) {
24         matcher.appendReplacement(
25             builder, matcher.group().toUpperCase());
26     }
27
28     // Dodanie pozostałej części pierwotnego tekstu do builder
29     matcher.appendTail(builder);
30     System.out.printf(
31         "%nPo appendReplacement i appendTail: %s%n", builder);
32
33     // Użycie metody replaceFirst z Matcher
34     matcher.reset(); // Przywrócenie matcher do stanu początkowego
35     System.out.printf("%nPrzed replaceFirst: %s%n", sentence);
36     String result = matcher.replaceFirst(m -> m.group().toUpperCase());
37     System.out.printf("Po replaceFirst: %s%n", result);
38
39     // Użycie metody replaceAll z Matcher
40     matcher.reset(); // Przywrócenie matcher do stanu początkowego
41     System.out.printf("%nPrzed replaceAll: %s%n", sentence);
42     result = matcher.replaceAll(m -> m.group().toUpperCase());
43     System.out.printf("Po replaceAll: %s%n", result);
44
45     // Użycie metody results do pobrania Stream<MatchResult>
46     System.out.printf("%nUżycie metody results z Matcher:%n");
47     pattern = Pattern.compile("\\w+"); // Wyrażenie regularne do dopasowania
48     matcher = pattern.matcher(sentence);
49     System.out.printf("Liczba słów: %d%n",
50         matcher.results().count());
51
52     matcher.reset(); // Przywrócenie matcher do stanu początkowego
53     System.out.printf("Średnia liczba znaków w słowie: %f%n",
54         matcher.results()
55             .mapToInt(m -> m.group().length())
56             .average().orElse(0));
57 }
58 }

```

zdanie: to pan a to plan z palmami panamskimi

Po appendReplacement i appendTail: to pAN a to pLAN z palmami pANamskimi

Przed replaceFirst: to pan a to plan z palmami panamskimi

Po replaceFirst: to pAN a to plan z palmami panamskimi

Przed replaceAll: to pan a to plan z palmami panamskimi

Po replaceAll: to pAN a to pLAN z palmami pANamskimi

Użycie metody results z Matcher:

Liczba słów: 8

Średnia liczba znaków w słowie: 3,750000

Rysunek 28.1. Nowe metody klasy Matcher w Javie 9 — ciąg dalszy

28.4.1. Metody `appendReplacement` i `appendTail`

Nowe metody przeciążone `appendReplacement` (wiersze 24. i 25.) i `appendTail` (wiersz 29.) klasy `Matcher` są używane w pętli z metodą `find` (wiersz 23.) i klasą `StringBuilder` w celu budowania nowego tekstu, w którym każde dopasowanie jest zastępowane wskazanym tekstem. Na końcu procesu obiekt `StringBuilder` zawiera pierwotny tekst zaktualizowany o zastąpienia. Wiersze od 13. do 26. działają w następujący sposób.

- Wiersz 13. tworzy obiekt `Pattern`, który będzie stanowił wyszukiwaną treść (tekst `an`).
- Wiersz 17. tworzy obiekt `Matcher` dla zawartości tekstu w zmiennej `sentence` (zadeklarowanej w wierszu 8.).
- Wiersz 20. tworzy obiekt `StringBuilder`, do którego trafią wyniki.
- Wiersz 23. używa metody `find` z `Matcher`, aby znaleźć wystąpienie `an` w pierwotnym tekście.
- Jeśli dopasowanie zostanie znalezione, metoda `find` zwraca `true`, a wiersz 24. wywołuje metodę `appendReplacement` z `Matcher`, aby zastąpić tekst `an` tekstem `AN`. Drugi argument metody wywołuje metodę `group` z `Matcher`, aby pobrać tekst pasujący do wyrażenia regularnego (w tym przypadku `an`). W uzyskanym tekście zamieniamy małe litery na duże. Metoda `appendReplacement` dodaje do `StringBuilder` przekazanego jako pierwszy argument wszystkie znaki aż do dopasowania, a następnie zastępuje dopasowane znaki drugim argumentem. Pętla kontynuuje działanie, próbując znaleźć następne dopasowanie w pierwotnym tekście znajdujące się po pierwszym znaku za poprzednim dopasowaniem.
- Gdy metoda `find` zwróci wartość `false`, pętla kończy się, a wiersz 29. używa metody `appendTail` z `Matcher`, aby dołączyć do `StringBuilder` pozostałe znaki pierwotnego tekstu.

Po zakończeniu całego procesu pierwotny tekst `to pan a to plan z palmami` panamskimi zamienia się w `to pAN a to pLAN z palmami pANamskimi`.

28.4.2. Metody `replaceFirst` i `replaceAll`

Przeciążone metody `replaceFirst` (wiersz 36.) i `replaceAll` (wiersz 42.) zastępują odpowiednio pierwsze lub wszystkie dopasowania w tekście, przyjmując obiekt `Function`, który otrzymuje `MatchResult` i zwraca zastąpienie. Wiersze 36. i 42. implementują interfejs `Function` za pomocą `lambd`, które używają grup do wydobycia dopasowania, a następnie zastąpienia go wersją pisaną dużymi literami. Wiersze 34. i 40. wywołują metodę `reset` z `Matcher`, aby następne wywołania metod `replaceFirst` i `replaceAll` zaczynały poszukiwania od początku tekstu znajdującego się w zmiennej `sentence`.

28.4.3. Metoda `results`

Nowa metoda `results` klasy `Matcher` (wiersze 50. i 54.) zwraca strumień danych `MatchResult`. W wierszach od 47. do 50. używamy wyrażenia regularnego `\w+`, aby dopasować się do znaków słowa, a następnie wykorzystać metodę `count` do zliczenia słów znajdujących się w zdaniu. Po powrocie do początku tekstu

(wiersz 52.) wiersze od 54. do 56. używają strumienia do odwzorowania każdego słowa na liczbę znaków (dzięki `mapToInt`), a następnie wyliczają średnią długość słowa w tekście za pomocą metody `average` z `IntStream`.

28.5. Nowe metody interfejsu Stream

Java 9 dodaje kilka nowych metod do strumieni: **`takeWhile`**, **`dropWhile`**, **`iterate`** i **`ofNullable`** (rysunek 37.2). Wszystkie poza `ofNullable` są dostępne również dla strumieni liczbowych takich jak `IntStream`.

```

1 // Rysunek 28.2. StreamMethods.java
2 // Nowe metody takeWhile, dropWhile, iterate i ofNullable
3 // dla strumieni dostępne w Javie 9
4 import java.util.stream.Collectors;
5 import java.util.stream.IntStream;
6 import java.util.stream.Stream;
7
8 public class StreamMethods {
9     public static void main(String[] args) {
10         int[] values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12         System.out.printf("Tablica values zawiera: %s%n",
13             IntStream.of(values)
14                 .mapToObj(String::valueOf)
15                 .collect(Collectors.joining(" ")));
16
17         // Pobierz największy przedrostek strumienia z elementami mniejszymi niż 6
18         System.out.println("Przykład użycia takeWhile i dropWhile:");
19         System.out.printf("Elementy mniejsze niż 6: %s%n",
20             IntStream.of(values)
21                 .takeWhile(e -> e < 6)
22                 .mapToObj(String::valueOf)
23                 .collect(Collectors.joining(" ")));
24
25         // Odrzuć największy przedrostek strumienia z elementami mniejszymi niż 6
26         System.out.printf("Elementy większe lub równe 6: %s%n",
27             IntStream.of(values)
28                 .dropWhile(e -> e < 6)
29                 .mapToObj(String::valueOf)
30                 .collect(Collectors.joining(" ")));
31
32         // Użyj iterate do wygenerowania strumienia potęg 3 mniejszych niż 10000
33         System.out.printf("%nPrzykład użycia iterate:%n");
34         System.out.printf("Potęgi 3 mniejsze niż 10 000: %s%n",
35             IntStream.iterate(3, n -> n < 10_000, n -> n * 3)
36                 .mapToObj(String::valueOf)
37                 .collect(Collectors.joining(" ")));
38
39         // Przykład użycia ofNullable
40         System.out.printf("%nPrzykład użycia ofNullable:%n");
41         System.out.printf("Liczba elementów strumienia: %d%n",
42             Stream.ofNullable(null).count());
43         System.out.printf("Liczba elementów strumienia: %d%n",
44             Stream.ofNullable("szary").count());

```

Rysunek 28.2. Nowe metody `takeWhile`, `dropWhile`, `iterate` i `ofNullable` dla strumieni dostępne w Javie 9

```
45     }
46 }
```

Tablica values zawiera: 1 2 3 4 5 6 7 8 9 10

Przykład użycia takeWhile i dropWhile:

Elementy mniejsze niż 6: 1 2 3 4 5

Elementy większe lub równe 6: 6 7 8 9 10

Przykład użycia iterate:

Potęgi 3 mniejsze niż 10 000: 3 9 27 81 243 729 2187 6561

Przykład użycia ofNullable:

Liczba elementów strumienia: 0

Liczba elementów strumienia: 1

Rysunek 28.2. Nowe metody takeWhile, dropWhile, iterate i ofNullable dla strumieni dostępne w Javie 9 — *ciąg dalszy*

28.5.1. Metody takeWhile i dropWhile strumienia

Wiersze od 19. do 30. ilustrują metody takeWhile i dropWhile, które na podstawie obiektu Predicate odpowiednio dołączają lub pomijają elementy strumienia. Metody te powinny być stosowane dla strumieni uporządkowanych. W odróżnieniu od metody filter, która przetwarza wszystkie elementy strumienia, każda z nowych metod przetwarza elementy tylko do momentu, w którym predykat stanie się nieprawdziwy (zwróci false).

Potok strumienia z wierszy od 19. do 23. pobiera wartości z początku strumienia dopóty, dopóki wartości int są mniejsze od 6. Predykat zwraca wartość true tylko dla pierwszych pięciu elementów; pozostałe elementy strumienia zostaną zignorowane. Dla pięciu elementów, które pozostały w strumieniu, stosujemy odwzorowanie na tekst, a każdą z wartości łączymy w jedną całość, oddzielając je znakiem spacji.

Potok strumienia z wierszy od 26. do 30. odrzuca wartości int z początku strumienia dopóty, dopóki wartości int są mniejsze od 6. Wynikowy strumień zawiera wartości większe lub równe 6. Dla elementów, które pozostały w strumieniu, odwzorowujemy każdy z nich na tekst, a następnie łączymy w całość ze znakami spacji jako separatorami.



28.1. Wskazówka zapobiegająca błędom

Wywołuj metody takeWhile i dropWhile tylko dla strumieni uporządkowanych. Jeśli metody te zostaną wywołane dla strumieni nieuporządkowanych, mogą zwrócić tylko część poprawnych wyników lub nawet nie zwrócić żadnych wyników, choć takowe istnieją.



28.1. Wskazówka poprawiająca wydajność

Zgodnie z dokumentacją interfejsu Stream mogą pojawić się problemy wydajnościowe w przypadku stosowania metod takeWhile i dropWhile dla uporządkowanych strumieni zrównoleglonych. Więcej informacji na ten temat znajdziesz na stronie <http://download.java.net/java/jdk9/docs/api/java/util/stream/Stream.html>.

28.5.2. Metoda `iterate` interfejsu Stream

W podrozdziale 4.8 przedstawiliśmy pętlę `while`, która obliczała potęgę 3 mniejsze niż 100. Wiersze od 34. do 37. pokazują, jak użyć nowego przeciążenia metody `iterate`, aby wygenerować strumień wartości `int` zawierający potęgę 3 mniejsze niż 10 000. Nowe przeciążenie przyjmuje jako argumenty następujące dane:

- wartość ziarna, które staje się pierwszym elementem strumienia;
- obiekt `Predicate` wskazujący moment zakończenia generowania elementów;
- obiekt `UnaryOperator` wywoływany początkowo dla wartości ziarna, a następnie dla każdej poprzedniej wartości aż do momentu, w którym obiekt `Predicate` zwróci `false`.

W tym przypadku wartość ziarna wynosi 3, obiekt `Predicate` wskazuje zamiar kontynuacji generowania, jeśli ostatni element jest mniejszy od 10 000, a obiekt `UnaryOperator` mnoży wartość poprzedniego elementu przez 3. Na końcu zamieniamy liczby na tekst, a następnie łączymy je w jeden tekst, rozdzielając wartości spacjami.

28.5.3. Metoda `ofNullable` interfejsu Stream

Nowa metoda statyczna `ofNullable` interfejsu Stream otrzymuje referencję do obiektu i jeśli referencja ta nie jest równa `null`, zwraca jednoelementowy strumień zawierający obiekt; w przeciwnym razie zwraca pusty strumień. Wiersze 42. i 44. ilustrują dosyć proste przykłady, w których uzyskujemy odpowiednio pusty strumień i strumień jednoelementowy.

Metoda `ofNullable` służy najczęściej do upewnienia się, że referencja jest różna od `null`, przed wykonaniem operacji na potoku strumienia. Przyjrzyjmy się bazie danych pracowników firmy. Program może odpytać bazę danych, aby znaleźć wszystkich pracowników (obiekty `Employee`) w danym dziale, a następnie zapamiętać ich jako kolekcję w obiekcie `Department` przechowywanym w zmiennej `department`. Gdyby zapytanie wykonać dla nieistniejącego działu, referencją w zmiennej byłoby `null`. Zamiast więc najpierw sprawdzać, czy `department` jest równe `null`, a potem wykonywać zadanie, czyli napisać kod:

```
if (department != null) {
    // Wykonaj zadanie
}
```

możemy użyć kodu takiego jak ten:

```
Stream.ofNullable(department)
    .flatMap(Department::streamEmployees)
    ... // Wykonaj zadania dla każdego obiektu Employee
```

Zakładamy tutaj, że klasa `Department` zawiera publiczną metodę `streamEmployees` zwracającą strumień obiektów `Employee`. Jeśli wartość zmiennej `department` jest różna od `null`, potok wykona zadanie `flatMap`, zamieniając obiekt `Department` w strumień obiektów `Employee` w celu dalszego przetworzenia. Jeśli wartość zmiennej `department` jest równa `null`, metoda `ofNullable` zwróci pusty strumień, więc potok po prostu nie będzie miał nic do roboty.

28.6. Moduły w JShell

W podrozdziale 25.10 pokazaliśmy, jak dodać własne klasy do ścieżek klas JShell, aby móc wejść z nimi w interakcję. Tutaj pokażemy, jak w podobny sposób wejść w interakcję z modulem `com.deitel.timelibrary` z podrozdziału 30.4. Na potrzeby tego podrozdziału otwórz wiersz polecenia, przejdź do folderu *TimeApp* z przykładami dla rozdziału 30. i uruchom narzędzie `jshell`.

Dodanie modułu do sesji JShell

Polecenie `/env` pozwala określić ścieżkę modułów, a także wskazać konkretne moduły wczytywane z tych ścieżek. Aby dodać moduł `com.deitel.timelibrary`, wykonaj następujące polecenie:

```
jshell> /env -module-path jars -add-modules com.deitel.timelibrary
| Setting new options and restoring state.

jshell>
```

Opcja `-module-path` wskazuje, gdzie znajdują się moduły, które chcemy wczytać (w tym przypadku jest to folder *jars* znajdujący się w folderze, w którym uruchomiliśmy JShell). Opcja `-add-modules` wskazuje konkretne moduły do wczytania (w tym przypadku moduł `com.deitel.timelibrary`).

Import klasy z pakietu eksportowanego przez moduł

Po wczytaniu modułu możemy importować typy z dowolnych pakietów eksportowanych przez moduł. Poniższe polecenie importuje klasę `Time1`:

```
jshell> import com.deitel.timelibrary.Time1

jshell>
```

Użycie zaimportowanej klasy

W tym momencie możemy zacząć korzystać z klasy `Time1` w taki sam sposób, jak z innych klas w rozdziale 25. Utwórz obiekt `Time1`:

```
jshell> Time1 time = new Time1()
time ==> 12:00:00 AM

jshell>
```

Następnie sprawdź składowe obiektu z wykorzystaniem mechanizmu automatycznego uzupełniania, wpisując `time.` i naciskając klawisz *Tab*:

```
jshell> time.
equals(          getClass()          hashCode()
notify()        notifyAll()
setTime(        toString()           toUniversalString()
wait(

jshell> time.
```

Aby wyświetlić tylko składowe zaczynające się od `to`, wpisz `to` i naciśnij klawisz *Tab*:

```
jshell> time.to
toString()      toUniversalString()

jshell> time.to
```

Następnie wpisz `U` i naciśnij klawisz *Tab*, aby automatycznie uzupełnić polecenie jako wywołanie metody `toUniversalString()`. Naciśnij klawisz *Enter*, aby wywołać metodę i przypisać czas w formacie 24-godzinnym do niejawnie zadeklarowanej zmiennej:

```
jshell> time.toUniversalString()
$3 ==> "00:00:00"

jshell>
```

28.7. API skórek dostępne w JavaFX 9

W rozdziale 22. przedstawiliśmy, w jaki sposób zmieniać styl obiektów JavaFX za pomocą technologii CSS (ang. *Cascading Style Sheet*), która powstała do definiowania stylu stron WWW. CSS umożliwia określenie **prezentacji** (czyli czcionek, odstępów, rozmiarów, kolorów i położenia) w sposób niezależny od **struktury** i **zawartości** interfejsu graficznego (kontenerów układu, kształtów, tekstu itp.). Jeśli za część prezentacyjną odpowiada tylko i wyłącznie arkusz stylów (określa zasady wyglądu GUI), można zmienić taki arkusz na inny, aby całkowicie odmienić wygląd strony (mówimy wtedy o nałożeniu nowej **skórki** lub **kompozycji**).

Każda kontrolka JavaFX korzysta z klasy skórki, która definiuje jej domyślny wygląd i sposób interakcji. W JavaFX 8 klasy skórek były zdefiniowane jako wewnętrzne API, ale wielu programistów rozszerzało te klasy, aby tworzyć własne skórki.



28.1. Wskazówka poprawiająca przenośność kodu

Ze względu na silną enkapsulację wewnętrzne API skórek z JavaFX 8 nie są już dostępne w Javie 9. Jeśli więc tworzyłeś własne skórki na podstawie API sprzed Javy 9, taki kod nie będzie działał w Javie 9, a także nie uda się go skompilować w JDK 9.

Jako część prac dotyczących modularyzacji Javy 9 udostępniono w JavaFX 9 publiczne API klas skórek jako pakiet `javafx.scene.control.skin` opisany w dokumencie JEP 253:

<http://openjdk.java.net/jeps/253>

Nowe klasy skórek są bezpośrednimi lub pośrednimi podklasami klasy `SkinBase` (pakiet `javafx.scene.control`). Można rozszerzać odpowiednie klasy skórek, aby dostosowywać do swoich potrzeb wygląd kontrolki. Następnie wystarczy we właściwości CSS JavaFX o nazwie **-fx-skin** podać pełną nazwę klasy skórki.

Najczęściej CSS to najprostszy sposób zmiany wyglądu kontrolki JavaFX. Aby jednak uzyskać bardziej precyzyjną kontrolę nad każdym aspektem kontrolki, włączając w to sterowanie rozmiarem, położeniem i interakcjami z klawiaturą i myszą, rozszerz klasę `SkinBase` lub jedną z jej podklas znajdujących się w pakiecie `javafx.scene.control.skin`.

28.8. Inne usprawnienia związane z interfejsem graficznym i grafiką

Poza zmianami opisanymi w podrozdziałach 13.8 i 28.7 JSR 379 zawiera dodatkowe usprawnienia dotyczące obsługi obrazów, a także lepszą integrację z pulpitem.

28.8.1. Obrazy o wielu rozdzielczościach

Aplikacje często wyświetlają różne wersje obrazu na podstawie rozdzielczości i wielkości ekranu. Java 9 dodaje wsparcie dla obrazów o wielu rozdzielczościach, w których jeden obraz jest tak naprawdę grupą obrazów, a klasa `Graphics` (pakiet `java.awt`) wybiera odpowiednią dla danego urządzenia wersję obrazu. Więcej informacji na ten temat znajdziesz na stronie:

<http://openjdk.java.net/jeps/251>

28.8.2. Obsługa obrazów TIFF

Framework Image I/O odpowiada za API umożliwiające odczyt i zapis obrazów. Framework obsługuje dodatkowe moduły pozwalające odczytywać obrazy w różnych formatach, ale wszystkie implementacje Javy muszą obsługiwać formaty PNG i JPEG. Od Javy 9 wszystkie implementacje muszą również obsługiwać format TIFF (nazywany też formatem TIF) — w macOS TIFF jest jednym ze standardowych formatów, a wiele innych platform również obsługuje ten format. Więcej informacji na temat frameworku Image I/O znajdziesz pod adresem:

<https://docs.oracle.com/javase/8/docs/technotes/guides/imageio/>

Dodatkowe informacje na temat obsługi TIFF znajdziesz pod adresem:

<http://openjdk.java.net/jeps/262>

28.8.3. Funkcjonalności pulpitu zależne od platformy

W Javie 9 różne wewnętrzne API wykorzystywane do różnych integracji z pulpitem systemu operacyjnego — na przykład do współpracy z obszarem dokowania w macOS — nie są dostępne ze względu na silną enkapsulację modułów. JEP 272 dodaje nowe publiczne API dotyczące tych funkcjonalności dla systemu macOS, a także zapewnia podobne funkcjonalności dla innych systemów operacyjnych (np. Windows i Linux). Innymi obsługiwanymi funkcjonalnościami są:

- nasłuchiwanie zdarzeń logowania, wylogowania, blokady i odblokowania ekranu, co pozwoli aplikacjom odpowiednio reagować na takie sytuacje;
- przykuwanie uwagi użytkownika poprzez miganie lub inne animacje w obszarze dokowania lub na pasku zadań;
- wyświetlanie pasków postępu w obszarze dokowania lub na pasku zadań.

Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/272>

28.9. Tematy związane z bezpieczeństwem Javy 9

Warto, aby programiści zdawali sobie sprawę z udoskonaleń związanych z bezpieczeństwem wprowadzonych w Javie 9. W tym podrozdziale jedynie pokrótce wspomnimy o kilku z nich, a także wskażemy miejsca, gdzie można dowiedzieć się więcej na ich temat.

28.9.1. Filtrowanie nadchodzących danych serializowanych

Mechanizm **serializacji obiektów** Javy umożliwia tworzenie obiektów serializowanych — ciągu bajtów, które zawierają dane obiektu oraz informacje o typie obiektu i typach danych. Po udanej serializacji obiekt można przywrócić do stanu pierwotnego, odczytując go w programie i **deserializując** — informacje o typie i wszystkie inne dane pozwalają odtworzyć oryginalną postać obiektu w pamięci.

Proces deserializacji niesie jednak ze sobą ryzyko problemów z bezpieczeństwem. Jeśli odczytywane bajty pochodzą z połączenia sieciowego, atakujący mógł przechwycić połączenie i wstrzyknąć błędne dane. Jeżeli nie dokonamy walidacji danych po deserializacji, utworzony obiekt może znaleźć się w stanie uniemożliwiającym poprawne działanie. Mechanizm deserializacji pozwala na odtworzenie obiektu tylko wtedy, gdy w trakcie działania programu istnieje dostęp do definicji oryginalnego typu, ale odtworzenie dotyczy może dowolnego typu. Jeśli serializowany obiekt zawiera tablicę, atakujący mógłby wstrzyknąć dowolnie dużą liczbę elementów i tym samym doprowadzić do zajęcia całej pamięci operacyjnej.

Dokument JEP 290 dostępny pod adresem:

<http://openjdk.java.net/jeps/290>

zawiera usprawnienia w mechanizmach bezpieczeństwa serializacji pozwalające programom dodać filtry ograniczające typy, które mogą być poddawane deserializacji, sprawdzające długości tablic itp.

28.9.2. Domyślne tworzenie magazynów kluczy PKCS12

Magazyn kluczy przechowuje certyfikaty bezpieczeństwa używane przy szyfrowaniu. Java wykorzystuje własne mechanizmy magazynów kluczy od wersji 1.2 (1998). Domyślnie Java 9 używać będzie popularnego i rozszerzalnego magazynu kluczy PKCS12, który jest bezpieczniejszy i pozwala na współpracę Javy z innymi systemami obsługującymi ten sam standard. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/229>

28.9.3. Obsługa protokołu DTLS (Datagram Transport Layer Security)

Datagram to bezpołączeniowy mechanizm komunikacji poprzez sieć. Java 9 dodaje obsługę protokołu DTLS (ang. *Datagram Transport Layer Security*), który umożliwia bezpieczną komunikację za pomocą datagramów. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/219>

28.9.4. Obsługa walidacji OCSP dla TLS

Certyfikatów bezpieczeństwa X.509 używa się w kryptografii klucza publicznego. JEP 249 do usprawnienie wydajnościowe i związane z bezpieczeństwem dotyczące sposobu sprawdzania dalszej ważności certyfikatów X.509. Szczegółów szukaj pod adresem:

<http://openjdk.java.net/jeps/249>

28.9.5. Rozszerzenie umożliwiające negocjację protokołu warstwy aplikacyjnej w TLS

To rozszerzenie bezpieczeństwa dotyczące pakietu `javax.net.ssl`, które pozwala aplikacji wybrać listę protokołów komunikacyjnych w przypadku nawiązywania bezpiecznych połączeń z drugą stroną. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/244>

28.10. Inne tematy związane z Javą 9

W tym podrozdziale pokrótce wspomnimy o kilku innych funkcjonalnościach zawartych w JSR 379. Ponieważ w trakcie przygotowywania tego materiału przed oficjalnym wydaniem Javy 9 mieliśmy dostęp do ograniczonej dokumentacji, skupiliśmy się głównie na informacjach zawartych w dokumentach JSR i JEP. Nie będziemy komentować niektórych nowych elementów Javy 9. Są to między innymi:

- „JEP 193: Variable Handles (<http://openjdk.java.net/jeps/193>),
- „JEP 268: XML Catalogs (<http://openjdk.java.net/jeps/268>),
- „JEP 274: Enhanced Method Handles (<http://openjdk.java.net/jeps/274>).

28.10.1. Usprawnienie łączenia tekstów

Dokument „JEP 280: Indify String Concatenation” to niewidoczne dla programistów rozszerzenie `javac`, które ma w przyszłości usprawnić wydajność łączenia tekstów. Celem jest wykonanie i poprawa wydajności łączenia tekstów w przyszłych wersjach Javy **bez** modyfikacji kodu bajtowego tworzonego przez `javac`. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/280>

28.10.2. Obsługa usług i API logów na poziomie platformy

Programiści bardzo często stosują frameworki logów, aby zbierać informacje o działaniu systemu pomocne w trakcie znajdowania błędów, przygotowywania statystyk, wykrywania włamań do systemu itp. Dokument „JEP 264: Platform Logging API and Service” dodaje API logów w celu użycia przez klasy platformy znajdujące się w module `java.base`. Programiści mogą zaimplementować dostawcę usług, który przekieruje informacje do używanego przez aplikację frameworku obsługującego logi. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/264>

28.10.3. Aktualizacja API procesów

Java 9 zawiera usprawnienia w API umożliwiającym programom Javy interakcję ze specyficznymi procesami systemu operacyjnego bez korzystania z kodu natywnego pisanego w C lub C++. Niektóre z usprawnień dotyczą dostępu do identyfikatorów procesów, ich argumentów, czasu uruchomienia, łącznego czasu procesora i nazwy, a także przerywania i monitorowania procesów z poziomu aplikacji Javy. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/102>

28.10.4. Podpowiedzi dotyczące oczekiwania

W podrozdziale 23.7 wprowadziliśmy technikę wielowątkową, w której wątek czekający na uzyskanie blokady obiektu korzysta z pętli do określenia, czy blokada jest dostępna, a jeśli nie jest — rozpoczyna oczekiwanie. Za każdym razem, gdy wątek zostanie powiadomiony o konieczności sprawdzenia stanu, pętla wykona iteracje aż do momentu uzyskania blokady. Jest to tak zwana pętla oczekiwania. Java 9 dodaje nowe API, które pozwala poinformować JVM, że jest to tego rodzaju pętla. Na niektórych platformach sprzętowych JVM może wykorzystać tę informację do poprawy wydajności i zmniejszenia zapotrzebowania na energię (co ma ogromne znaczenie w urządzeniach przenośnych). Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/285>

28.10.5. Obsługa paczek zasobów z kodowaniem UTF-8

Klasa `ResourceBundle` (pakiet `java.util`) umożliwia programom wczytywanie informacji specyficznych dla konkretnej instalacji, np. tekstów w różnych językach. Mechanizm ten wykorzystuje się bardzo często do lokalizowania aplikacji na potrzeby różnych rejonów świata. Java 9 uaktualnia klasę `ResourceBundle` o obsługę zasobów zakodowanych w formacie UTF-8 (<https://pl.wikipedia.org/wiki/UTF-8>). Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/226>

28.10.6. Domyślne korzystanie z danych CLDR

CLDR (ang. *Common Locale Data Repository*) z Unicode (<http://cldr.unicode.org/>) to rozbudowane repozytorium, z którego mogą korzystać programiści w trakcie umiędzynarodawiania swoich aplikacji. Dane w tym repozytorium to między innymi informacje o:

- formatowaniu dat, czasu, liczb i walut;
- tłumaczeniu nazw języków, krajów, regionów, miesięcy, dni itp.;
- ogólnych danych związanych z językami, np. reguły stosowania wielkich liter, odmian, zasad sortowania itp.;
- krajach i regionach.

Obsługa CLDR została wprowadzona w Javie 8, ale dopiero w Javie 9 stała się domyślna. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/252>

28.10.7. Usunięcie ostrzeżeń o wycofaniu z instrukcji importu

W wielu firmach zasady pisania kodu wymagają kompilacji kodu bez żadnych ostrzeżeń. W JDK 8, gdy zaimportowaliśmy wycofywany typ lub statycznie zaimportowaliśmy wycofaną składową typu, kompilator zgłaszał ostrzeżenie, nawet jeśli ten typ lub ta składowa nie były nigdy zastosowane w kodzie. Java umożliwia wyciszenie ostrzeżeń dotyczących wycofywanych elementów za pomocą adnotacji `@SuppressWarnings`, ale nie można jej stosować dla instrukcji `import`. Z tego powodu nie było możliwe uniknięcie niektórych ostrzeżeń na etapie kompilacji kodu. JDK 9 nie generuje już ostrzeżeń dla deklaracji `import`. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/211>

28.10.8. Wielowydaniowe pliki JAR

Nawet po wydaniu Javy 9 wiele osób i firm będzie nadal korzystało ze starszych wersji Javy, i to przez wiele lat. Na jednym z wykładów na konferencji JavaOne w 2016 roku zapytano uczestników, której wersji Javy używają. Kilku programistów wskazało, że w ich firmie nadal korzysta się z Javy 1.4, która została wydana ponad 15 lat temu.

Dostawcy bibliotek bardzo często obsługują wiele wersji Javy. Przed Javą 9 wymagało to dostarczania osobnych plików JAR dla każdej wersji Javy. JDK 9 wprowadza obsługę wielowydaniowych plików JAR — pojedynczych plików JAR zawierających wiele wersji tej samej klasy kierowanych do różnych wydań Javy. Co więcej, wielowydaniowe pliki JAR mogą zawierać deskryptory modułów pozwalające na ich użycie wraz z systemem modułów platformy Java (rozdział 30.). Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/238>

28.10.9. Unicode 8

Java 9 obsługuje najnowszą wersję standardu Unicode (unicode.org), czyli 8. Dokonano odpowiednich modyfikacji w klasach `String` i `Character`, a także w kilku innych klasach zależnych od Unicode. Więcej informacji na ten temat znajdziesz pod adresem:

<http://openjdk.java.net/jeps/267>

28.10.10. Rozbudowa obsługi współbieżności

Dokument „JEP 266: More Concurrency Updates” dodaje nowe funkcjonalności w trzech kategoriach:

- obsługa strumieni reaktywnych — technika asynchronicznego przetwarzania strumieni — poprzez klasę `Flow` i jej zagnieżdżone interfejsy; więcej informacji na temat strumieni reaktywnych znajdziesz pod adresem:

https://en.wikipedia.org/wiki/Reactive_Streams

- różne usprawnienia, które zespół Javy zebrał od momentu wydania Javy 8;
- dodatkowe metody klasy `CompletableFuture` (zobacz listę niżej).

Nowe metody klasy *CompletableFuture*

W podrozdziale 23.14 wprowadziliśmy klasę *CompletableFuture*, która umożliwia **asynchroniczne** wykonywanie obiektów *Runnable* realizujących zadania lub obiektów *Supplier* zwracających wartości. Java 9 wzbogaca klasę *CompletableFuture* o następujące metody:

- *newIncompleteFuture*,
- *defaultExecutor*,
- *copy*,
- *minimalCompletionStage*,
- *completeAsync*,
- *orTimeout*,
- *completeOnTimeout*,
- *delayedExecutor*,
- *completedStage*,
- *failedFuture*,
- *failedStage*.

Aby poznać dodatkowe informacje związane z rozszerzeniami w kwestii współbieżności, odwiedź stronę:

<http://openjdk.java.net/jeps/266>

oraz przejrzyj dokumentację online dotyczącą *java.util.concurrent* dla Javy 9 (zawiera opis metod klasy *CompletableFuture*), a także opis powiązanych pakietów modułu *java.base*:

<http://download.java.net/java/jdk9/docs/api/overview-summary.html>

28.11. Elementy usunięte z JDK i Javy 9

Aby wspomóc przygotowanie platformy Java do modułowości, w Javie 9 usunięto kilka elementów zarówno z platformy, jak i z API. Elementy te są wymienione w podrozdziałach 8. i 9. JSR 379:

<http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html>

Usunięte funkcjonalności platformy

Podrozdział 8. JSR 379 zawiera listę usuniętych elementów platformy. Jedną ze zmian jest usunięcie mechanizmu rozszerzeń. Przed Javą 9 możliwe było umieszczenie pliku JAR biblioteki w specjalnym folderze JRE, a biblioteka stawała się dostępna dla wszystkich aplikacji Javy. Klasy w tym folderze miały gwarancję wczytania przed klasami specyficznymi dla aplikacji, więc czasem korzystano z tego mechanizmu, aby uaktualnić biblioteki do nowszych wersji. W Javie 9 mechanizm rozszerzeń zastępują **aktualizowalne moduły**:

<http://openjdk.java.net/projects/jigsaw/goals-reqs/03#upgradeablemodules>

Tego rodzaju moduły będą stosowane głównie dla standardowych technologii, które rozwijają się niezależnie od platformy Java SE, ale są dołączane do paczki zawierającej platformę. Przykładem może być tu JAXB (ang. *Java Architecture*

for XML Binding). Jeśli zostanie wydana nowa wersja JAXB, jej moduł można umieścić w opcji `--upgrade-module-path` polecenia `java`. System wykonawczy użyje nowej wersji zamiast wersji starszej dołączonej do platformy.

Usunięte metody

Podrozdział 9. JSR 379 zawiera listę metod usuniętych z różnych klas Javy w celu wspomoczenia procesu przejścia na rozwiązanie modułowe. Zgodnie z informacjami zawartymi w dokumencie JSR metody te były stosowane bardzo rzadko, a pozostawienie ich wymusiłoby umieszczenie pakietów modułu `java.desktop` w module `java.base`, co powodowałoby znacznie większy minimalny rozmiar systemu wykonawczego. Nie miałoby to sensu, bo wiele aplikacji nie wymaga modułu `java.desktop` związanego z integracją pulpitu i interfejsem graficznym.

28.12. Elementy zaproponowane do usunięcia w przyszłych wersjach Javy

Platforma Java jest używana już od ponad 20 lat. W ciągu tego okresu niektóre API były wycofywane na rzecz nowszych — czasem aby naprawić błędy lub poprawić bezpieczeństwo, ale czasem po prostu dlatego, że ulepszone API sprawiało, że starsze stawały się w zasadzie bezużyteczne. Z drugiej jednak strony większość wycofanych API — niektóre nawet usunięte już w wersji 1.2 Javy wydanej w grudniu 1998 roku — nadal pozostaje dostępna w każdym nowym wydaniu Javy, głównie w celu zachowania zgodności wstecznej.

28.12.1. Ulepszone wycofywanie

Dokument JEP 277 dostępny pod adresem:

<http://openjdk.java.net/jeps/277>

dodaje nowe funkcjonalności do adnotacji `@Deprecated`, które pozwalają programistom umieścić bardziej szczegółowe informacje o wycofywanym API, w tym informację, czy wycofanie nastąpi w następnej wersji. Rozszerzone wersje adnotacji są stosowane w całym API Javy 9, aby podkreślić w dokumentacji dostępnej online, czego nie należy używać i co zostanie usunięte z przyszłych wydań. Przykładowo wszystko, co zawiera pakiet `java.applet`, jest oznaczone do wycofania (punkt 31.12.4). Gdy więc przyjrzyysz się dokumentacji pakietu dostępnej pod adresem:

<http://download.java.net/java/jdk9/docs/api/java/applet/package-summary.html>

zauważysz uwagi o wycofaniu w opisie pakietu i przy każdej nazwie pakietu. Jeśli skorzystasz z tego API w kodzie, na etapie kompilacji otrzymasz ostrzeżenie.

28.12.2. Elementy, które prawdopodobnie zostaną usunięte w przyszłych wydaniach Javy

W podrozdziale 10. JSR 379 znajduje się lista różnych pakietów, klas, pól i metod, które prawdopodobnie będą usuwane z przyszłych wydań Javy. Dokument wskazuje, że pakiety te ewoluują niezależnie od platformy Java SE lub stanowią

część specyfikacji Java EE. Według JSR zaproponowane do usunięcia klasy, pola i metody najczęściej nie działają, nie są przydatne lub zostały zastąpione przez nowsze API.

28.12.3. Znajdowanie wycofywanych funkcjonalności

Każda strona dostępnej online dokumentacji API Javy:

<http://download.java.net/java/jdk9/docs/api/overview-summary.html>

zawiera teraz łącze *DEPRECATED*, więc można przejrzeć listę wycofywanych API dostępną pod adresem:

<http://download.java.net/java/jdk9/docs/api/deprecated-list.html>

Po kliknięciu konkretnego elementu najprawdopodobniej otrzymasz informację o powodzie wycofania elementu, a także instrukcję dotyczącą zmiany kodu.



28.2. Wskazówka zapobiegająca błędom

Unikaj korzystania z wycofanego API w nowym kodzie. Jeśli konserwujesz lub przerabiasz starszy kod Javy, uważnie sprawdź wszystkie wycofywane API i rozważ zastąpienie ich innym kodem zgodnie z alternatywami wskazanymi w dokumentacji Javy. W ten sposób będziesz mieć pewność, że kod będzie działał i kompilował się w przyszłych wersjach platformy Java.

28.12.4. Aplety Javy

Wraz z wydaniem Javy 9 API dotyczące apletów jest traktowane jako wycofywane (zobacz dokument JEP 289: <http://openjdk.java.net/jeps/289>). Dawniej aplety umożliwiały Javie działanie w środowisku przeglądarki internetowej poprzez mechanizm pluginów. Choć API to **nie** zostało jeszcze wskazane do usunięcia, może się tak stać w przyszłych wydaniach Javy. Większość przeglądarek internetowych usunęła obsługę pluginów Javy ze względu na częste problemy z bezpieczeństwem.

28.13. Podsumowanie

W tym rozdziale pokrótce przypomnieliśmy funkcjonalności Javy 9 omawiane w poprzednich rozdziałach, a następnie przeszliśmy do pozostałych tematów związanych z Javą 9. Przedstawiliśmy podstawy nowego systemu numerowania wersji Javy. zilustrowaliśmy przykładem nowe metody klasy `Matcher`: `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` i `results`. Przyjrzeliliśmy się również nowym metodom interfejsu `Stream` — `takeWhile` i `dropWhile` — oraz przeciążeniu `iterate`. Opisałyśmy zmiany dokonywane w `JavaFX 9`, w tym nowe, publiczne API dla skórek, a także rozbudowę innych funkcjonalności interfejsu graficznego. Pokazaliśmy też na przykładzie sposoby korzystania z modułów w `JShell`.

Opisałyśmy zmiany w Javie 9 związane z bezpieczeństwem i innymi funkcjonalnościami. Wspomnieliśmy o elementach, które przestają być dostępne w `JDK 9` i Javie 9. Na końcu zajęliśmy się pakietami, klasami i metodami proponowanymi do usunięcia z przyszłych wydań Javy.

