ROZDZIAŁ

31

Zaawansowane zagadnienia z obszaru JavaFX i FXML

Cele

- Określanie stylów węzłów interfejsu użytkownika za pomocą stylów CSS z JavaFX (podrozdział 31.2).
- Tworzenie krzywych kwadratowych, krzywych sześciennych i ścieżek za pomocą klas `QuadCurve`, `CubicCurve` i `Path` (podrozdział 31.3).
- Zmiana współrzędnych węzłów poprzez przesuwanie, rotowanie i skalowanie (podrozdział 31.4).
- Definiowanie obramowania kształtów za pomocą różnych pędzli (podrozdział 31.5).
- Tworzenie menu z wykorzystaniem klas `Menu`, `MenuItem`, `CheckMenuItem` i `Radio`
  ↪`MenuItem` (podrozdział 31.6).
- Tworzenie menu kontekstowego za pomocą klasy `ContextMenu` (podrozdział 31.7).
- Używanie klasy `SplitPane` do tworzenia regulowanych paneli poziomych i pionowych (podrozdział 31.8).
- Tworzenie paneli tabelowych za pomocą kontrolki `TabPane` (podrozdział 31.9).
- Tworzenie i wyświetlanie tabel za pomocą klas `TableView` i `TableColumn` (podrozdział 31.10).
- Tworzenie interfejsów użytkownika w JavaFX za pomocą języka FMXL i wizualnego kreatora scen (podrozdział 31.11).

31.1. Wprowadzenie

JavaFX umożliwia pisanie aplikacji internetowych z rozbudowanym interfejsem.

W rozdziałach 14. – 16. poznałeś podstawy JavaFX, programowania sterowanego zdarzeniami, animacji i prostych kontrolek interfejsu użytkownika. W tym rozdziale omówione są zaawansowane aspekty tworzenia aplikacji z rozbudowanym GUI.

31.2. Style CSS z JavaFX

Do określania stylów węzłów interfejsu użytkownika możesz używać stylów CSS z JavaFX.

Kaskadowe arkusze stylów z JavaFX są oparte na języku CSS definiującym styl stron internetowych. Takie style pozwalają oddzielić treść strony od jej wyglądu. Style CSS z JavaFX umożliwiają zdefiniowanie stylu interfejsu użytkownika i oddzielenie od niego treści. W pliku ze stylami CSS z JavaFX możesz zdefiniować wygląd interfejsu użytkownika — podać kolory, czcionkę, marginesy czy obramowanie komponentów. Takie pliki umożliwiają łatwe modyfikowanie stylów bez zmian w kodzie źródłowym w Javie.

Właściwości stylów w JavaFX są definiowane z przedrostkiem `-fx-`, co pozwala odróżnić je od właściwości ze zwykłych stylów CSS. Wszystkie właściwości dostępne w JavaFX są opisane w pliku *http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html*. Na listingu 31.1 pokazany jest przykładowy arkusz stylów.

LISTING 31.1. mystyle.css

```
.plaincircle {
  -fx-fill: white;
  -fx-stroke: black;
}
.circleborder {
  -fx-stroke-width: 5;
  -fx-stroke-dash-array: 12 2 4 2;
}
.border {
  -fx-border-color: black;
  -fx-border-width: 5;
}
#redcircle {
  -fx-fill: red;
  -fx-stroke: red;
}
#greencircle {
  -fx-fill: green;
  -fx-stroke: green;
}
```

W arkuszu do definiowania stylów używana jest klasa lub identyfikator. Do węzła można zastosować wiele klas stylów, a identyfikator określa unikatowy węzeł. Do definiowania klasy stylów służy składnia `.klasastylu`. Tu używane są klasy `plaincircle`, `circleborder` i `border`. Do definiowania identyfikatorów służy składnia `#iden`
`↪tyfikator`. Tu używane są identyfikatory `redcircle` i `greencircle`.

Każdy węzeł w JavaFX ma zmienną `styleClass` typu `List<String>`. Można ją pobrać za pomocą wywołania `getStyleClass()`. Do węzła można przypisać wiele klas i tylko jeden identyfikator. Każdy węzeł w JavaFX ma zmienną `id` typu `String`, której wartość można ustawić za pomocą metody `setID(string id)`. Węzłowi można przypisać tylko jeden identyfikator.

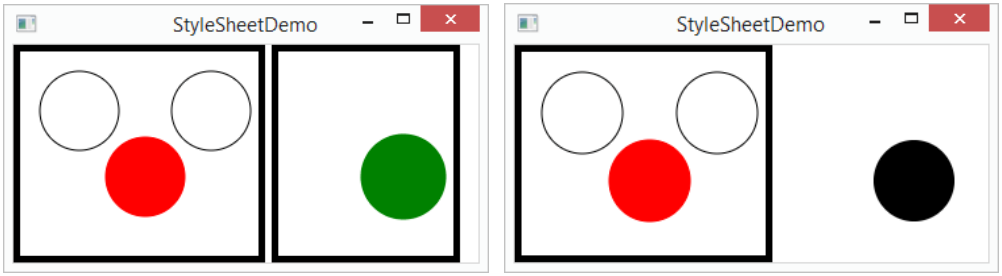
Klasy `Scene` i `Parent` mają właściwość `stylesheets`, którą można pobrać za pomocą metody `getStylesheets()`. Ta właściwość jest typu `ObservableList<String>`. Można przypisać do niej wiele arkuszy stylów. Obiekty klas `Scene` i `Parent` umożliwiają wczytywanie arkuszy stylów (zauważ, że `Parent` jest nadklasą kontenerów i kontrolek interfejsu użytkownika).

Listing 31.2 pokazuje, jak użyć arkusza stylów zdefiniowanego na listingu 31.1. Efekt pokazany jest na rysunku 31.1.

LISTING 31.2. StyleSheetDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.HBox;
 4 import javafx.scene.layout.Pane;
 5 import javafx.scene.shape.Circle;
 6 import javafx.stage.Stage;
 7
 8 public class StyleSheetDemo extends Application {
 9   @Override // Przesłanianie metody start z klasy Application
10   public void start(Stage primaryStage) {
11     HBox hBox = new HBox(5);
12     Scene scene = new Scene(hBox, 300, 250);
13     scene.getStylesheets().add("mystyle.css"); // Wczytywanie arkusza stylów
14
15     Pane pane1 = new Pane();
16     Circle circle1 = new Circle(50, 50, 30);
17     Circle circle2 = new Circle(150, 50, 30);
18     Circle circle3 = new Circle(100, 100, 30);
19     pane1.getChildren().addAll(circle1, circle2, circle3);
20     pane1.getStyleClass().add("border");
21
22    circle1.getStyleClass().add("plaincircle"); // Dodawanie klasy
23    circle2.getStyleClass().add("plaincircle"); // Dodawanie klasy
24    circle3.setId("redcircle"); // Dodawanie identyfikatora
25
26    Pane pane2 = new Pane();
27    Circle circle4 = new Circle(100, 100, 30);
28    circle4.getStyleClass().addAll("circleborder", "plainCircle");
29    circle4.setId("greencircle"); // Dodawanie klasy
30    pane2.getChildren().add(circle4);
31    pane2.getStyleClass().add("border");
32
33 hBox.getChildren().addAll(pane1, pane2);
34
35     primaryStage.setTitle("StyleSheetDemo"); // Ustawianie nagłówka
36     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
37     primaryStage.show(); // Wyświetlanie okna
38   }
39 }
```

Ten program wczytuje arkusz stylów z pliku *mystyle.css*, dodając go do właściwości `stylesheets` (wiersz 13.). Aby kod zadziałał poprawnie, plik powinien znajdować się w tym samym katalogu co kod źródłowy. Po wczytaniu arkusza stylów program przypisuje klasę `plaincircle` do obiektów `circle1` i `circle2` (wiersze 22. i 23.)



RYSUNEK 31.1. Arkusz stylów służy do określania stylów węzłów na scenie

oraz przypisuje identyfikator `redcircle` do obiektu `circle3` (wiersz 24.). W wierszach 28. i 29. program przypisuje klasy `circleborder` i `plaincircle` oraz identyfikator `greencircle` do obiektu `circle4`. Klasa `border` jest przypisywana do obiektów `pane1` i `pane2` (wiersze 20. i 31.).

Arkusz stylów jest tu dostępny dla sceny (wiersz 13.). Wszystkie węzły sceny mogą korzystać z tego arkusza. Co się stanie, jeśli usuniesz wiersz 13. i po wierszu 15. wstawisz następującą instrukcję?

```
pane1.getStylesheets().add("mystyle.css");
```

Wtedy tylko panel `pane1` i zawarte w nim węzły będą miały dostęp do arkusza stylów. Panel `pane2` i węzeł `circle4` nie będą mogły korzystać z tego arkusza. Dlatego wszystkie obiekty w panelu `pane1` będą wyświetlane w taki sam sposób jak przed zmianą, a obiekty `pane2` i `circle4` będą pokazywane bez klasy i identyfikatora stylów (rysunek 31.1b).

Zauważ, że do obiektu `circle4` przypisywane są zarówno klasa `plaincircle`, jak i identyfikator `greencircle`. Klasa `plaincircle` ustawia właściwość `fill` na kolor biały, a identyfikator `greencircle` przypisuje do tej samej właściwości kolor zielony. Ustawienia z identyfikatorów są traktowane priorytetowo względem ustawień z klas. Dlatego w programie obiekt `circle4` jest wyświetlany na zielono.

31.2.1. Jak wczytać arkusz stylów do obiektu klasy `Scene` lub `Parent`? Czy można wczytać wiele arkuszy stylów?

31.2.2. Jeśli arkusz stylów zostanie wczytany dla węzła, czy panel i wszystkie zawarte w nim węzły będą miały dostęp do tego arkusza?

31.2.3. Czy do węzła można przypisać kilka klas stylów? Czy do węzła można przypisać kilka identyfikatorów stylów?

31.2.4. Jeśli ta sama właściwość jest zdefiniowana w zastosowanych do węzła klasie i identyfikatorze, które ustawienie zostanie wybrane?

31.3. Klasy QuadCurve, CubicCurve i Path

JavaFX udostępnia klasy `QuadCurve`, `CubicCurve` i `Path` służące do tworzenia zaawansowanych kształtów.

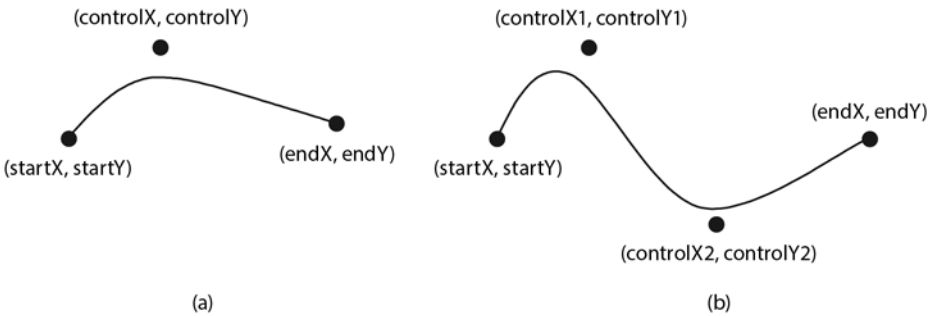
W podrozdziale 14.11 opisane zostało rysowanie prostych kształtów z użyciem klas `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon` i `Polyline`. W tym podrozdziale dowiesz się, jak rysować zaawansowane kształty z użyciem klas `CubicCurve`, `QuadCurve` i `Path`.

31.3.1. QuadCurve i CubicCurve

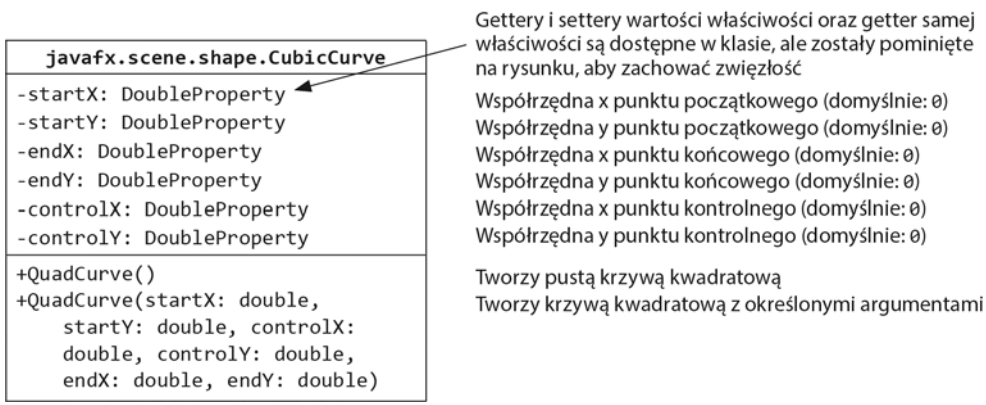
JavaFX udostępnia klasy `QuadCurve` i `CubicCurve` reprezentujące krzywe kwadratowe i sześcienne. Krzywa kwadratowa jest matematycznie definiowana jako wielomian kwadratowy. Aby utworzyć obiekt typu `QuadCurve`, użyj konstruktora bezargumentowego lub konstruktora przedstawionego poniżej:

```
QuadCurve(double startX, double startY,
  double controlX, double controlY, double endX, double endY)
```

Argumenty (`startX`, `startY`) i (`endX`, `endY`) określają dwa punkty końcowe, a (`controlX`, `controlY`) jest punktem kontrolnym. Punkt kontrolny zwykle nie znajduje się na krzywej, tylko służy do wyznaczania jej nachylenia (rysunek 31.2a). Na rysunku 31.3 pokazany jest UML-owy diagram klasy `QuadCurve`.



Rysunek 31.2. (a) Krzywa kwadratowa jest definiowana za pomocą trzech punktów; (b) Do zdefiniowania krzywej sześciennej używane są cztery punkty

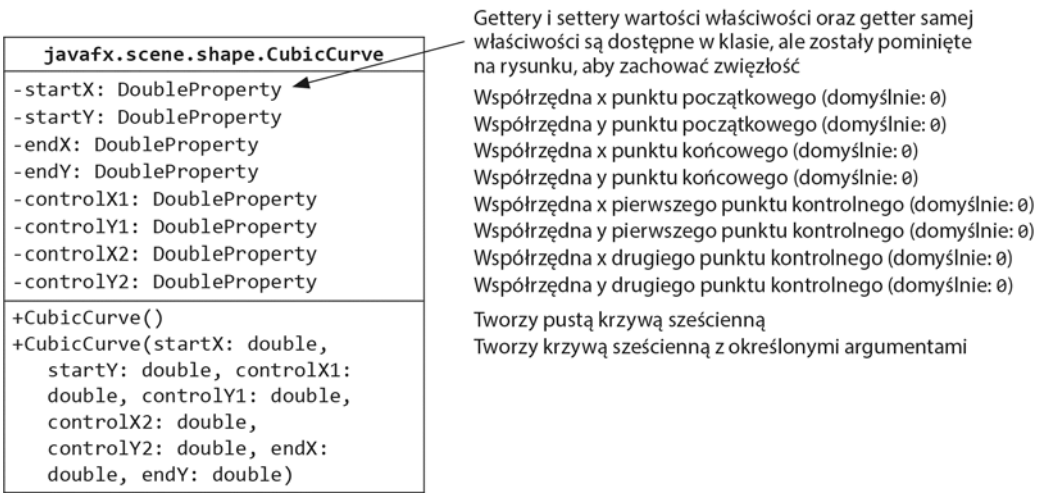


Rysunek 31.3. Klasa QuadCurve definiuje krzywą kwadratową

Krzywa sześcienna jest matematycznie definiowana jako wielomian sześcienny. Aby utworzyć obiekt typu `CubicCurve`, użyj konstruktora bezargumentowego lub konstruktora przedstawionego poniżej:

```
CubicCurve(double startX, double startY, double controlX1,
  double controlY1, double controlX2, double controlY2,
  double endX, double endY)
```

Argumenty (`startX`, `startY`) i (`endX`, `endY`) określają dwa punkty końcowe, a (`controlX1`, `controlY1`) i (`controlX2`, `controlY2`) to dwa punkty kontrolne. Punkty kontrolne zwykle nie znajdują się na krzywej, tylko służą do wyznaczania jej nachylenia (rysunek 31.2b). Na rysunku 31.4 pokazany jest UML-owy diagram klasy `CubicCurve`.



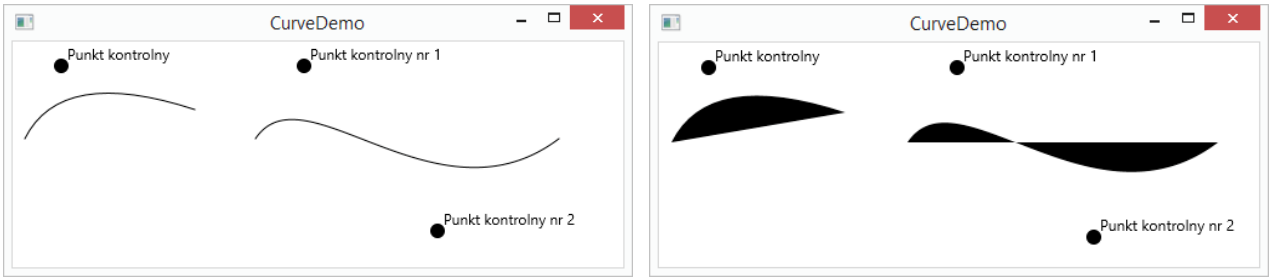
RYSUNEK 31.4. Klasa CubicCurve definiuje krzywą sześcienną

Na listingu 31.3 pokazany jest program ilustrujący, jak rysować krzywe kwadratowe i sześcienne. Na rysunku 31.5a zobaczysz przebieg tego programu.

LISTING 31.3. CurveDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.text.Text;
 5 import javafx.scene.shape.Circle;
 6 import javafx.scene.paint.Color;
 7 import javafx.scene.shape.*;
 8 import javafx.stage.Stage;
 9
10 public class CurveDemo extends Application {
11   @Override // Przesłanianie metody start z klasy Application
12   public void start(Stage primaryStage) {
13     Pane pane = new Pane();
14
15     // Tworzenie obiektu typu QuadCurve
16     QuadCurve quadCurve = new QuadCurve(10, 80, 40, 20, 150, 56);
17     quadCurve.setFill(Color.WHITE);
18     quadCurve.setStroke(Color.BLACK);
19
20     pane.getChildren().addAll(quadCurve, new Circle(40, 20, 6),
21     new Text(40 + 5, 20 - 5, "Punkt kontrolny"));
22
23     // Tworzenie obiektu typu CubicCurve
24     CubicCurve cubicCurve = new CubicCurve
```

```
25        (200, 80, 240, 20, 350, 156, 450, 80);
26      cubicCurve.setFill(Color.WHITE);
27      cubicCurve.setStroke(Color.BLACK);
28
29      pane.getChildren().addAll(cubicCurve, new Circle(240, 20, 6),
30        new Text(240 + 5, 20 - 5, "Punkt kontrolny nr 1"),
31        new Circle(350, 156, 6),
32        new Text(350 + 5, 156 - 5, "Punkt kontrolny nr 2"));
33
34      Scene scene = new Scene(pane, 300, 250);
35      primaryStage.setTitle("CurveDemo"); // Ustawianie nagłówka okna
36      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
37      primaryStage.show(); // Wyświetlanie okna
38    }
39 }
```



Rysunek 31.5. Krzywe kwadratowe i sześcienne można rysować za pomocą klas QuadCurve i CubicCurve

Ten program tworzy obiekt typu `QuadCurve` z określonymi punktami początkowym, kontrolnym i końcowym (wiersz 16.) oraz dodaje ten obiekt do panelu (wiersz 20.). Aby wskazać punkt kontrolny, program wyświetla go z wypełnieniem (wiersz 21.).

Dalej program tworzy obiekt typu `CubicCurve` z określonymi punktami początkowym, kontrolnymi i końcowym (wiersze 24. i 25.) oraz dodaje ten obiekt do panelu (wiersz 29.). Aby pokazać punkty kontrolne, program wyświetla w panelu także je (wiersze 29. – 32.).

Zauważ, że krzywe standardowo są wypełnione kolorem. Aby wyświetlić krzywe, program ustawia kolor krzywej na biały i kolor pędzla na czarny (wiersze 17. i 18. oraz 26. i 27.). Jeśli usuniesz wymienione wiersze z programu, efekt będzie taki jak na rysunku 31.5b.

31.3.2. Klasa Path

Klasa `Path` reprezentuje dowolną ścieżkę geometryczną. Ścieżka jest tworzona w wyniku dodania do niej elementów. `PathElement` jest klasą bazową dla klas elementów `MoveTo`, `HLineTo`, `VLineTo`, `LineTo`, `ArcTo`, `QuadCurveTo`, `CubicCurveTo` i `ClosePath`.

Obiekt typu `Path` można utworzyć za pomocą konstruktora bezargumentowego. Proces tworzenia ścieżki możesz potraktować jak rysowanie jej za pomocą długopisu. Ścieżka nie ma domyślnej pozycji początkowej. Należy ustawić taką pozycję, dodając do ścieżki element `MoveTo(startX, startY)`. Element `HLineTo(newX)` pozwala narysować linię poziomą od aktualnej pozycji do nowej współrzędnej *x*. Element `VLineTo(newY)` służy do rysowania linii pionowych od aktualnej pozycji do nowej współrzędnej *y*. Element `LineTo(newX, newY)` reprezentuje linię z aktualnej pozycji do nowej pozycji. Element `ArcTo(radiusX, radiusY, xAxisRotation, newX, newY, largeArcFlag, sweepArcFlag)` tworzy łuk o określonym promieniu biegnący od aktualnej pozycji do nowej pozycji.

Element `QuadCurveTo(controlX, controlY, newX, newY)` dodaje krzywą kwadratową z określonym punktem kontrolnym biegnącą od aktualnej pozycji do nowej pozycji. Element `CubicCurveTo(controlX1, controlY1, controlX2, controlY2, newX, newY)` rysuje krzywą kwadratową z dwoma punktami kontrolnymi biegnącą od aktualnej pozycji do nowej pozycji. Element `ClosePath()` kończy ścieżkę, rysując linię od punktu początkowego do punktu końcowego.

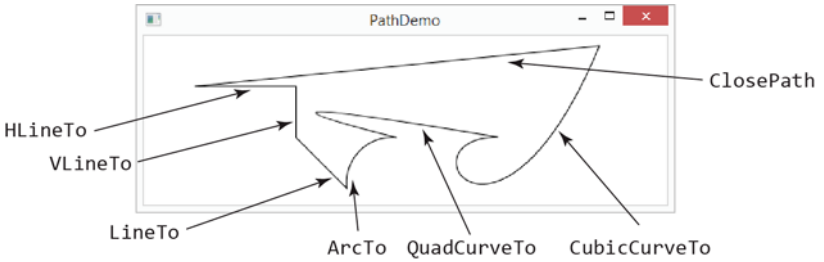
Na listingu 31.4 pokazany jest kod tworzący ścieżkę. Przykładowy przebieg programu przedstawia rysunek 31.6.

LISTING 31.4. PathDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.paint.Color;
 5 import javafx.scene.shape.*;
 6 import javafx.stage.Stage;
 7
 8 public class PathDemo extends Application {
 9   @Override // Przesłanianie metody start z klasy Application
10   public void start(Stage primaryStage) {
11     Pane pane = new Pane();
12
13     // Tworzenie ścieżki
14     Path path = new Path();
15     path.getElements().add(new MoveTo(50.0, 50.0));
16     path.getElements().add(new HLineTo(150.5));
17     path.getElements().add(new VLineTo(100.5));
18     path.getElements().add(new LineTo(200.5, 150.5));
19
20     ArcTo arcTo = new ArcTo(45, 45, 250, 100.5,
21       false, true);
22     path.getElements().add(arcTo);
23
24     path.getElements().add(new QuadCurveTo(50, 50, 350, 100));
25     path.getElements().add(
26       new CubicCurveTo(250, 100, 350, 250, 450, 10));
27
28     path.getElements().add(new ClosePath());
29
30     pane.getChildren().add(path);
31     path.setFill(null);
32     Scene scene = new Scene(pane, 300, 250);
33     primaryStage.setTitle("PathDemo"); // Ustawianie nagłówka okna
34     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
35     primaryStage.show(); // Wyświetlanie okna
36   }
37 }
```

Ten program tworzy obiekt typu `Path` (wiersz 14.), określa pozycję (wiersz 15.), po czym dodaje linię poziomą (wiersz 16.), pionową (wiersz 17.) i zwykłą (wiersz 18.). Metoda `getElements()` zwraca obiekt typu `Observable ➥List<PathElement>`.

Następnie program tworzy obiekt typu `ArcTo` (wiersze 20. i 21.). Klasa `ArcTo` zawiera właściwości `largeArcFlag` i `sweepFlag`. Domyślnie ich wartości to `false`. Możesz zmienić je na `true`, aby wyświetlić duży łuk w odwrotnym kierunku.



RYSUNEK 31.6. Aby narysować ścieżkę, dodaj jej elementy

Dalej program dodaje krzywe: kwadratową (wiersz 24.) i sześcienną (wiersze 25. i 26.), po czym zamyka ścieżkę (wiersz 28.).

Domyślnie ścieżka nie jest wypełniona kolorem. Aby podać kolor wypełnienia ścieżki, zmień wartość jej właściwości `fill`.

31.3.1. Utwórz krzywą `QuadCurve` z punktem początkowym (`100, 75.5`), punktem kontrolnym (`40, 55.5`) i punktem końcowym (`56, 80`). Ustaw właściwość `fill` na wartość `white` i właściwość `stroke` na wartość `green`.

31.3.2. Utwórz krzywą `CubicCurve` z punktem początkowym (`100, 75.5`), punktem kontrolnym nr 1 (`40, 55.5`), punktem kontrolnym nr 2 (`78.5, 25.5`) i punktem końcowym (`56, 80`). Ustaw właściwość `fill` na wartość `white` i właściwość `stroke` na wartość `green`.

31.3.3. Czy ścieżka ma domyślną pozycję początkową? Jak ustawić pozycję ścieżki?

31.3.4. Jak zamknąć ścieżkę?

31.3.5. Jak wyświetlić wypełnioną ścieżkę?

31.4. Modyfikowanie współrzędnych

JavaFX umożliwia modyfikowanie współrzędnych za pomocą przesunięć, rotacji i skalowania.

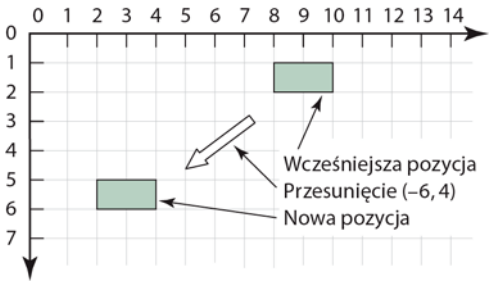
Używałeś już metody `rotate` do rotowania węzłów. Możesz też wykonywać przesunięcia i skalowanie.

31.4.1. Przesunięcia

Do przesuwania współrzędnych węzła możesz używać metod `setTranslateX(double x)`, `setTranslateY(double y)` i `setTranslateZ(double z)` z klasy `Node`. Wywołanie `setTranslateX(5)` powoduje przesunięcie węzła o 5 pikseli w prawo, a `setTranslateY(-10)` oznacza przesunięcie węzła o 10 pikseli w górę względem wcześniejszej pozycji. Na rysunku 31.7 pokazany jest prostokąt przed przesunięciem i po nim. Po wywołaniach `rectangle.setTrans`
`↪lateX(-6)` i `rectangle.setTranslateY(4)` prostokąt jest przesuwany 6 pikseli w lewo i 4 piksele w dół względem wcześniejszej pozycji. Zauważ, że modyfikowanie współrzędnych za pomocą przesunięć, rotacji i skalowania nie zmienia zawartości kształtu. Na przykład jeśli prostokąt ma wysokość 30 i szerokość 100 pikseli, to po przesunięciu nadal będzie miał takie wymiary.

LISTING 31.5. TranslationDemo.java

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
```



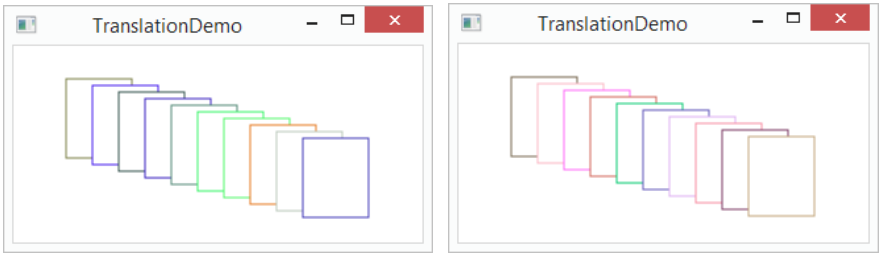
RYSUNEK 31.7. Po zastosowaniu przesunięcia (–6, 4) prostokąt jest przenoszony o określoną odległość względem poprzedniej pozycji

```
 5 import javafx.scene.shape.Rectangle;
 6 import javafx.stage.Stage;
 7
 8 public class TranslationDemo extends Application {
 9   @Override // Przesłanianie metody start z klasy Application
10   public void start(Stage primaryStage) {
11     Pane pane = new Pane();
12
13     double x = 10;
14     double y = 10;
15     java.util.Random random = new java.util.Random();
16     for (int i = 0; i < 10; i++) {
17       Rectangle rectangle = new Rectangle(10, 10, 50, 60);
18       rectangle.setFill(Color.WHITE);
19       rectangle.setStroke(Color.color(random.nextDouble(),
20         random.nextDouble(), random.nextDouble()));
21       rectangle.setTranslateX(x += 20);
22       rectangle.setTranslateY(y += 5);
23       pane.getChildren().add(rectangle);
24     }
25
26     Scene scene = new Scene(pane, 300, 250);
27     primaryStage.setTitle("TranslationDemo"); // Ustawianie nagłówka okna
28     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
29     primaryStage.show(); // Wyświetlanie okna
30   }
31 }
```

Ten program wielokrotnie tworzy 10 prostokątów (wiersz 17.). Dla każdego z nich właściwość `fill` jest ustawiana na wartość `white` (wiersz 18.), a właściwość `stroke` na losowy kolor (wiersze 19. i 20.). Kolejne prostokąty są też przesuwane w nową lokalizację (wiersze 21. i 22.). Do wyznaczania właściwości `translateX` i `translateY` używane są zmienne `x` i `y`. Te dwie zmienne mają w kolejnych prostokątach różne wartości (rysunek 31.8).

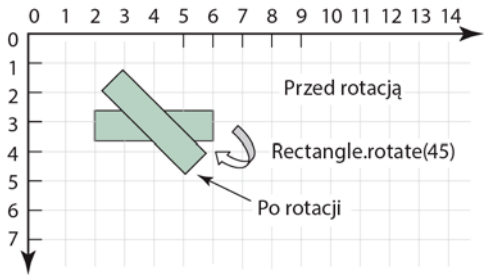
31.4.2. Rotacje

Rotacje zostały przedstawione w rozdziale 14. W tym punkcie znajdziesz ich dokładniejsze omówienie. Metoda `rotate(double theta)` z klasy `Node` pozwala zrotować węzeł o `theta` stopni względem punktu osiowego zgodnie z ruchem wskazówek zegara (`theta` to wartość typu `double`). Punkt osiowy jest wyliczany automatycznie na pod-



RYSUNEK 31.8. Kolejne prostokąty są wyświetlane w nowych lokalizacjach

stawie granic węzła. Dla kół, elips i prostokątów punktem osiowym jest środek węzła. Wywołanie `rectangle.rotate(45)` rotuje prostokąt o 45 stopni zgodnie z ruchem wskazówek zegara w kierunku wschodnim względem środka (rysunek 31.9).



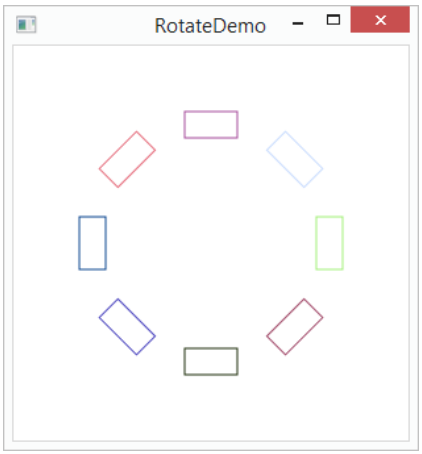
RYSUNEK 31.9. Po wywołaniu rectangle.rotate(45) prostokąt zostaje zrotowany o 45 stopni względem środka

Na listingu 31.6 pokazany jest program ilustrujący skutki rotacji współrzędnych. Rysunek 31.10 przedstawia przykładowy przebieg tego programu.

LISTING 31.6. RotateDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.paint.Color;
 5 import javafx.scene.shape.Rectangle;
 6 import javafx.stage.Stage;
 7
 8 public class RotateDemo extends Application {
 9   @Override // Przesłanianie metody start z klasy Application
10   public void start(Stage primaryStage) {
11     Pane pane = new Pane();
12     java.util.Random random = new java.util.Random();
13     // Promień koła do wyznaczania pozycji prostokątów
14     double radius = 90;
15     double width = 20; // Szerokość prostokąta
16     double height = 40; // Wysokość prostokąta
17     for (int i = 0; i < 8; i++) {
18       // Środek prostokąta
19       double x = 150 + radius * Math.cos(i * 2 * Math.PI / 8);
20       double y = 150 + radius * Math.sin(i * 2 * Math.PI / 8);
```

```
21        Rectangle rectangle = new Rectangle(
22          x - width / 2, y - height / 2, width, height);
23        rectangle.setFill(Color.WHITE);
24        rectangle.setStroke(Color.color(random.nextDouble(),
25          random.nextDouble(), random.nextDouble()));
26        rectangle.setRotate(i * 360 / 8); // Rotowanie prostokąta
27        pane.getChildren().add(rectangle);
28      }
29
30      Scene scene = new Scene(pane, 300, 300);
31      primaryStage.setTitle("RotateDemo"); // Ustawianie nagłówka okna
32      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
33      primaryStage.show(); // Wyświetlanie okna
34    }
35 }
```



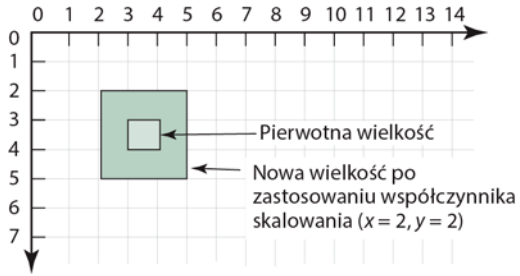
Rysunek 31.10. Metoda rotate rotuje węzeł

Ten program tworzy w pętli osiem prostokątów (wiersze 17. – 28.). Środek każdego prostokąta jest wyznaczany przez koło o środku w punkcie (150, 150) (wiersze 19. i 20.). Prostokąt jest tworzony przez określenie pozycji jego lewego górnego rogu, szerokości i wysokości (wiersze 21. i 22.). W wierszu 26. prostokąt jest rotowany, a w wierszu 27. dodawany do panelu.

31.4.3. Skalowanie

Do określenia współczynnika skalowania można użyć metod `setScaleX(double sx)`, `setScaleY(double sy)` i `set`
`↪ScaleZ(double sz)` z klasy `Node`. W zależności od współczynnika skalowania węzeł stanie się większy lub mniejszy. Skalowanie modyfikuje współrzędne węzła w taki sposób, że każda jednostka długości wzdłuż danej osi jest mnożona przez współczynnik skalowania. Podobnie jak w trakcie rotacji skalowanie powiększa lub zmniejsza węzeł względem punktu osiowego. W prostokącie punktem osiowym jest środek figury. Na przykład jeśli zastosujesz współczynnik skalowania ($x = 2$, $y = 2$), cały prostokąt (włącznie z pędzlem) stanie się dwa razy większy i „rozrośnie się” w lewo, w prawo, w górę i w dół względem środka (rysunek 31.11).

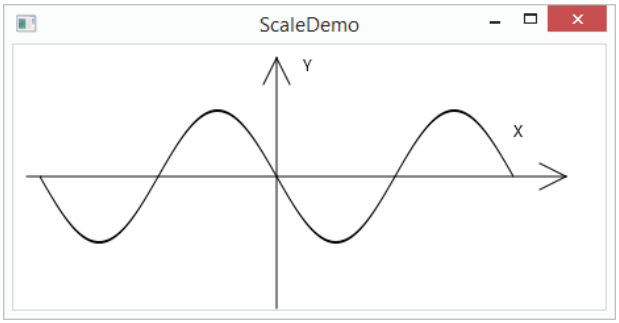
Na listingu 31.7 pokazany jest program, który ilustruje efekt zastosowania skalowania. Rysunek 31.12 przedstawia przykładowy przebieg tego programu.



RYSUNEK 31.11. Po zastosowaniu współczynnika skalowania (x = 2, y = 2) węzeł staje się dwa razy większy

LISTING 31.7. ScaleDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.shape.Line;
 5 import javafx.scene.text.Text;
 6 import javafx.scene.shape.Polyline;
 7 import javafx.stage.Stage;
 8
 9 public class ScaleDemo extends Application {
10   @Override // Przesłanianie metody start w klasie Application
11   public void start(Stage primaryStage) {
12     // Tworzenie obiektu typu Polyline w celu narysowania wykresu funkcji sinus
13     Polyline polyline = new Polyline();
14     for (double angle = -360; angle <= 360; angle++) {
15       polyline.getPoints().addAll(
16         angle, Math.sin(Math.toRadians(angle)));
17     }
18     polyline.setTranslateY(100);
19     polyline.setTranslateX(200);
20     polyline.setScaleX(0.5);
21     polyline.setScaleY(50);
22     polyline.setStrokeWidth(1.0 / 25);
23
24     // Rysowanie osi x
25     Line line1 = new Line(10, 100, 420, 100);
26     Line line2 = new Line(420, 100, 400, 90);
27     Line line3 = new Line(420, 100, 400, 110);
28
29     // Rysowanie osi y
30     Line line4 = new Line(200, 10, 200, 200);
31     Line line5 = new Line(200, 10, 190, 30);
32     Line line6 = new Line(200, 10, 210, 30);
33
34     // Rysowanie etykiet na osiach x i y
35     Text text1 = new Text(380, 70, "X");
36     Text text2 = new Text(220, 20, "Y");
37
38     // Dodawanie węzłów do panelu
39     Pane pane = new Pane();
40     pane.getChildren().addAll(polyline, line1, line2, line3, line4,
```



Rysunek 31.12. Metoda scale skaluje współrzędne węzła

```
41        line5, line6, text1, text2);
42
43      Scene scene = new Scene(pane, 450, 200);
44      primaryStage.setTitle("ScaleDemo"); // Ustawianie nagłówka okna
45      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
46      primaryStage.show(); // Wyświetlanie okna
47    }
48 }
```

Ten program tworzy obiekt typu `Polyline` (wiersz 13.) i dodaje do niego punkty, aby uzyskać wykres funkcji sinus (wiersze 14. – 17.). Ponieważ $|\sin(x)| <= 1$, współrzędne *y* są zbyt małe. Aby wykres był widoczny, program zwiększa współrzędną *y* 50-krotnie (wiersz 21.) i zmniejsza współrzędną *x* o połową (wiersz 20.).

Zauważ, że skalowanie powoduje także zmianę grubości pędzla. Aby to zrekompensować, szerokość pędzla jest celowo ustawiana na wartość `1.0/25` (wiersz 22.).

31.4.1. Czy przesuwanie współrzędnych można stosować do każdego węzła? Czy przesunięcie współrzędnych zmienia zawartość obiektu typu `Shape`?

31.4.2. Czy wywołanie `setTranslateX(6)` przypisuje do współrzędnej *x* węzła wartość 6? Czy wywołanie `setTranslateX(6)` przesuwa współrzędną *x* o 6 pikseli w prawo względem bieżącej lokalizacji?

31.4.3. Czy wywołanie `rotate(Math.PI / 2)` rotuje węzeł o 90 stopni? Czy wywołanie `rotate(90)` rotuje węzeł o 90 stopni?

31.4.4. Jak określany jest punkt osiowy na potrzeby rotacji?

31.4.5. Jaka metoda umożliwia dwukrotne zwiększenie węzła względem osi *x*?

31.5. Pędzle

Pędzel definiuje styl obramowania kształtu.

JavaFX umożliwia określanie atrybutów obramowania kształtu za pomocą metod z rysunku 31.13.

Metoda `setStroke(paint)` określa kolor pędzla. Szerokość pędzla można ustawić za pomocą metody `setStro`
`↪keWidth(width)`.

Metoda `setStrokeType(type)` określa typ pędzla. Od typu zależy, czy kreska będzie widoczna wewnątrz obramowania (`StrokeType.INSIDE`), na zewnątrz obramowania (`StrokeType.OUTSIDE`), czy na samym obramowaniu (`StrokeType.CENTERED`; jest to ustawienie domyślne). Te możliwości ilustruje rysunek 31.14.

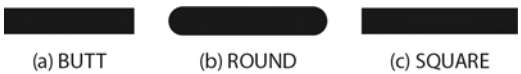
javafx.scene.shape.Shape	
`+setStroke(paint: Paint): void`	Określa kolor pędzla
`+setStrokeWidth(width: double): void`	Ustawia szerokość pędzla (domyślnie: 1)
`+setStrokeType(type: StrokeType): void`	Ustawia typ pędzla określający, czy rysowanie odbywa się wewnątrz obramowania, na zewnętrz, czy na samym obramowaniu (domyślnie: `CENTERED`)
`+setStrokeLineCap(type: StrokeLineCap): void`	Określa styl zakończeń kreski (domyślnie: `BUTT`)
`+setStrokeLineJoin(type: StrokeLineJoin): void`	Określa sposób łączenia dwóch fragmentów linii (domyślnie: `MITER`)
`+getStrokeDashArray(): ObservableList<Double>`	Zwraca listę definiującą wzór przerywanej linii
`+setStrokeDashOffset(distance: double): void`	Określa przesunięcie pierwszego fragmentu przerywanej linii

Rysunek 31.13. Klasa Shape zawiera metody do ustawiania właściwości pędzla

Rysunek 31.14. (a) Bez pędzla; (b) Kreska umieszczona wewnątrz obramowania; (c) Kreska umieszczona na obramowaniu; (d) Kreska umieszczona na zewnątrz obramowania

Zauważ, że ustawienie `CENTERED` powoduje rozszerzenie obramowania węzła o połowę wartości `strokeWidth` w obie strony (wewnątrz i na zewnątrz).

Metoda `setStrokeLineCap(capType)` określa styl końcówek kreski. Dostępne style to: `StrokeLineCap.BUTT` (domyślny), `StrokeLineCap.ROUND` i `StrokeLineCap.SQUARE`. Są one pokazane na rysunku 31.15. Styl `BUTT` kończy niedomkniętą ścieżkę bez żadnych dekoracji. Styl `ROUND` kończy niedomkniętą ścieżkę połową koła, którego promień to połowa szerokości pędzla. Styl `SQUARE` kończy niedomkniętą stronę ścieżki kwadratem wychodzącym za końcówkę ścieżki o połowę szerokości pędzla.

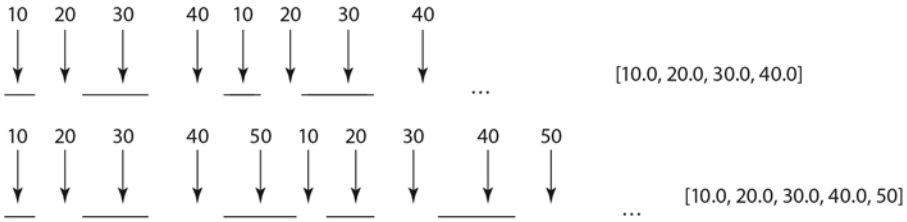


Rysunek 31.15. (a) Brak dekoracji (ustawienie BUTT); (b) Do niedomkniętej ścieżki dodawana jest połowa koła; (c) Niedomknięta ścieżka jest wydłużana o kwadrat wychodzący za końcówkę ścieżki o połowę szerokości pędzla

Metoda `setStrokeLineJoin` definiuje dekoracje stosowane w miejscach łączenia fragmentów ścieżki. Możesz używać trzech rodzajów linii pokazanych na rysunku 31.16: `StrokeLineJoin.MITER` (domyślna), `StrokeLineJoin.BEVEL` i `StrokeLineJoin.ROUND`.

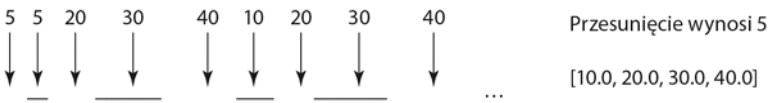
Rysunek 31.16. Fragmenty ścieżki można łączyć na trzy sposoby: (a) MITER, (b) BEVEL i (c) ROUND

Klasa `Shape` udostępnia właściwość `strokeDashArray` typu `ObservableList<Double>`. Ta właściwość służy do definiowania przerywanej linii. Kolejne wartości na liście określają długość przezroczystych i nieprzezroczystych fragmentów. Na przykład lista `[10.0, 20.0, 30.0, 40.0]` reprezentuje wzorzec widoczny na rysunku 31.17.



Rysunek 31.17. Liczby na liście oznaczają naprzemiennie przezroczyste i nieprzezroczyste fragmenty

Metoda `setStrokeDashOffset(distance)` definiuje przesunięcie pierwszej kreski we wzorcu. Na rysunku 31.18 przesunięcie jest równe `5`, a lista to `[10.0, 20.0, 30.0, 40.0]`.

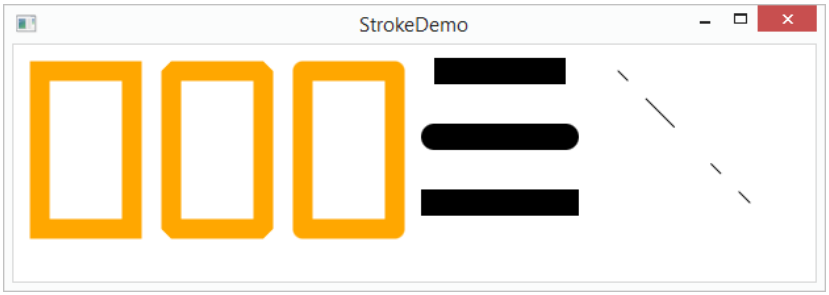


Rysunek 31.18. Przesunięcie pierwszej kreski

Na listingu 31.8 pokazany jest program ilustrujący działanie metod do ustawiania atrybutów pędzla. Rysunek 31.19 przedstawia przykładowy przebieg programu.

Listing 31.8. StrokeDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.paint.Color;
 5 import javafx.stage.Stage;
 6 import javafx.scene.shape.Rectangle;
 7 import javafx.scene.shape.*;
 8
 9 public class StrokeDemo extends Application {
10   @Override // Przesłanianie metody start z klasy Application
11   public void start(Stage primaryStage) {
12     Rectangle rectangle1 = new Rectangle(20, 20, 70, 120);
13     rectangle1.setFill(Color.WHITE);
14     rectangle1.setStrokeWidth(15);
15     rectangle1.setStroke(Color.ORANGE);
16
17     Rectangle rectangle2 = new Rectangle(20, 20, 70, 120);
18     rectangle2.setFill(Color.WHITE);
19     rectangle2.setStrokeWidth(15);
20     rectangle2.setStroke(Color.ORANGE);
21     rectangle2.setTranslateX(100);
22     rectangle2.setStrokeLineJoin(StrokeLineJoin.BEVEL);
23
24     Rectangle rectangle3 = new Rectangle(20, 20, 70, 120);
25     rectangle3.setFill(Color.WHITE);
26     rectangle3.setStrokeWidth(15);
27     rectangle3.setStroke(Color.ORANGE);
28     rectangle3.setTranslateX(200);
29     rectangle3.setStrokeLineJoin(StrokeLineJoin.ROUND);
```



RYSUNEK 31.19. Możesz określić atrybuty pędzli

```
30
31      Line line1 = new Line(320, 20, 420, 20);
32      line1.setStrokeLineCap(StrokeLineCap.BUTT);
33      line1.setStrokeWidth(20);
34
35      Line line2 = new Line(320, 70, 420, 70);
36      line2.setStrokeLineCap(StrokeLineCap.ROUND);
37      line2.setStrokeWidth(20);
38
39      Line line3 = new Line(320, 120, 420, 120);
40      line3.setStrokeLineCap(StrokeLineCap.SQUARE);
41      line3.setStrokeWidth(20);
42
43      Line line4 = new Line(460, 20, 560, 120);
44      line4.getStrokeDashArray().addAll(10.0, 20.0, 30.0, 40.0);
45
46      Pane pane = new Pane();
47      pane.getChildren().addAll(rectangle1, rectangle2, rectangle3,
48        line1, line2, line3, line4);
49
50      Scene scene = new Scene(pane, 610, 180);
51      primaryStage.setTitle("StrokeDemo"); // Ustawianie nagłówka okna
52      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
53      primaryStage.show(); // Wyświetlanie okna
54    }
55
56    // Program jest uruchamiany w wierszu poleceń
57    public static void main(String[] args) {
58      launch(args);
59    }
60  }
```

Ten program tworzy trzy prostokąty (wiersze 12. – 29.). W prostokącie nr 1 używane są domyślne połączenia (`MITER`), w prostokącie nr 2 — połączenia `BEVEL` (wiersz 22.), a w prostokącie nr 3 — połączenia `ROUND` (wiersz 29.).

Dalej program tworzy trzy linie z końcówkami `BUTT`, `ROUND` i `SQUARE` (wiersze 31. – 41.).

W wierszu 44. program tworzy linię i ustawia wzorzec reprezentujący przerywaną linię. Zauważ, że właściwość `strokeDashArray` jest typu `ObservableList<Double>`. Do listy trzeba dodać wartości typu `Double`. Dodanie wartości takiej jak 10 spowoduje błąd.

31.5.1. Metody do ustawiania pędzla i jego atrybutów są zdefiniowane w klasie `Node` czy w klasie `Shape`?
31.5.2. Jak ustawić szerokość pędzla na 3 piksele?
31.5.3. Jakie typy pędzli są dostępne? Który typ jest domyślny? Jak ustawić typ pędzla?
31.5.4. Jakie typy połączeń kresek są dostępne? Jaki jest domyślny typ połączeń? Jak ustawić typ połączenia?
31.5.5. Jakie typy zakończeń kreski są dostępne? Jaki typ zakończeń jest domyślny? Jak ustawić typ zakończenia kreski?
31.5.6. Jak określić wzorzec przerywanej linii?

31.6. Menu

JavaFX umożliwia tworzenie menu.

Menu ułatwiają wybieranie opcji i są powszechnie stosowane w aplikacjach okienkowych. JavaFX udostępnia pięć klas do tworzenia menu: `MenuBar`, `Menu`, `MenuItem`, `CheckMenuItem` i `RadioButtonMenuItem`.

`MenuBar` to komponent najwyższego poziomu używany do przechowywania menu. Menu składa się z opcji, które użytkownik może wybierać (albo włączać i wyłączać). Opcja może być obiektem typu `MenuItem`, `CheckMenu`↪`Item` lub `RadioButtonMenuItem`. Opcje można łączyć z węzłami i skrótami klawiaturowymi.

31.6.1. Tworzenie menu

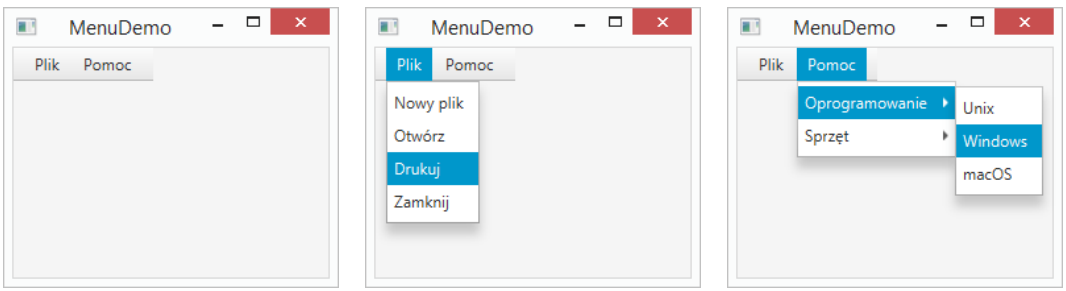
Proces implementowania menu w JavaFX wygląda tak:

1. Utwórz pasek menu i dodaj go do panelu. Poniższy kod tworzy panel i pasek menu oraz dodaje pasek do panelu:

```
MenuBar menuBar = new MenuBar();
Pane pane = new Pane();
pane.getChildren().add(menuBar);
```

2. Utwórz menu i dodaj je do paska menu. Poniższy kod tworzy dwa menu i dodaje je do paska, co ilustruje rysunek 31.20a:

```
Menu menuFile = new Menu("Plik");
Menu menuHelp = new Menu("Pomoc");
menuBar.getMenus().addAll(menuFile, menuHelp);
```



Rysunek 31.20. (a) Menu są umieszczone w pasku menu; (b) Kliknięcie menu na pasku powoduje wyświetlenie elementów menu; (c) Kliknięcie elementu menu pozwala rozwinąć powiązane elementy z podmenu

3. Utwórz elementy menu i dodaj je do menu:

```
menuFile.getItems().addAll(new MenuItem("Nowy plik"),
  new MenuItem("Otwórz"), new MenuItem("Drukuj"),
  new MenuItem("Zamknij"));
```

Ten kod dodaje elementy menu *Nowy plik*, *Otwórz*, *Drukuj* i *Zamknij* (w tej kolejności) do menu *Plik* (rysunek 31.20b).

3.1. Tworzenie podmenu.

W menu możesz zagnieżdżać inne menu (nazywane podmenu). Oto przykład:

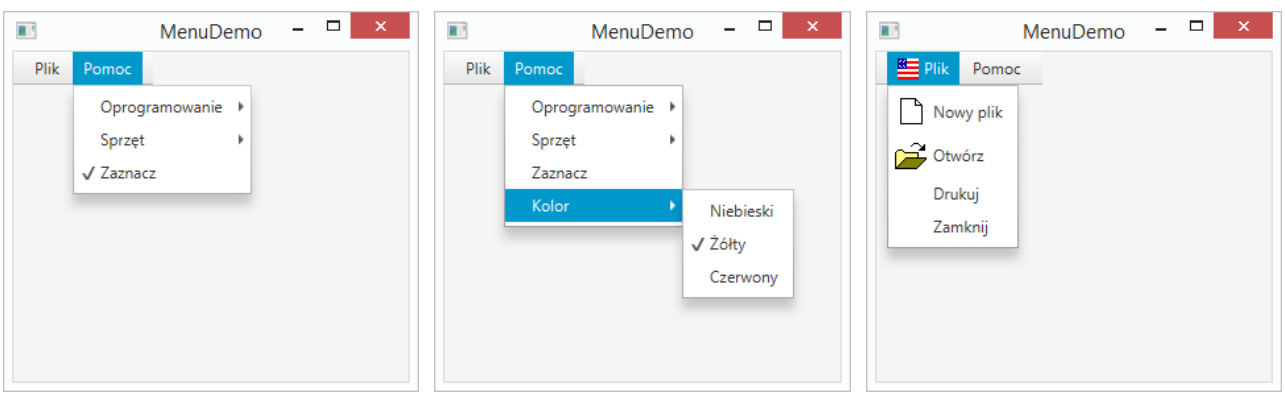
```
Menu softwareHelpSubMenu = new Menu("Oprogramowanie");
Menu hardwareHelpSubMenu = new Menu("Sprzęt");
menuHelp.getItems().add(softwareHelpSubMenu);
menuHelp.getItems().add(hardwareHelpSubMenu);
softwareHelpSubMenu.getItems().add(new MenuItem("Unix"));
softwareHelpSubMenu.getItems().add(new MenuItem("Windows"));
softwareHelpSubMenu.getItems().add(new MenuItem("macOS"));
```

Ten kod dodaje do `MenuHelp` dwa podmenu: `softwareHelpSubMenu` i `hardwareHelpSubMenu`. Do podmenu `softwareHelpSubMenu` dodawane są elementy *Unix*, *Windows* i *macOS* (rysunek 31.20c).

3.1. Tworzenie elementów z polem wyboru.

Do menu możesz dodać element typu `CheckMenuItem`. Ten typ jest podklasą klasy `MenuItem` wzbogaconą o stan logiczny i wyświetlającą „ptaszek", gdy ten stan to `true`. Możesz kliknąć taki element, aby włączyć lub wyłączyć opcję. Poniższa instrukcja dodaje element z polem wyboru *Zaznacz* (rysunek 31.21a).

```
menuHelp.getItems().add(new CheckMenuItem("Zaznacz"));
```



Rysunek 31.21. (a) Element menu z polem wyboru umożliwia włączenie lub wyłączenie opcji (podobnie jak zwykłe pole wyboru); (b) Element typu RadioMenuItem umożliwia wybór między kilkoma wykluczającymi się opcjami; (c) Do elementów menu możesz dodawać ikony i skróty klawiaturowe

3.1. Tworzenie elementów z polem radio.

Do menu możesz dodawać elementy z polem radio. Służy do tego klasa `RadioMenuItem`. Często jest ona przydatna, gdy masz w menu grupę wykluczających się opcji. Poniższe instrukcje dodają podmenu *Kolor* i przyciski opcji służące do wyboru koloru (rysunek 31.21b):

```
RadioMenuItem rmiBlue, rmiYellow, rmiRed;
colorHelpSubMenu.getItems().add(rmiBlue =
  new RadioMenuItem("Niebieski"));
colorHelpSubMenu.getItems().add(rmiYellow =
  new RadioMenuItem("Żółty"));
colorHelpSubMenu.getItems().add(rmiRed =
  new RadioMenuItem("Czerwony"));

ToggleGroup group = new ToggleGroup();
rmiBlue.setToggleGroup(group);
rmiYellow.setToggleGroup(group);
rmiRed.setToggleGroup(group);
```

4. Elementy menu generują zdarzenia `ActionEvent`. Na potrzeby ich obsługi zaimplementuj metodę `setOnAction`.
5. Ikony i skróty klawiaturowe.

   `Menu`, `CheckMenuItem` i `RadioMenuItem` to podklasy klasy `MenuItem`. `MenuItem` ma właściwość `graphic`, do której można przypisać węzeł (zwykle typu `ImageView`) wyświetlany przy elemencie menu. Klasy `Menu`, `MenuItem`, `CheckMenuItem` i `RadioMenuItem` mają też dodatkowe konstruktory, które można zastosować, by dodać grafikę. Poniższy kod dodaje grafikę do menu, elementu menu, elementu z polem wyboru i elementu z przyciskiem opcji (rysunek 31.21c).

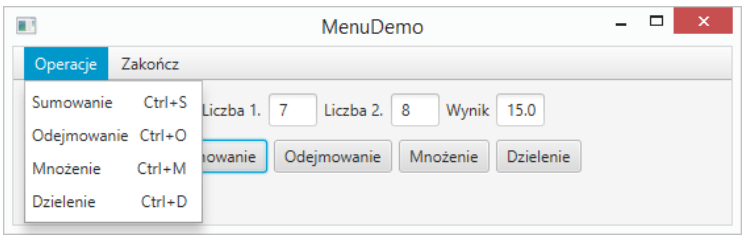
```
Menu menuFile = new Menu("Plik",
  new ImageView("image/usIcon.gif"));
MenuItem menuItemOpen = new MenuItem("Nowy plik",
  new ImageView("image/new.gif"));
CheckMenuItem checkMenuItem = new CheckMenuItem("Zaznacz",
  new ImageView("image/us.gif"));
RadioMenuItem rmiBlue = new RadioMenuItem("Niebieski",
  new ImageView("image/us.gif"));
```

6. Skrót klawiaturowy pozwala wybrać element menu bezpośrednio, przez wciśnięcie klawisza *Ctrl* i klawisza skrótu. Poniższy kod dodaje do elementu menu *Otwórz* skrót *Ctrl+O*:

```
menuItemOpen.setAccelerator(
  KeyCombination.keyCombination("Ctrl+O"));
```

31.6.2. Przykład — używanie menu

W tym punkcie zobaczysz, jak utworzyć interfejs do wykonywania operacji arytmetycznych. Ten interfejs zawiera etykiety i pola tekstowe dla liczby nr 1, liczby nr 2 i wyniku. Pole tekstowe *Wynik* wyświetla wynik operacji arytmetycznej na wartościach z pól *Liczba 1.* i *Liczba 2.* Rysunek 31.22 przedstawia przykładowy przebieg programu.



Rysunek 31.22. Operacje arytmetyczne można wykonywać, klikając przyciski lub wybierając opcje z menu Operacje

Oto etapy tworzenia tego programu (listing 31.9):

1. Utwórz pasek menu i dodaj go do panelu `VBox`. Utwórz menu *Operacje* i *Zakończ*; dodaj je do paska menu. Do menu *Operacje* dodaj elementy menu *Sumowanie*, *Odejmowanie*, *Mnożenie* i *Dzielenie*. Do menu *Zakończ* dodaj element *Zamknij*.
2. Utwórz panel `HBox` przechowujący etykiety i pola tekstowe; dodaj go do panelu `VBox`.
3. Utwórz panel `HBox` zawierający cztery przyciski: *Sumowanie*, *Odejmowanie*, *Mnożenie* i *Dzielenie*. Umieć go w panelu `VBox`.
4. Zaimplementuj metody do obsługi zdarzeń generowanych przez elementy menu i przyciski.

LISTING 31.9. MenuDemo.java

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCombination;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MenuDemo extends Application {
  private TextField tfNumber1 = new TextField();
  private TextField tfNumber2 = new TextField();
  private TextField tfResult = new TextField();

  @Override // Przesłanianie metody start z klasy Application
  public void start(Stage primaryStage) {
    MenuBar menuBar = new MenuBar();

    Menu menuOperation = new Menu("Operacje");
    Menu menuExit = new Menu("Zakończ");
    menuBar.getMenus().addAll(menuOperation, menuExit);

    MenuItem menuItemAdd = new MenuItem("Sumowanie");
    MenuItem menuItemSubtract = new MenuItem("Odejmowanie");
    MenuItem menuItemMultiply = new MenuItem("Mnożenie");
    MenuItem menuItemDivide = new MenuItem("Dzielenie");
    menuOperation.getItems().addAll(menuItemAdd, menuItemSubtract,
      menuItemMultiply, menuItemDivide);

    MenuItem menuItemClose = new MenuItem("Zamknij");
    menuExit.getItems().add(menuItemClose);

    menuItemAdd.setAccelerator(
      KeyCombination.keyCombination("Ctrl+S"));
    menuItemSubtract.setAccelerator(
      KeyCombination.keyCombination("Ctrl+O"));
    menuItemMultiply.setAccelerator(
      KeyCombination.keyCombination("Ctrl+M"));
```

```
    menuItemDivide.setAccelerator(
      KeyCombination.keyCombination("Ctrl+D"));

    HBox hBox1 = new HBox(5);
    tfNumber1.setPrefColumnCount(2);
    tfNumber2.setPrefColumnCount(2);
    tfResult.setPrefColumnCount(2);
    hBox1.getChildren().addAll(new Label("Liczba 1."), tfNumber1,
      new Label("Liczba 2."), tfNumber2, new Label("Wynik"),
      tfResult);
    hBox1.setAlignment(Pos.CENTER);

    HBox hBox2 = new HBox(5);
    Button btAdd = new Button("Sumowanie");
    Button btSubtract = new Button("Odejmowanie");
    Button btMultiply = new Button("Mnożenie");
    Button btDivide = new Button("Dzielenie");
    hBox2.getChildren().addAll(btAdd, btSubtract, btMultiply, btDivide);
    hBox2.setAlignment(Pos.CENTER);

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(menuBar, hBox1, hBox2);
    Scene scene = new Scene(vBox, 300, 250);
    primaryStage.setTitle("MenuDemo"); // Ustawianie nagłówka okna
    primaryStage.setScene(scene); // Umieszczanie sceny w oknie
    primaryStage.show(); // Wyświetlanie okna

    // Obsługa zdarzeń generowanych przez elementy menu
    menuItemAdd.setOnAction(e -> perform('+'));
    menuItemSubtract.setOnAction(e -> perform('-'));
    menuItemMultiply.setOnAction(e -> perform('*'));
    menuItemDivide.setOnAction(e -> perform('/'));
    menuItemClose.setOnAction(e -> System.exit(0));

    // Obsługa zdarzeń generowanych przez przyciski
    btAdd.setOnAction(e -> perform('+'));
    btSubtract.setOnAction(e -> perform('-'));
    btMultiply.setOnAction(e -> perform('*'));
    btDivide.setOnAction(e -> perform('/'));
  }

  private void perform(char operator) {
    double number1 = Double.parseDouble(tfNumber1.getText());
    double number2 = Double.parseDouble(tfNumber2.getText());

    double result = 0;
    switch (operator) {
      case '+': result = number1 + number2; break;
      case '-': result = number1 - number2; break;
      case '*': result = number1 * number2; break;
      case '/': result = number1 / number2; break;
    }

    tfResult.setText(result + "");
  }
}
```

Ten program tworzy pasek menu (wiersz 22.), który zawiera dwa menu: `menuOperation` i `menuExit` (wiersze 24. – 36.). Menu `menuOperation` obejmuje cztery elementy do wykonywania operacji arytmetycznych: *Sumowanie*, *Odejmowanie*, *Mnożenie* i *Dzielenie*. Menu `menuExit` zawiera element *Zamknij* kończący pracę programu. Elementy z menu *Operacje* mają przypisane skróty klawiaturowe (wiersze 38. – 45.).

Etykiety i pola tekstowe są umieszczane w panelu `HBox` (wiersze 47. – 54.), a cztery przyciski — w innym panelu tego typu (wiersze 56. – 62.). Pasek menu i dwa panele `HBox` są dodawane do panelu `VBox` (wiersz 65.), który jest umieszczany na scenie (wiersz 66.).

Użytkownik wpisuje dwie liczby w przeznaczonych na to polach. Po wybraniu operacji w menu jej wynik (dla dwóch wpisanych liczb) jest wyświetlany w polu *Wynik*. Użytkownik może też kliknąć przyciski, aby wykonać operacje.

Działania powiązane z opcjami menu i przyciskami są zapisane w wierszach 72. – 82. Prywatna metoda `perform(char operator)` (wiersze 85. – 98.) pobiera operandy z pól tekstowych *Liczba 1.* i *Liczba 2.*, stosuje operator do tych operandów i wyświetla wynik w polu tekstowym *Wynik*.

31.6.1. Jak tworzone są pasek menu, menu, element menu, element z polem wyboru i element z polem radio?

31.6.2. Jak dodać menu do paska menu? Jak umieścić w menu element menu, element z polem wyboru i element z polem radio?

31.6.3. Czy w elemencie menu można umieścić inny element menu, element z polem wyboru lub element z polem radio?

31.6.4. Jak powiązać grafikę z menu, elementem menu, elementem z polem wyboru i elementem z polem radio?

31.6.5. Jak powiązać skrót klawiaturowy *Ctrl+O* z elementem menu, elementem z polem wyboru i elementem z polem radio?

31.7. Menu kontekstowe

JavaFX umożliwia tworzenie menu kontekstowych.

Menu kontekstowe działa podobnie jak zwykłe, ale nie jest powiązane z paskiem menu i może pojawić się w dowolnym miejscu ekranu. Proces tworzenia menu kontekstowego wygląda podobnie jak w przypadku zwykłego menu. Najpierw należy utworzyć obiekt typu `ContextMenu`, a następnie dodać do niego elementy typu `MenuItem`, `Check`
`↪MenuItem` i `RadioMenuItem`. Ten kod tworzy menu `ContextMenu` i dodaje do niego elementy `MenuItem`:

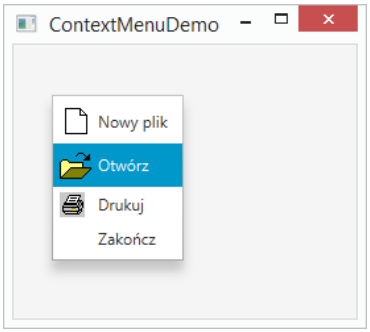
```
ContextMenu contextMenu = new ContextMenu();
ContextMenu.getItems().add(new MenuItem("Nowy"));
ContextMenu.getItems().add(new MenuItem("Otwórz"));
```

Zwykłe menu zawsze jest dodawane do paska menu, natomiast menu kontekstowe jest wiązane z węzłem nadrzędnym i wyświetlane za pomocą menu `show` z klasy `ContextMenu`. Poniżej pokazane jest, jak określić węzeł nadrzędny i lokalizację menu kontekstowego za pomocą systemu współrzędnych tego węzła:

```
contextMenu.show(node, x, y);
```

Menu kontekstowe zwykle wyświetlane jest przez wskazanie komponentu GUI i kliknięcie odpowiednim przyciskiem myszy (wyzwalaczem menu kontekstowego). Wyzwalacz menu kontekstowego zależy od systemu. W systemie Windows menu kontekstowe jest wyświetlane po zwolnieniu prawego przycisku myszy. W systemie Motif menu kontekstowe pojawia się po przytrzymaniu trzeciego przycisku myszy.

Kod z listingu 31.10 tworzy panel. Kliknięcie tego panelu odpowiednim przyciskiem powoduje wyświetlenie menu kontekstowego (rysunek 31.23).



RYSUNEK 31.23. Użycie wyzwalacza powoduje wyświetlenie menu kontekstowego

Oto etapy tworzenia programu z listingu 31.10:

1. Utwórz menu kontekstowe za pomocą klasy `ContextMenu`. Utwórz elementy *Nowy*, *Otwórz*, *Drukuj* i *Zakończ* typu `MenuItem`.
2. Dodaj elementy menu do menu kontekstowego.
3. Utwórz panel i umieść go na scenie.
4. Zaimplementuj metody obsługi zdarzeń generowanych przez elementy menu.
5. Zaimplementuj metodę obsługi zdarzenia `mousePressed` do wyświetlania menu kontekstowego.

LISTING 31.10. ContextMenuDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.control.ContextMenu;
 4 import javafx.scene.control.MenuItem;
 5 import javafx.scene.image.ImageView;
 6 import javafx.scene.layout.Pane;
 7 import javafx.stage.Stage;
 8
 9 public class ContextMenuDemo extends Application {
10   @Override // Przesłanianie metody start z klasy Application
11   public void start(Stage primaryStage) {
12     ContextMenu contextMenu = new ContextMenu();
13     MenuItem menuItemNew = new MenuItem("Nowy",
14       new ImageView("image/new.gif"));
15     MenuItem menuItemOpen = new MenuItem("Otwórz",
16       new ImageView("image/open.gif"));
17     MenuItem menuItemPrint = new MenuItem("Drukuj",
18       new ImageView("image/print.gif"));
19     MenuItem menuItemExit = new MenuItem("Zakończ");
20     contextMenu.getItems().addAll(menuItemNew, menuItemOpen,
21       menuItemPrint, menuItemExit);
22
23     Pane pane = new Pane();
24     Scene scene = new Scene(pane, 300, 250);
25     primaryStage.setTitle("ContextMenuDemo"); // Ustawianie nagłówka okna
26     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
27     primaryStage.show(); // Wyświetlanie okna
```

```
28
29      pane.setOnMousePressed(
30        e -> contextMenu.show(pane, e.getScreenX(), e.getScreenY()));
31
32      menuItemNew.setOnAction(e -> System.out.println("Nowy"));
33      menuItemOpen.setOnAction(e -> System.out.println("Otwórz"));
34      menuItemPrint.setOnAction(e -> System.out.println("Drukuj"));
35      menuItemExit.setOnAction(e -> System.exit(0));
36    }
37 }
```

Proces tworzenia menu kontekstowego wygląda podobnie jak budowanie zwykłych menu. Aby uzyskać menu kontekstowe, utwórz menu typu `ContextMenu` (wiersz 12.), a następnie dodaj do niego elementy typu `MenuItem` (wiersze 13. – 21.).

Aby wyświetlić menu kontekstowe, użyj metody `show` i podaj węzeł nadrzędny oraz lokalizację menu kontekstowego (wiersze 29. i 30.). Metoda `show` jest wywoływana, gdy menu kontekstowe zostaje uruchomione w wyniku kliknięcia panelu myszą (wiersz 30.).

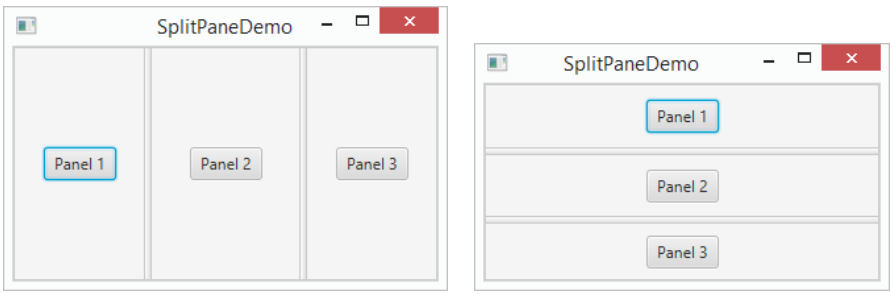
31.7.1. Jak utworzyć menu kontekstowe? Jak dodać do niego elementy menu, elementy z polem wyboru i elementy z polem radio?

31.7.2. Jak wyświetlić menu kontekstowe?

31.8. Panele SplitPane

Za pomocą klasy `SplitPane` można wyświetlić kilka paneli, których wielkość użytkownik może regulować.

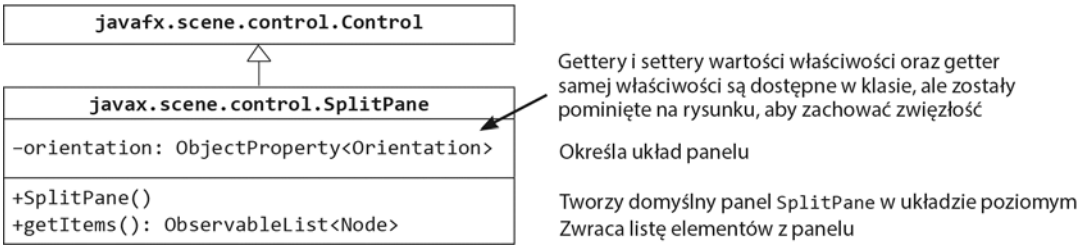
Kontrolka `SplitPane` obejmuje dwa komponenty rozdzielone paskiem podziału (rysunek 31.24).



Rysunek 31.24. Panel SplitPane dzieli kontener na dwie części

Fragmenty rozdzielone paskiem mogą być ułożone w poziomie lub w pionie. Pasek dzielący oba fragmenty można przeciągnąć, aby zmienić ilość miejsca zajmowanego przez każdy fragment. Na rysunku 31.25 wymienione są często używane właściwości, konstruktory i metody z klasy `SplitPane`.

Program z listingu 31.11 umożliwia wybranie państwa za pomocą przycisków radio, po czym wyświetla flagę i opis kraju w odrębnych polach (rysunek 31.26). W polu tekstowym wyświetlany jest opis aktualnie wybranego państwa. Przyciski radio, zwykłe przyciski i obszar tekstowy są umieszczone w dwóch panelach `SplitPane`.

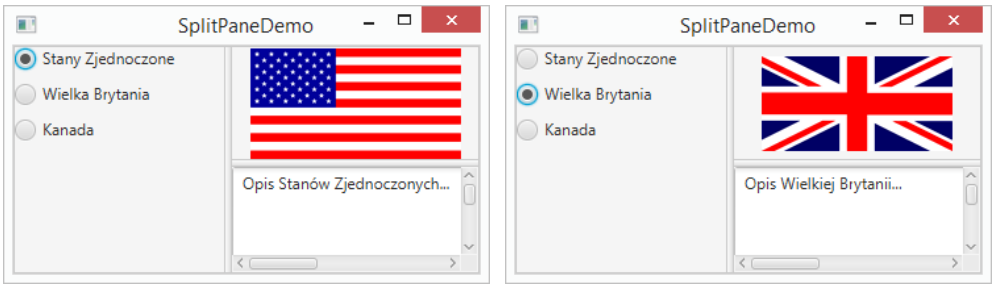


RYSUNEK 31.25. Klasa SplitPane udostępnia metody do określania właściwości panelu i operowania jego fragmentami

LISTING 31.11. SplitPaneDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.geometry.Orientation;
 3 import javafx.scene.Scene;
 4 import javafx.scene.control.RadioButton;
 5 import javafx.scene.control.ScrollPane;
 6 import javafx.scene.control.SplitPane;
 7 import javafx.scene.control.TextArea;
 8 import javafx.scene.control.ToggleGroup;
 9 import javafx.scene.image.Image;
10 import javafx.scene.image.ImageView;
11 import javafx.scene.layout.StackPane;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class SplitPaneDemo extends Application {
16   private Image usImage = new Image(
17     "https://liveexample.pearsoncmg.com/common/image/us.gif");
18   private Image ukImage = new Image(
19     "https://liveexample.pearsoncmg.com/common/image/uk.gif");
20   private Image caImage = new Image(
21     "https://liveexample.pearsoncmg.com/common/image/ca.gif");
22   private String usDescription = "Opis USA...";
23   private String ukDescription = "Opis Wielkiej Brytanii...";
24   private String caDescription = "Opis Kanady...";
25
26   @Override // Przesłanianie metody start z klasy Application
27   public void start(Stage primaryStage) {
28     VBox vBox = new VBox(10);
29     RadioButton rbUS = new RadioButton("USA");
30     RadioButton rbUK = new RadioButton("Wielka Brytania");
31     RadioButton rbCA = new RadioButton("Kanada");
32     vBox.getChildren().addAll(rbUS, rbUK, rbCA);
33
34     SplitPane content = new SplitPane();
35     content.setOrientation(Orientation.VERTICAL);
36     ImageView imageView = new ImageView(usImage);
37     StackPane imagePane = new StackPane();
38     imagePane.getChildren().add(imageView);
39     TextArea taDescription = new TextArea();
40     taDescription.setText(usDescription);
41     content.getItems().addAll(
42       imagePane, new ScrollPane(taDescription));
```

```
43
44      SplitPane sp = new SplitPane();
45      sp.getItems().addAll(vBox, content);
46
47      Scene scene = new Scene(sp, 300, 250);
48      primaryStage.setTitle("SplitPaneDemo"); // Ustawianie nagłówka okna
49      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
50      primaryStage.show(); // Wyświetlanie okna
51
52      // Grupowanie przycisków radio
53      ToggleGroup group = new ToggleGroup();
54      rbUS.setToggleGroup(group);
55      rbUK.setToggleGroup(group);
56      rbCA.setToggleGroup(group);
57
58      rbUS.setSelected(true);
59      rbUS.setOnAction(e -> {
60        imageView.setImage(usImage);
61        taDescription.setText(usDescription);
62      });
63
64      rbUK.setOnAction(e -> {
65        imageView.setImage(ukImage);
66        taDescription.setText(ukDescription);
67      });
68
69      rbCA.setOnAction(e -> {
70        imageView.setImage(caImage);
71        taDescription.setText(caDescription);
72      });
73    }
74 }
```



Rysunek 31.26. Możesz zmieniać wielkość komponentów w panelach SplitPane

Ten program umieszcza trzy przyciski opcji w panelu `VBox` (wiersze 28. – 32.) i tworzy pionowy panel `SplitPane` do przechowywania widoku obrazu i obszaru tekstowego (wiersze 34. – 42.). Panele `SplitPane` można zagnieżdżać jeden w drugim. Tu program tworzy poziomy panel `SplitPane` i umieszcza w nim panel `VBox` oraz poziomy panel `SplitPane` (wiersze 44. i 45.).

Dodanie panelu `SplitPane` do istniejącego panelu tego typu skutkuje uzyskaniem trzech komponentów. Program tworzy tu dwa panele `SplitPane` (wiersze 34. i 42.) zawierające komponenty z przyciskami radio, zwykłymi przyciskami i panelem `ScrollPane`.

W wierszach 53. – 56. program grupuje przyciski radio, a wierszach 59. – 72. określa operacje powiązane z tymi przyciskami.

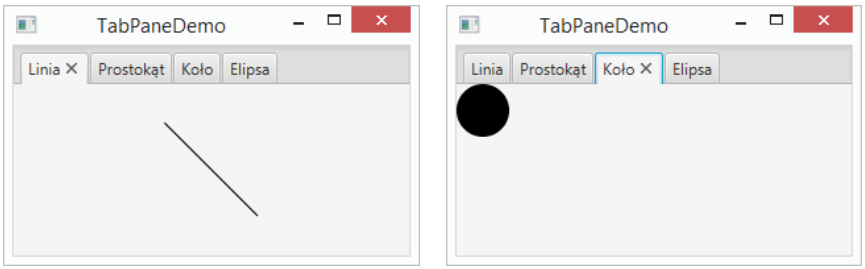
31.8.1. Jak utworzyć poziomy panel `SplitPane`? Jak utworzyć pionowy panel tego typu?

31.8.2. Jak dodać elementy do panelu `SplitPane`? Czy można dodać panel `SplitPane` do innego panelu tego typu?

31.9. Panele TabPane

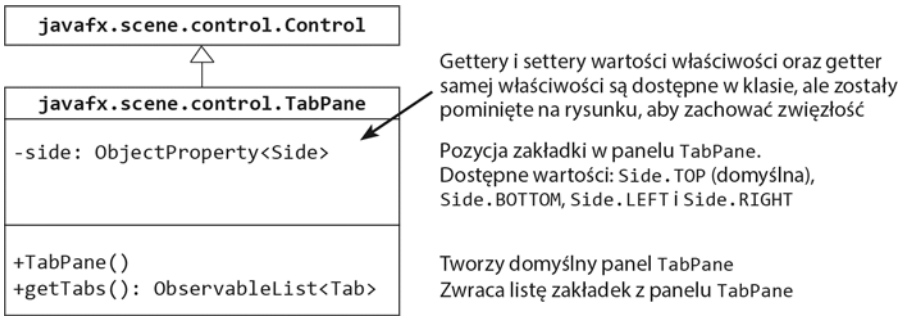
Klasa TabPane umożliwia wyświetlanie wielu paneli w zakładkach.

`TabPane` jest przydatną kontrolką wyświetlającą zakładki (rysunek 31.27). Te zakładki można przełączać, a w danym momencie widoczna jest zawartość tylko jednej z nich. Do panelu `TabPane` można dodawać zakładki umieszczane w górnej, lewej, dolnej lub prawej części panelu.



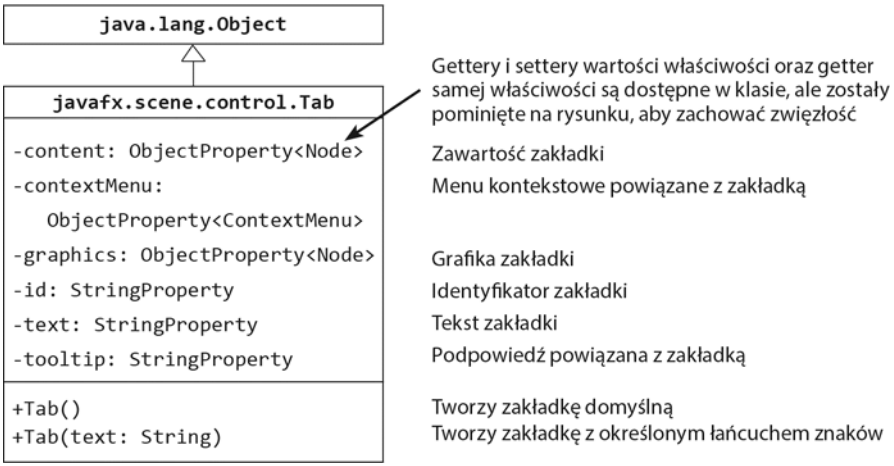
Rysunek 31.27. Panel TabPane przechowuje grupy zakładek

Każda zakładka reprezentuje jedną stronę. Do definiowania zakładek służy klasa `Tab`. Zakładki mogą zawierać dowolne węzły: panele, kształty, kontrolki itd. Dzięki temu możesz utworzyć wielopoziomowy panel z zakładkami. Na rysunkach 31.28 i 31.29 opisane są często używane właściwości, konstruktory i metody z klas `TabPane` i `Tab`.



Rysunek 31.28. Klasa TabPane wyświetla zakładki i zarządza nimi

Kod z listingu 31.12 tworzy panel `TabPane` do wyświetlania czterech rodzajów figur: linii, prostokąta, prostokąta z zaokrąglonymi rogami i owalu. Możesz wybrać wyświetlaną figurę, klikając odpowiednią zakładkę (rysunek 31.27).



RYSUNEK 31.29. Zakładka zawiera węzły

LISTING 31.12. TabPaneDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.control.Tab;
 4 import javafx.scene.control.TabPane;
 5 import javafx.scene.layout.StackPane;
 6 import javafx.scene.shape.Circle;
 7 import javafx.scene.shape.Ellipse;
 8 import javafx.scene.shape.Line;
 9 import javafx.scene.shape.Rectangle;
10 import javafx.stage.Stage;
11
12 public class TabPaneDemo extends Application {
13   @Override // Przesłanianie metody start z klasy Application
14   public void start(Stage primaryStage) {
15     TabPane tabPane = new TabPane();
16     Tab tab1 = new Tab("Linia");
17     StackPane pane1 = new StackPane();
18     pane1.getChildren().add(new Line(10, 10, 80, 80));
19     tab1.setContent(pane1);
20     Tab tab2 = new Tab("Prostokąt");
21     tab2.setContent(new Rectangle(10, 10, 200, 200));
22     Tab tab3 = new Tab("Koło");
23     tab3.setContent(new Circle(50, 50, 20));
24     Tab tab4 = new Tab("Owal");
25     tab4.setContent(new Ellipse(10, 10, 100, 80));
26     tabPane.getTabs().addAll(tab1, tab2, tab3, tab4);
27
28     Scene scene = new Scene(tabPane, 300, 250);
29     primaryStage.setTitle("DisplayFigure"); // Ustawianie nagłówka okna
30     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
31     primaryStage.show(); // Wyświetlanie okna
32   }
33 }
```

Ten program tworzy panel `TabPane` (wiersz 15.) i cztery zakładki (wiersze 16., 20., 22. i 24.). W wierszu 18. tworzony jest panel `StackPane` do przechowywania linii; w wierszu 19. program umieszcza ten panel w zakładce `tab1`. W zakładkach `tab2`, `tab3` i `tab4` umieszczane są prostokąt, koło i owal. Zauważ, że linia w zakładce `tab1` jest wyśrodkowana, ponieważ znajduje się w panelu `StackPane`. Pozostałe figury są umieszczane bezpośrednio w zakładkach i pojawiają się w ich lewym górnym rogu.

Domyślnie zakładki są umieszczane w górnej części panelu `TabPane`. Ich lokalizację możesz zmienić za pomocą metody `setSide`.

31.9.1. Jak utworzyć panel `TabPane`? Jak utworzyć zakładkę? Jak dodać zakładkę do panelu `TabPane`?

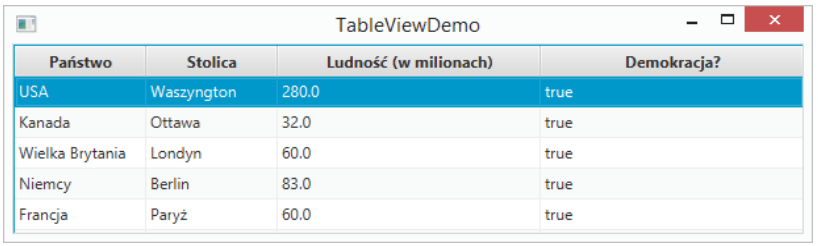
31.9.2. Jak umieścić zakładki po lewej stronie panelu `TabPane`?

31.9.3. Czy sama zakładka może zawierać zarówno tekst, jak i grafikę? Napisz kod do przypisywania grafiki do zakładki `tab1` z listingu 31.12.

31.10. TableView

Za pomocą klasy `TableView` można wyświetlać tabele.

Kontrolka `TableView` wyświetla dane w wierszach i kolumnach jako dwuwymiarową siatkę (rysunek 31.30).



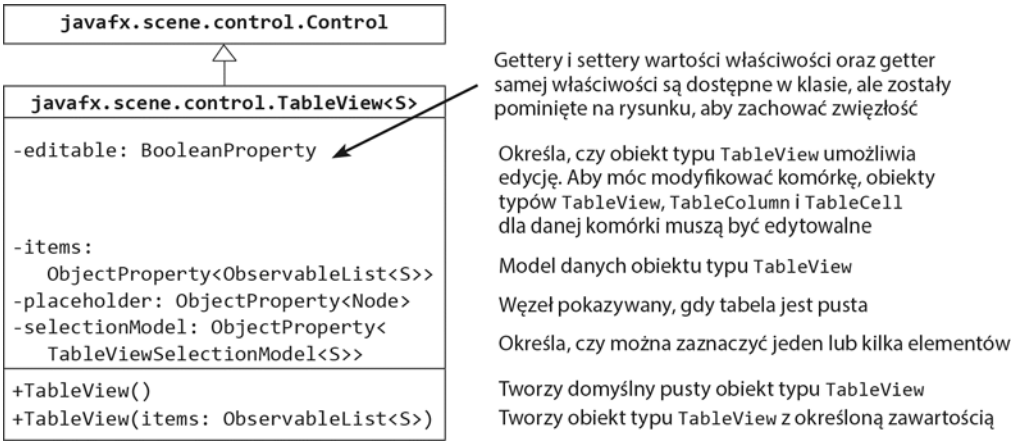
RYSUNEK 31.30. Klasa TableView wyświetla dane w tabeli

Do wyświetlania tabeli i operowania nią służą klasy `TableView`, `TableColumn` i `TableCell`. Klasa `TableView` wyświetla tabelę, `TableColumn` definiuje kolumny tabeli, a `TableCell` reprezentuje komórki. Budowanie obiektu typu `TableView` to wieloetapowy proces. Najpierw trzeba utworzyć obiekt tego typu i powiązać z nim dane. W drugim kroku należy utworzyć kolumny za pomocą klasy `TableColumn` i skonfigurować fabrykę wartości komórek, aby określić, jak zapełnić wszystkie komórki w jednej kolumnie.

Na listingu 31.13 pokazane jest, jak używać klas `TableView` i `TableColumn`. Przykładowy przebieg programu jest przedstawiony na rysunku 31.31.

LISTING 31.13. TableViewDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.beans.property.SimpleBooleanProperty;
 3 import javafx.beans.property.SimpleDoubleProperty;
 4 import javafx.beans.property.SimpleStringProperty;
 5 import javafx.collections.FXCollections;
 6 import javafx.collections.ObservableList;
 7 import javafx.scene.Scene;
 8 import javafx.scene.control.TableColumn;
 9 import javafx.scene.control.TableView;
10 import javafx.scene.control.cell.PropertyValueFactory;
```



RYSUNEK 31.31. Obiekt typu TableView wyświetla tabelę

```
11 import javafx.scene.layout.Pane;
12 import javafx.stage.Stage;
13
14 public class TableViewDemo extends Application {
15   @Override // Przesłanianie metody start z klasy Application
16   public void start(Stage primaryStage) {
17     TableView<Country> tableView = new TableView<>();
18     ObservableList<Country> data =
19       FXCollections.observableArrayList(
20         new Country("USA", "Waszyngton", 280, true),
21         new Country("Kanada", "Ottawa", 32, true),
22         new Country("Wielka Brytania", "Londyn", 60, true),
23         new Country("Niemcy", "Berlin", 83, true),
24         new Country("Francja", "Paryż", 60, true));
25     tableView.setItems(data);
26
27     TableColumn countryColumn = new TableColumn("Państwo");
28     countryColumn.setMinWidth(100);
29     countryColumn.setCellValueFactory(
30       new PropertyValueFactory<Country, String>("country"));
31
32     TableColumn capitalColumn = new TableColumn("Stolica");
33     capitalColumn.setMinWidth(100);
34     capitalColumn.setCellValueFactory(
35       new PropertyValueFactory<Country, String>("capital"));
36
37     TableColumn populationColumn =
38       new TableColumn("Ludność (w milionach)");
39     populationColumn.setMinWidth(200);
40     populationColumn.setCellValueFactory(
41       new PropertyValueFactory<Country, Double>("population"));
42
43     TableColumn democraticColumn =
44       new TableColumn("Demokracja?");
45     democraticColumn.setMinWidth(200);
46     democraticColumn.setCellValueFactory(
```

```
47        new PropertyValueFactory<Country, Boolean>("democratic"));
48
49      tableView.getColumns().addAll(countryColumn, capitalColumn,
50        populationColumn, democraticColumn);
51
52      Pane pane = new Pane();
53      pane.getChildren().add(tableView);
54      Scene scene = new Scene(pane, 300, 250);
55      primaryStage.setTitle("TableViewDemo"); // Ustawianie nagłówka okna
56      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
57      primaryStage.show(); // Wyświetlanie okna
58    }
59
60    public static class Country {
61      private final SimpleStringProperty country;
62      private final SimpleStringProperty capital;
63      private final SimpleDoubleProperty population;
64      private final SimpleBooleanProperty democratic;
65
66      private Country(String country, String capital,
67          double population, boolean democratic) {
68        this.country = new SimpleStringProperty(country);
69        this.capital = new SimpleStringProperty(capital);
70        this.population = new SimpleDoubleProperty(population);
71        this.democratic = new SimpleBooleanProperty(democratic);
72      }
73
74      public String getCountry() {
75        return country.get();
76      }
77
78      public void setCountry(String country) {
79        this.country.set(country);
80      }
81
82      public String getCapital() {
83        return capital.get();
84      }
85
86      public void setCapital(String capital) {
87        this.capital.set(capital);
88      }
89
90      public double getPopulation() {
91        return population.get();
92      }
93
94      public void setPopulation(double population) {
95        this.population.set(population);
96      }
97
98      public boolean isDemocratic() {
99        return democratic.get();
100     }
101
102     public void setDemocratic(boolean democratic) {
```

```
103       this.democratic.set(democratic);
104     }
105   }
106 }
```

Ten program tworzy obiekt klasy `TableView` (wiersz 17.). Jest to klasa generyczna, konkretyzowana tu za pomocą typu `Country`. Utworzony obiekt służy więc do wyświetlania państw. Używane dane są typu `ObservableList<Country>`. Program tworzy listę tego typu (wiersze 18. – 24.) i wiąże ją z obiektem typu `TableView` (wiersz 25.).

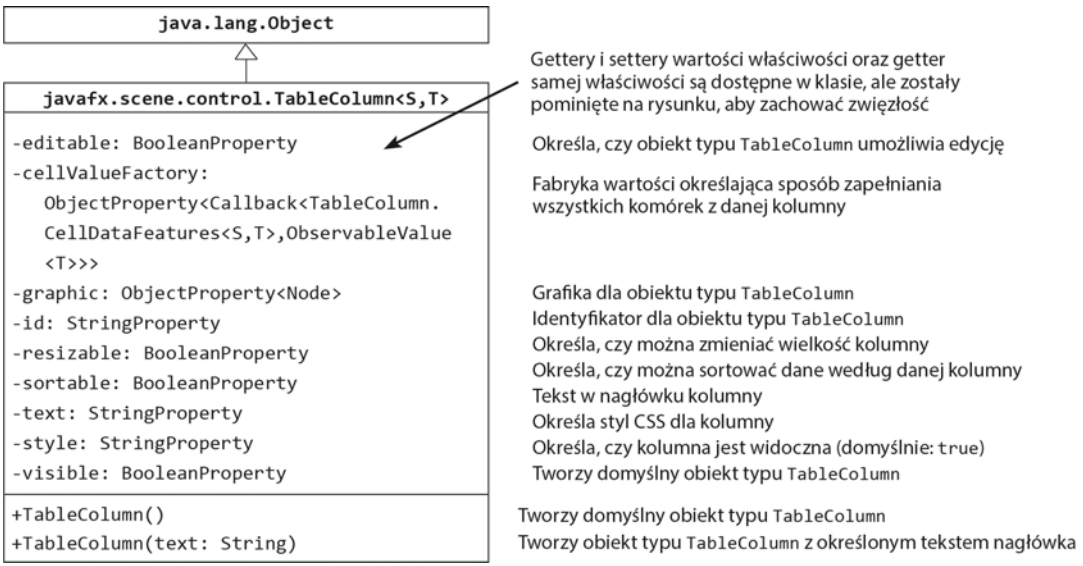
Dalej program tworzy obiekt typu `TableColumn` dla każdej kolumny tabeli (wiersze 27. – 47.). Dla każdej kolumny tworzony i konfigurowany jest też obiekt typu `PropertyValueFactory` (wiersz 30.) służący do zapełniania kolumn danymi. `PropertyValueFactory<S, T>` jest klasą generyczną. `S` reprezentuje klasę wyświetlaną w obiekcie typu `TableView`, a `T` określa klasę z wartościami z kolumny. Obiekt typu `PropertyValueFactory` wiąże właściwość z klasy `S` z kolumną.

Gdy tworzysz tabelę w aplikacji opartej na JavaFX, najlepiej jest zdefiniować model danych w klasie. Klasa `Country` definiuje dane wyświetlane w obiekcie typu `TableView`. Każda właściwość z klasy `Country` definiuje kolumnę tabeli. Te właściwości można zdefiniować jako właściwości wiążące, udostępniające gettery i settery wartości.

Program dodaje kolumny do obiektu typu `TableView` (wiersze 49. i 50.), dodaje ten obiekt do panelu (wiersz 53.) i umieszcza panel na scenie (wiersz 54.). Zauważ, że wiersz 31. można uprościć:

```
new PropertyValueFactory<>("country");
```

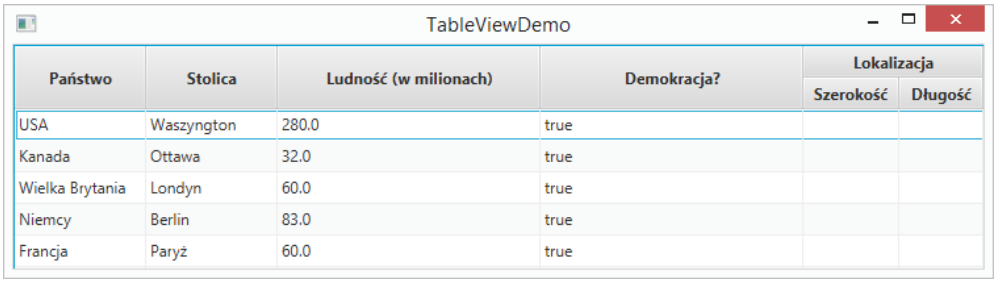
Ten przykład pokazuje, jak wyświetlać dane w tabeli za pomocą klas `TableView` i `TableColumn`. Często używane właściwości i metody tych klas znajdziesz na rysunkach 31.32 i 31.33.



Rysunek 31.32. Klasa TableColumn definiuje kolumnę z obiektu typu TableView

Można tworzyć kolumny zagnieżdżone. Poniższy kod tworzy dwie podkolumny w kolumnie *Lokalizacja* (rysunek 31.33).

```
TableColumn locationColumn = new TableColumn("Lokalizacja");
locationColumn.getColumns().addAll(new TableColumn("Szerokość"),
  new TableColumn("Długość"));
```



RYSUNEK 31.33. Możesz dodawać podkolumny w kolumnie

Modelem danych obiektu typu `TableView` jest lista obserwowalna. Modyfikacje w danych są automatycznie odzwierciedlane w tabeli. Kod z listingu 31.14 umożliwia użytkownikom dodawanie nowych wierszy do tabeli.

LISTING 31.14. AddNewRowDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.beans.property.SimpleBooleanProperty;
 3 import javafx.beans.property.SimpleDoubleProperty;
 4 import javafx.beans.property.SimpleStringProperty;
 5 import javafx.collections.FXCollections;
 6 import javafx.collections.ObservableList;
 7 import javafx.scene.Scene;
 8 import javafx.scene.control.Button;
 9 import javafx.scene.control.CheckBox;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TableColumn;
12 import javafx.scene.control.TableView;
13 import javafx.scene.control.TextField;
14 import javafx.scene.control.cell.PropertyValueFactory;
15 import javafx.scene.layout.BorderPane;
16 import javafx.scene.layout.FlowPane;
17 import javafx.stage.Stage;
18
19 public class AddNewRowDemo extends Application {
20   @Override // Przesłanianie metody start z klasy Application
21   public void start(Stage primaryStage) {
22     TableView<Country> tableView = new TableView<>();
23     ObservableList<Country> data =
24       FXCollections.observableArrayList(
25         new Country("USA", "Waszyngton", 280, true),
26         new Country("Kanada", "Ottawa", 32, true),
27         new Country("Wielka Brytania", "Londyn", 60, true),
28         new Country("Niemcy", "Berlin", 83, true),
29         new Country("Francja", "Paryż", 60, true));
30     tableView.setItems(data);
31
32     TableColumn countryColumn = new TableColumn("Państwo");
33     countryColumn.setMinWidth(100);
34     countryColumn.setCellValueFactory(
35       new PropertyValueFactory<Country, String>("country"));
36
37     TableColumn capitalColumn = new TableColumn("Stolica");
```

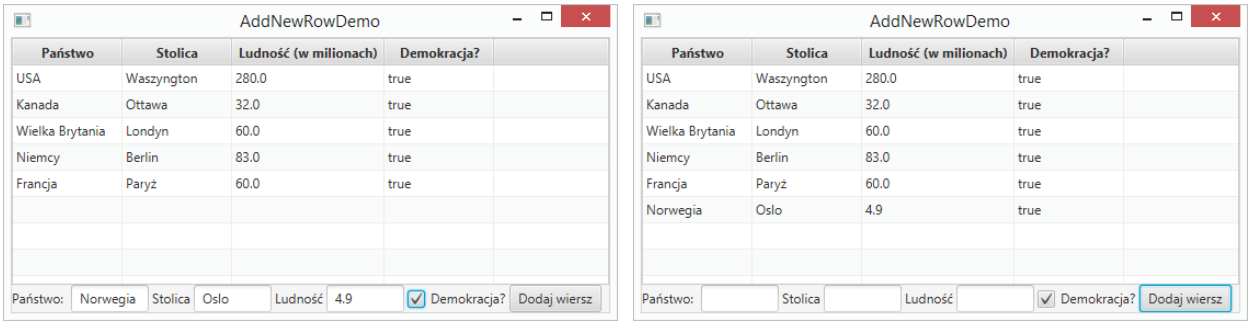
```
38      capitalColumn.setMinWidth(100);
39      capitalColumn.setCellValueFactory(
40        new PropertyValueFactory<Country, String>("capital"));
41
42      TableColumn populationColumn =
43        new TableColumn("Ludność (w milionach)");
44      populationColumn.setMinWidth(100);
45      populationColumn.setCellValueFactory(
46        new PropertyValueFactory<Country, Double>("population"));
47
48      TableColumn democraticColumn =
49        new TableColumn("Demokracja?");
50      democraticColumn.setMinWidth(100);
51      democraticColumn.setCellValueFactory(
52        new PropertyValueFactory<Country, Boolean>("democratic"));
53
54      tableView.getColumns().addAll(countryColumn, capitalColumn,
55        populationColumn, democraticColumn);
56
57      FlowPane flowPane = new FlowPane(3, 3);
58      TextField tfCountry = new TextField();
59      TextField tfCapital = new TextField();
60      TextField tfPopulation = new TextField();
61      CheckBox chkDemocratic = new CheckBox("Demokracja?");
62      Button btAddRow = new Button("Dodaj wiersz");
63      tfCountry.setPrefColumnCount(5);
64      tfCapital.setPrefColumnCount(5);
65      tfPopulation.setPrefColumnCount(5);
66      flowPane.getChildren().addAll(new Label("Państwo: "),
67        tfCountry, new Label("Stolica"), tfCapital,
68        new Label("Ludność"), tfPopulation, chkDemocratic,
69        btAddRow);
70
71      btAddRow.setOnAction(e -> {
72        data.add(new Country(tfCountry.getText(), tfCapital.getText(),
73          Double.parseDouble(tfPopulation.getText()),
74          chkDemocratic.isSelected()));
75        tfCountry.clear();
76        tfCapital.clear();
77        tfPopulation.clear();
78      });
79
80      BorderPane pane = new BorderPane();
81      pane.setCenter(tableView);
82      pane.setBottom(flowPane);
83
84      Scene scene = new Scene(pane, 500, 250);
85      primaryStage.setTitle("AddNewRowDemo"); // Ustawianie nagłówka okna
86      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
87      primaryStage.show(); // Wyświetlanie okna
88    }
89
90    public static class Country {
91      private final SimpleStringProperty country;
92      private final SimpleStringProperty capital;
93      private final SimpleDoubleProperty population;
```

```
94     private final SimpleBooleanProperty democratic;
95
96     private Country(String country, String capital,
97         double population, boolean democratic) {
98       this.country = new SimpleStringProperty(country);
99       this.capital = new SimpleStringProperty(capital);
100      this.population = new SimpleDoubleProperty(population);
101      this.democratic = new SimpleBooleanProperty(democratic);
102    }
103
104    public String getCountry() {
105      return country.get();
106    }
107
108    public voidsetCountry(String country) {
109      this.country.set(country);
110    }
111
112    public String getCapital() {
113      return capital.get();
114    }
115
116    public void setCapital(String capital) {
117      this.capital.set(capital);
118    }
119
120    public double getPopulation() {
121      return population.get();
122    }
123
124    public void setPopulation(double population) {
125      this.population.set(population);
126    }
127
128    public boolean isDemocratic() {
129      return democratic.get();
130    }
131
132    public void setDemocratic(boolean democratic) {
133      this.democratic.set(democratic);
134    }
135  }
136 }
```

Ten program jest podobny do wersji z listingu 31.13, ale zawiera nowy kod umożliwiający dodawanie nowych danych (wiersze 57. – 82.). Użytkownik wprowadza dane w polach tekstowych i polu wyboru, a następnie za pomocą przycisku *Dodaj wiersz* dodaje nowy wiersz. Ponieważ dane są zapisane na liście obserwowalnej, zmiana w danych jest automatycznie odzwierciedlana w tabeli.

Na rysunku 31.34a w polach tekstowych wprowadzone są informacje o nowym państwie. Po kliknięciu przycisku *Dodaj wiersz* nowe państwo zostanie wyświetlone w tabeli.

Obiekt typu `TableView` nie tylko wyświetla dane, ale też umożliwia ich edycję. Aby umożliwić edycję danych w tabeli, wykonaj następujące kroki:



RYSUNEK 31.34. Zmiana w modelu danych jest automatycznie odzwierciedlana w tabeli

1. Ustaw właściwość `editable` obiektu typu `TableView` na `true`.
2. Jako fabrykę komórek kolumny zastosuj klasę `TextFieldTableCell`.
3. Zaimplementuj metodę `setOnEditCommit` dla kolumny, aby przypisywać zmodyfikowaną wartość do modelu danych.

Oto kod umożliwiający edycję wartości w kolumnie `countryColumn`:

```
tableView.setEditable(true);
countryColumn.setCellFactory(TextFieldTableCell.forTableColumn());
countryColumn.setOnEditCommit(
  new EventHandler<CellEditEvent<Country, String>>() {
    @Override
    public void handle(CellEditEvent<Country, String> t) {
      t.getTableView().getItems().get(
        t.getTablePosition().getRow())
      .setCountry(t.getNewValue());
    }
  }
);
```

31.10.1. Jak utworzyć obiekt typu `TableView`? Jak utworzyć kolumnę tabeli? Jak dodać kolumnę do obiektu typu `TableView`?

31.10.2. Jakiego typu jest model danych w obiektach typu `TableView`? Jak powiązać model danych z obiektem typu `TableView`?

31.10.3. Jak skonfigurować fabrykę wartości komórek dla obiektu typu `TableColumn`?

31.10.4. Jak dodać grafikę do nagłówka kolumny?

31.11. Pisanie programów dla architektury JavaFX za pomocą języka FXML

Interfejsy użytkownika w architekturze JavaFX można tworzyć za pomocą skryptów w języku FXML.

Istnieją dwa sposoby tworzenia aplikacji opartych na JavaFX. Pierwszy polega na pisaniu ich w całości w kodzie źródłowym w Javie. Drugi to używanie języka FXML. Jest to oparty na XML-u język skryptowy do opisywania interfejsu użytkownika. Używanie FXML-a pozwala oddzielić interfejs użytkownika od logiki napisanej w Javie. JavaFX Scene Builder to graficzny kreator do tworzenia interfejsów użytkownika bez ręcznego pisania skryptów

w FXML-u. Wystarczy przeciągnąć komponenty interfejsu użytkownika na panel i ustawić ich właściwości w oknie *Inspector*. Narzędzie Scene Builder automatycznie generuje skrypty interfejsu użytkownika w FXML-u. W tym podrozdziale zobaczysz, jak za pomocą narzędzia Scene Builder budować aplikacje oparte na JavaFX.

Uwaga

Ważne jest, aby najpierw nauczyć się pisać kod w JavaFX bez używania FXML-a. Pozwala to opanować podstawy architektury JavaFX przed nauką FXML-a. Gdy już będziesz znać podstawy JavaFX, łatwo przyjdzie Ci tworzyć programy w FXML-u. Dlatego FXML jest opisywany po omówieniu programowania w JavaFX.

31.11.1. Instalowanie narzędzia JavaFX Scene Builder

Narzędzie JavaFX Scene Builder może działać niezależnie lub w ramach środowiska IDE takiego jak NetBeans lub Eclipse. W tym podrozdziale zobaczysz, jak korzystać z tego narzędzia w środowisku NetBeans. Najnowszą wersję narzędzia Scene Builder możesz pobrać ze strony *http://gluonhq.com/open-source/scene-builder/*.

31.11.2. Tworzenie projektu JavaFX FXML

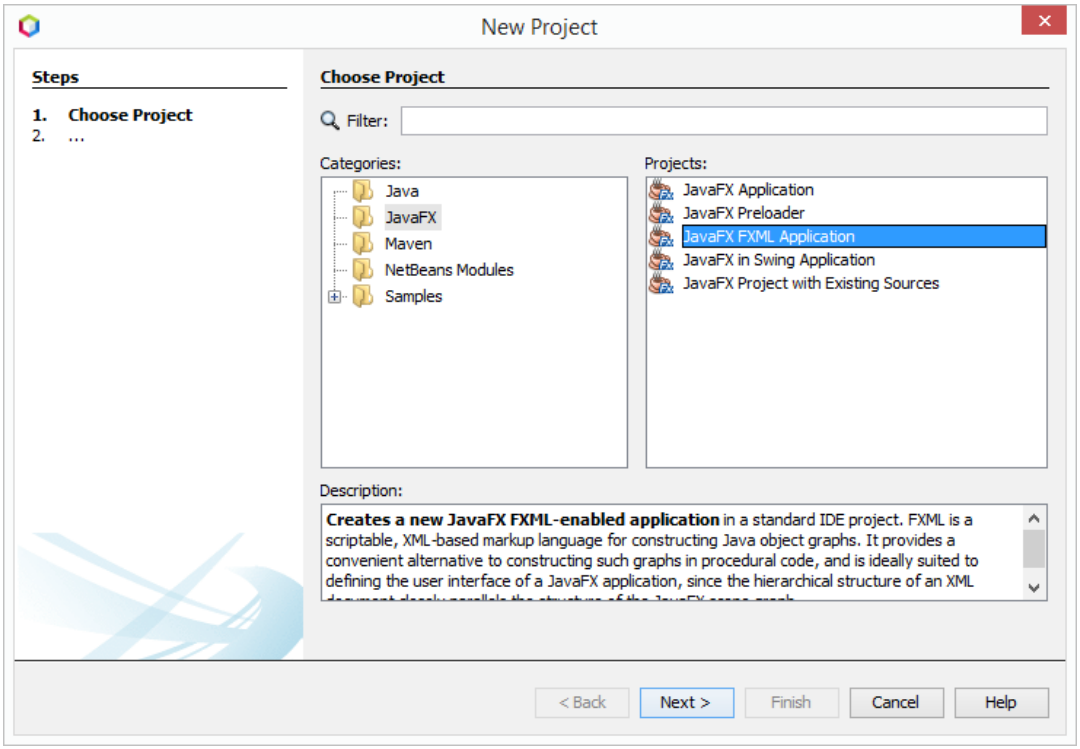
Aby użyć FXML-a w aplikacji opartej na JavaFX, musisz utworzyć projekt JavaFX FXML w środowisku NetBeans. Oto potrzebne kroki:

1. Wybierz opcję *File/New Project*, aby wyświetlić okno dialogowe *New Project* (rysunek 31.35).
2. Wybierz opcję *JavaFX* na liście *Categories* i *JavaFX FXML Application* na liście *Projects*. Kliknij przycisk *Next*, aby przejść do okna *New JavaFX Application* (rysunek 31.36).
3. Wpisz *Calculator* jako nazwę projektu i kliknij przycisk *Finish*, aby zakończyć proces. Utworzony projekt wygląda jak na rysunku 31.37.

W tym projekcie tworzone są trzy pliki: *Calculator.java*, *FXMLDocument.fxml* i *FXMLDocumentController.java*. Kod źródłowy z tych plików przedstawiają listingi 31.15, 31.16 i 31.17. W kontekście architektury MVC te trzy pliki reprezentują model, widok i kontroler. Model danych można zdefiniować w klasie *Calculator.java*. Plik *.fxml* opisuje interfejs użytkownika. Plik kontrolera definiuje sposób przetwarzania zdarzeń generowanych w interfejsie użytkownika.

LISTING 31.15. Calculator.java

```
 1 package calculator;
 2
 3 import javafx.application.Application;
 4 import javafx.fxml.FXMLLoader;
 5 import javafx.scene.Parent;
 6 import javafx.scene.Scene;
 7 import javafx.stage.Stage;
 8
 9 public class Calculator extends Application {
10   @Override
11   public void start(Stage stage) throws Exception {
12     Parent root =
13       FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
14     Scene scene = new Scene(root);
15     stage.setScene(scene);
16     stage.show();
17   }
```

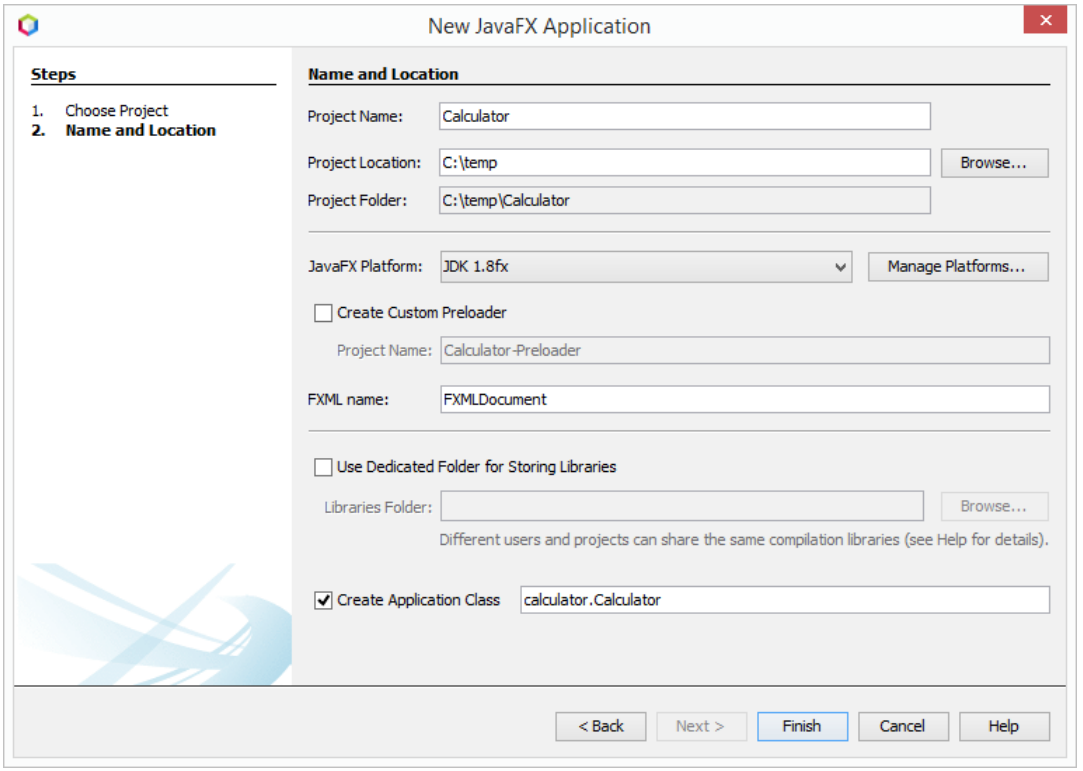


RYSUNEK 31.35. Aby utworzyć projekt FXML-owy, wybierz opcję JavaFX na liście Categories i opcję JavaFX FXML Application na liście Projects

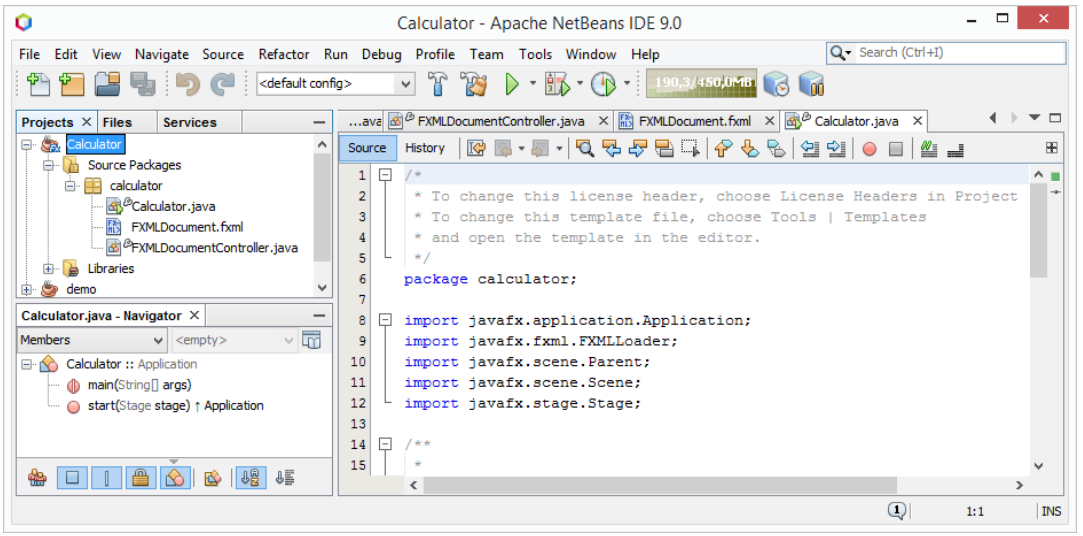
```
18
19    /**
20     * @param args the command line arguments
21     */
22    public static void main(String[] args) {
23      launch(args);
24    }
25 }
```

LISTING 31.16. FXMLDocument.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
  <children>
```



RYSUNEK 31.36. W oknie dialogowym New JavaFX Application możesz wpisać informacje o projekcie



RYSUNEK 31.37. Projekt FXML-owy został utworzony

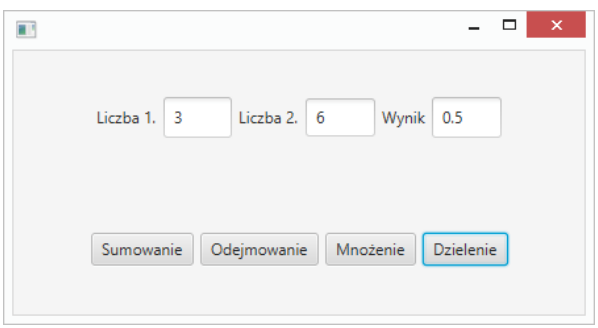
```
    <Button layoutX="126" layoutY="90" text="Click Me!"
            onAction="#handleButtonAction" fx:id="button" />
    <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
           fx:id="label" />
  </children>
</AnchorPane>
```

Listing 31.17. FXMLDocumentController.java

```
package calculator;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable {
  @FXML
  private Label label;

  @FXML
  private void handleButtonAction(ActionEvent event) {
    System.out.println("You clicked me!");
    label.setText("Hello World!");
  }

  @Override
  public void initialize(URL url, ResourceBundle rb) {
    // TODO
  }
}
```

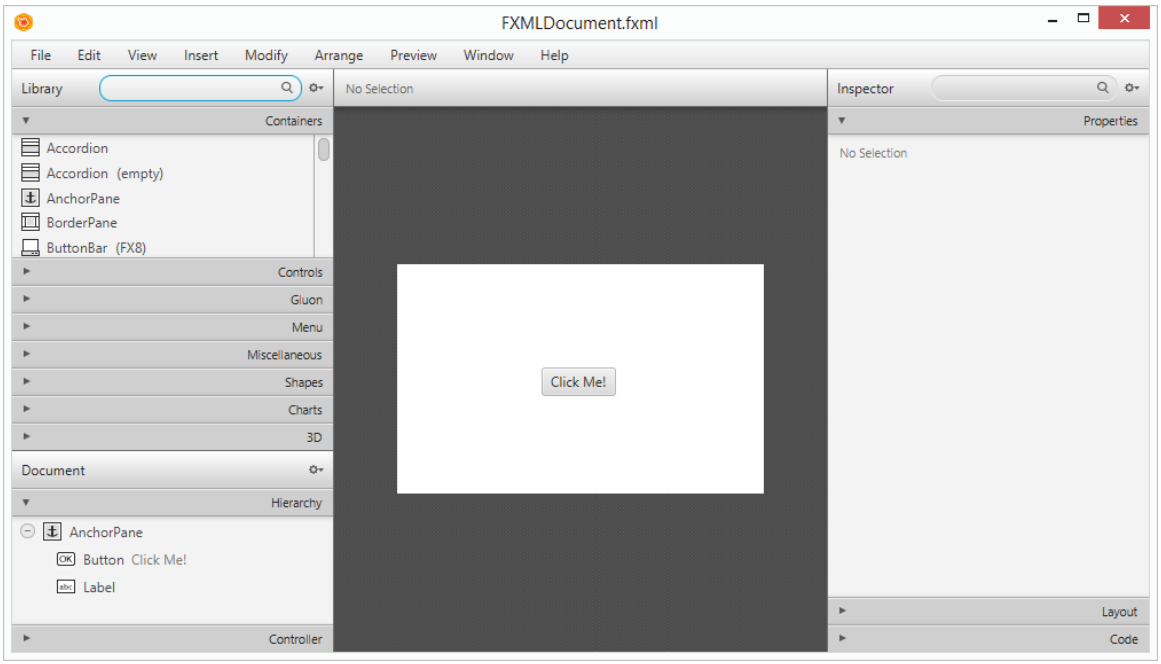
31.11.3. Tworzenie interfejsów użytkownika

Teraz zobaczysz, jak utworzyć prosty kalkulator taki jak na rysunku 31.38. Ten program ma umożliwiać wpisywanie liczb oraz ich dodawanie, odejmowanie, mnożenie i dzielenie.



Rysunek 31.38. Aplikacja wykonuje operacje arytmetyczne

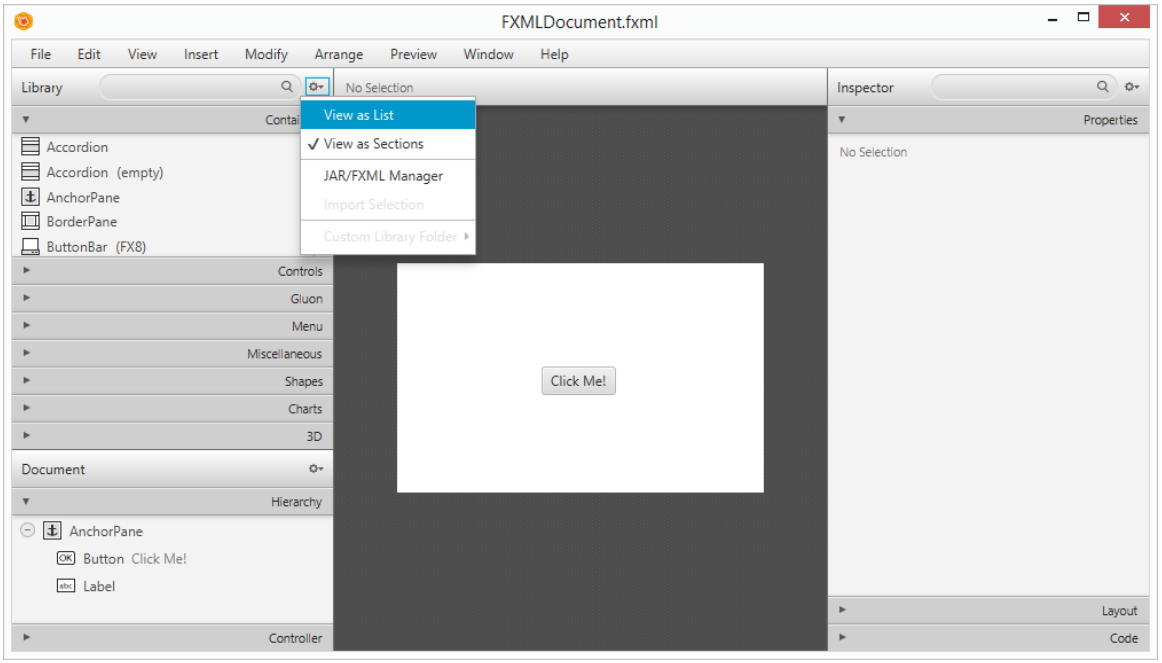
Gdy tworzysz projekt JavaFX FXML, NetBeans generuje domyślny plik *.fxml* z prostym przykładowym interfejsem użytkownika. Aby wyświetlić ten interfejs, kliknij dwukrotnie plik *.fxml*; otworzy się narzędzie Scene Builder (rysunek 31.39). Zauważ, że NetBeans potrafi automatycznie wykrywać zainstalowane w komputerze narzędzie Scene Builder.



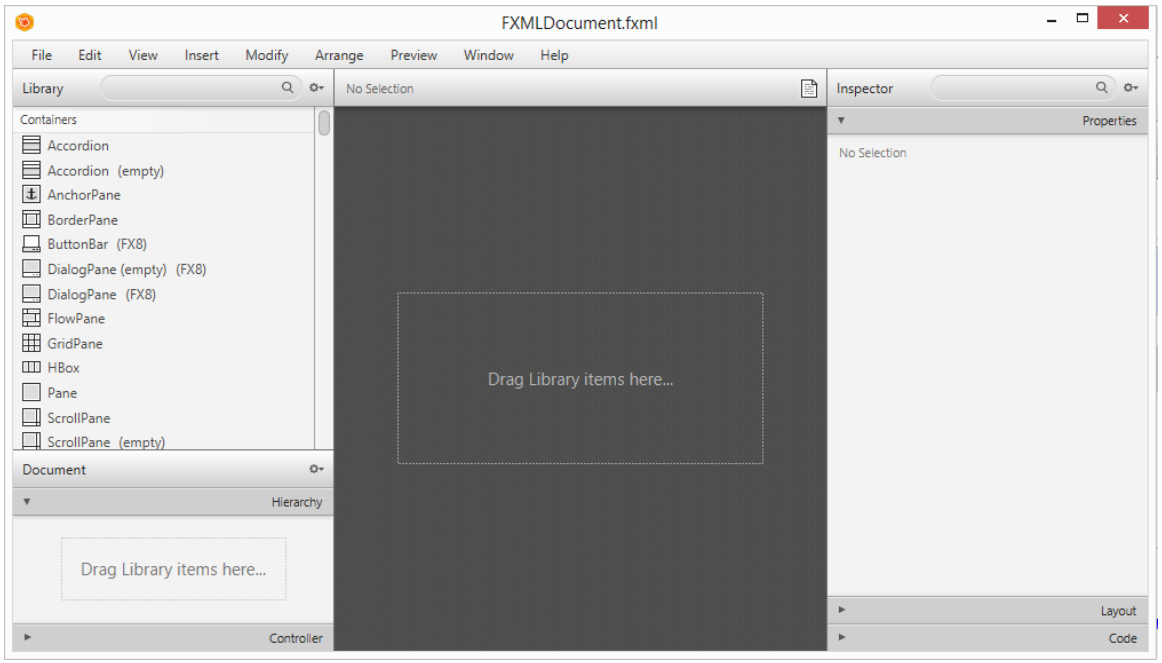
Rysunek 31.39. Kliknij dwukrotnie plik .fxml, aby otworzyć narzędzie Scene Builder

W celu rozpoczęcia tworzenia nowego interfejsu użytkownika usuń domyślny interfejs z pliku *.fxml* (rysunek 31.41). Oto proces tworzenia nowego interfejsu:

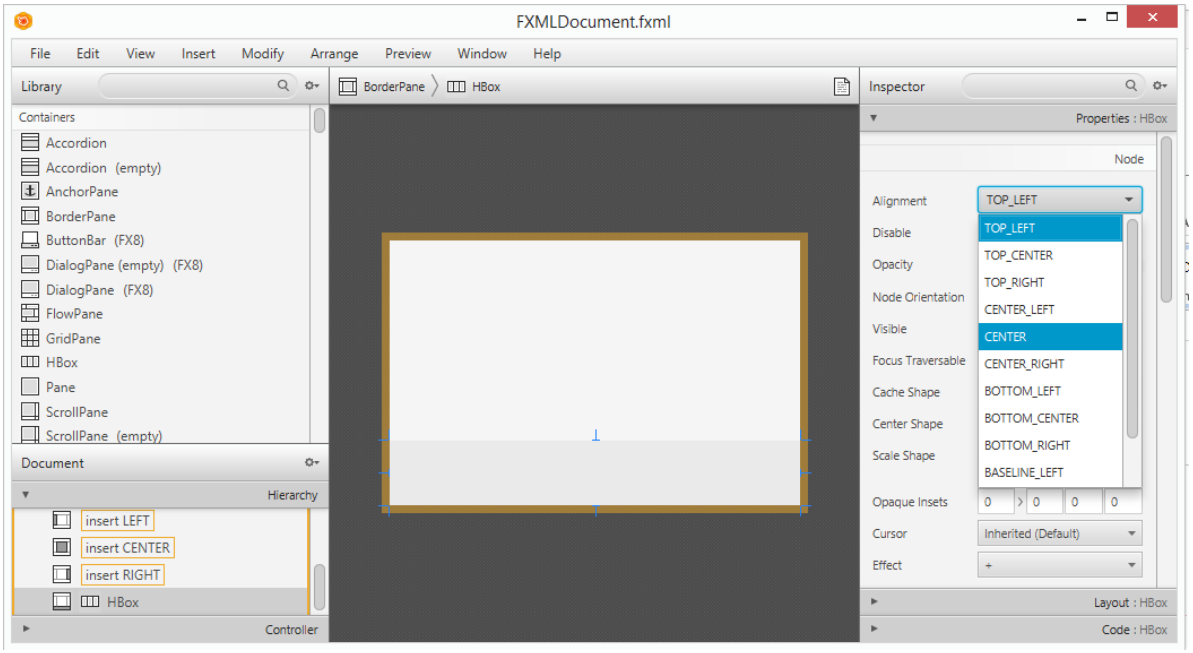
1. Krok opcjonalny: w niektórych systemach komponenty w panelu *Library* nie są podzielone na kategorie. Kliknij ikonę *Library*, aby otworzyć menu kontekstowe z rysunku 31.40, i wybierz opcję *View as List*.
2. Przeciągnij do interfejsu panel `BorderPane`, po czym umieść panele `HBox` pośrodku i w dolnej części panelu `BorderPane`. Ustaw wyrównanie obu paneli `HBox` na wartość `CENTER` (rysunek 31.42). Ustaw właściwość *Spacing* w sekcji *Layout* panelu *Inspector* na 5. Gdy zaznaczysz komponent w oknie, w panelu *Inspector* pojawią się właściwości tego komponentu. Można wtedy ustawić ich wartości.
3. Przeciągnij do okna kontrolki typów `Label`, `TextField`, `Label`, `TextField`, `Label` i `TextField`, po czym zmień tekst w kontrolkach `Label` na *Liczba 1.*, *Liczba 2.* i *Wynik* (rysunek 31.43). Ustaw właściwość *Pref Column Count* kontrolek `TextField` na wartość *2* w sekcji *Layout* panelu *Inspector*. W sekcji *Code* tego panelu ustaw identyfikatory kontrolek `TextField` na `tfNumber1`, `tfNumber2` i `tfResult` (rysunek 31.44). Te identyfikatory będą pomocne do wskazywania pól tekstowych i pobierania ich wartości w kontrolerze.
4. Przeciągnij cztery przyciski (kontrolki `Button`) do drugiego panelu `HBox`. Ustaw ich właściwość *Text* na *Sumowanie*, *Odejmowanie*, *Mnożenie* i *Dzielenie* (rysunek 31.45).



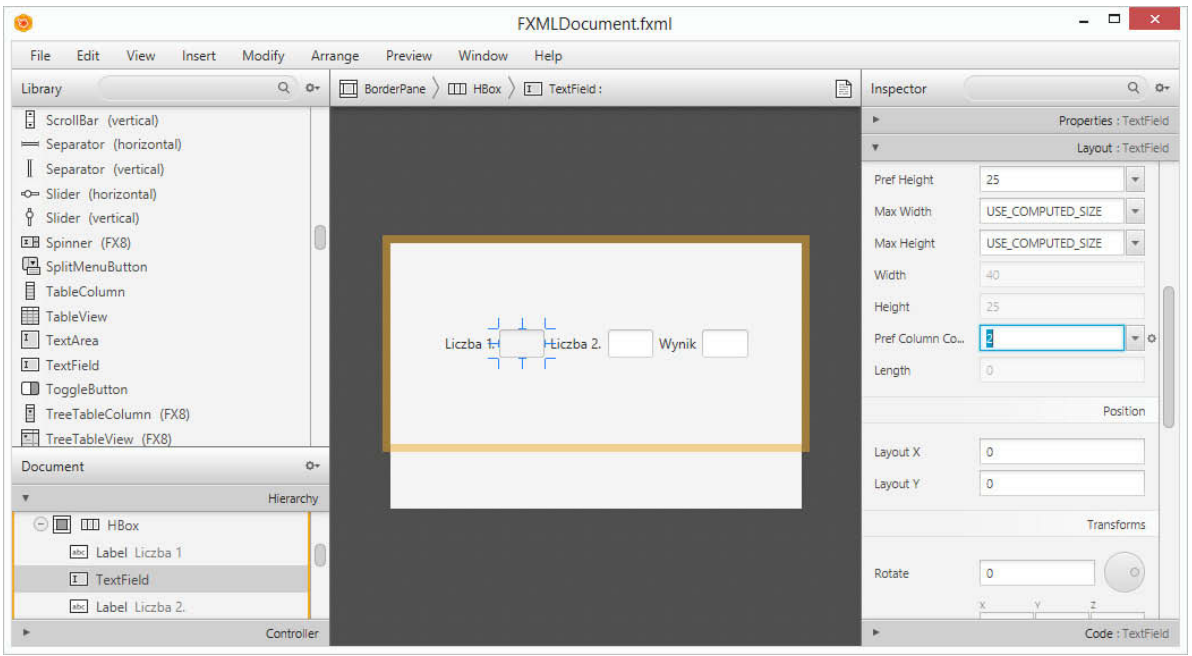
RYSUNEK 31.40. Aby otworzyć panel Library, kliknij ikonę Library i wybierz opcję View as List



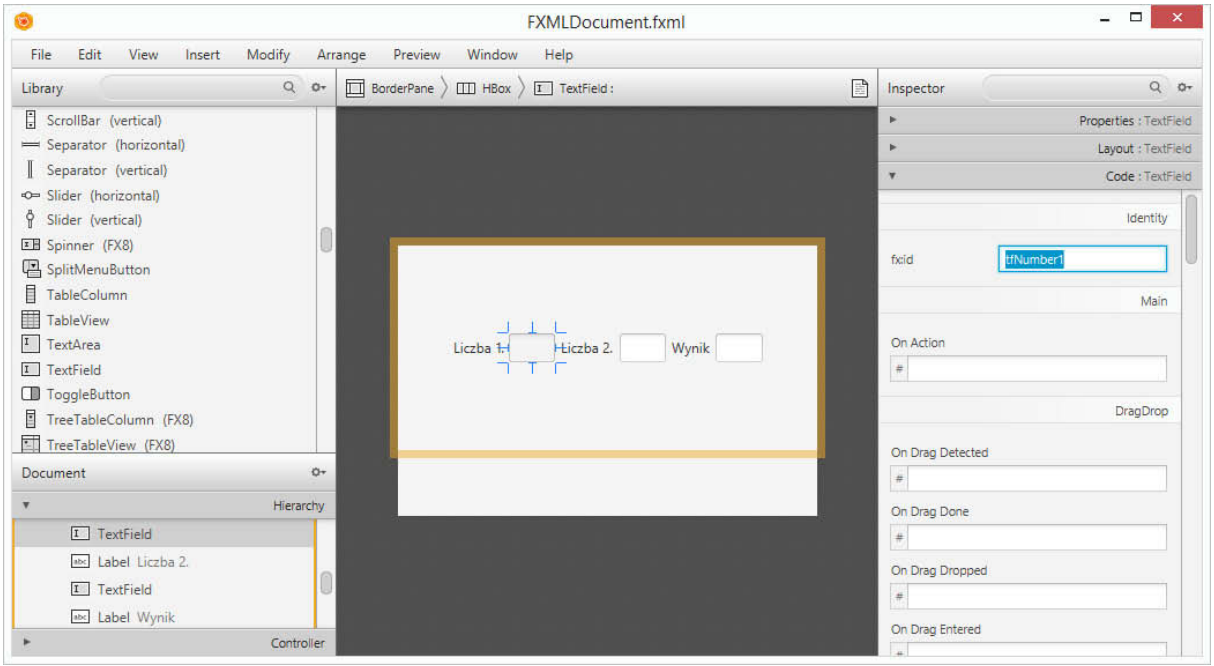
RYSUNEK 31.41. Po usunięciu domyślnego przycisku z panelu interfejs jest pusty



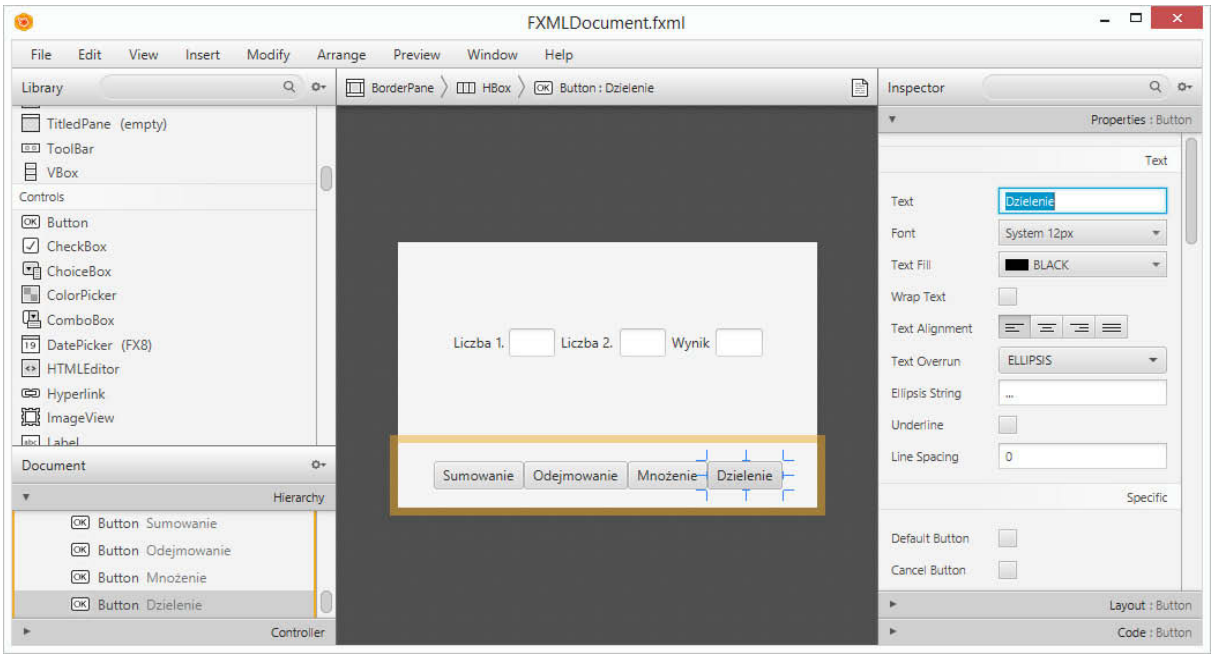
Rysunek 31.42. Do interfejsu przenoszony jest panel BorderPane, w którego dolnej części umieszczany jest panel HBox



Rysunek 31.43. Etykiety i pola tekstowe przeciągnięte na interfejs użytkownika

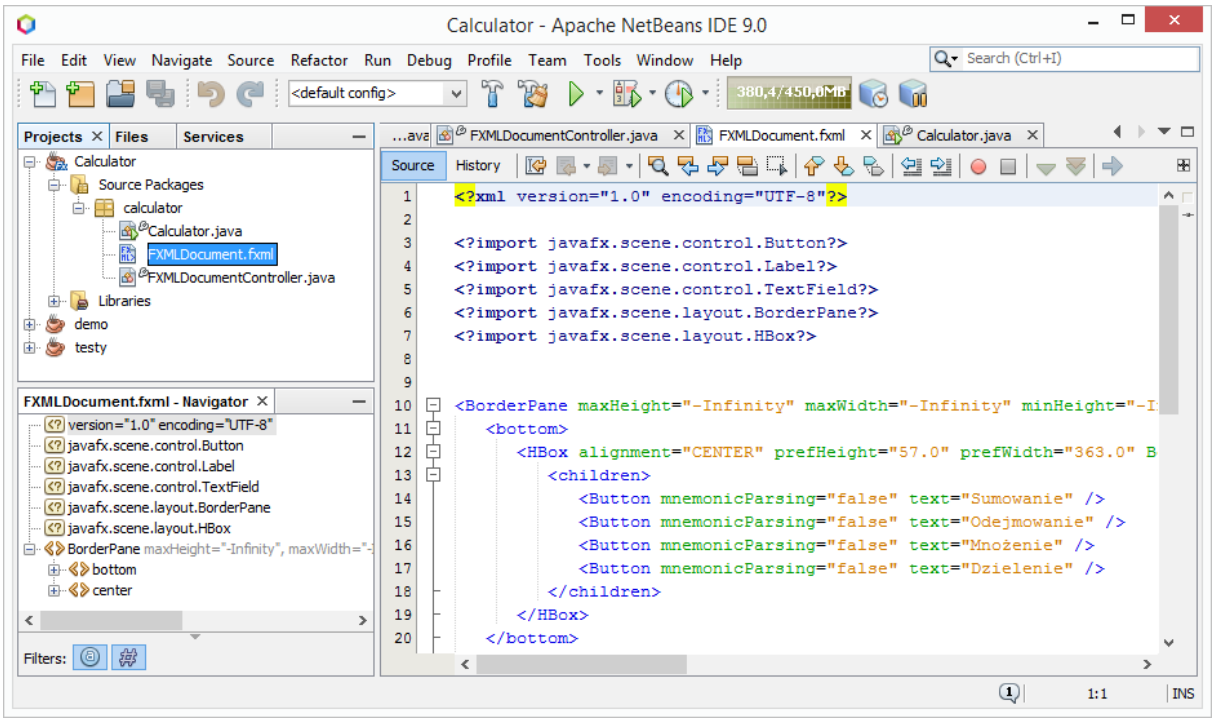


RYSUNEK 31.44. Ustaw odpowiednie identyfikatory pól tekstowych



RYSUNEK 31.45. Przyciski przeciągnięte do panelu HBox

Po utworzeniu i zmodyfikowaniu interfejsu użytkownika w oknie graficznym należy zapisać zmiany za pomocą opcji *File/Save* z paska menu narzędzia Scene Builder. Plik *.fxml* zostanie zaktualizowany i zsynchronizowany ze zmianami z okna graficznego. Zawartość pliku *.fxml* możesz podejrzeć w środowisku NetBeans (rysunek 31.46).



Rysunek 31.46. Możesz podejrzeć zawartość pliku FXML

31.11.4. Obsługa zdarzeń w kontrolerze

Plik *.fxml* opisuje interfejs użytkownika. Na listingu 31.18 pokazany jest kod logiki z pliku kontrolera.

Listing 31.18. FXMLDocumentController.java

```
 1 package calculator;
 2
 3 import javafx.event.ActionEvent;
 4 import javafx.fxml.FXML;
 5 import javafx.scene.control.TextField;
 6
 7 public class FXMLDocumentController {
 8   @FXML
 9   private TextField tfNumber1, tfNumber2, tfResult;
10
11   @FXML
12   private void addButtonAction(ActionEvent event) {
13     tfResult.setText(getResult('+') + "");
14   }
```

```
15
16   @FXML
17   private void subtractButtonAction(ActionEvent event) {
18     tfResult.setText(getResult('-') + "");
19   }
20
21   @FXML
22   private void multiplyButtonAction(ActionEvent event) {
23     tfResult.setText(getResult('*') + "");
24   }
25
26   @FXML
27   private void divideButtonAction(ActionEvent event) {
28     tfResult.setText(getResult('/') + "");
29   }
30
31   private double getResult(char op) {
32     double number1 = Double.parseDouble(tfNumber1.getText());
33     double number2 = Double.parseDouble(tfNumber2.getText());
34     switch (op) {
35       case '+': return number1 + number2;
36       case '-': return number1 - number2;
37       case '*': return number1 * number2;
38       case '/': return number1 / number2;
39     }
40     return Double.NaN;
41   }
42 }
```

W klasie kontrolera zadeklarowane są trzy pola tekstowe: `tfNumber1`, `tfNumber2` i `tfResult` (wiersz 9.). Adnotacja `@FXML` oznacza, że opatrzone nią pola są powiązane z polami tekstowymi z interfejsu użytkownika. Warto przypomnieć, że w interfejsie użytkownika identyfikatory trzech pól tekstowych to `tfNumber1`, `tfNumber2` i `tfResult`.

Kod do obsługi zdarzeń generowanych przez przyciski jest zdefiniowany w metodach `addButtonAction`, `subtractButtonAction`, `multiplyButtonAction` i `divideButtonAction` (wiersze 11. – 29.). Adnotacja `@FXML` oznacza, że metody są powiązane z operacjami przycisków z widoku.

Za pomocą adnotacji `@FXML` pola i metody w kontrolerze są wiązane z komponentami i operacjami zdefiniowanymi w pliku *.fxml*.

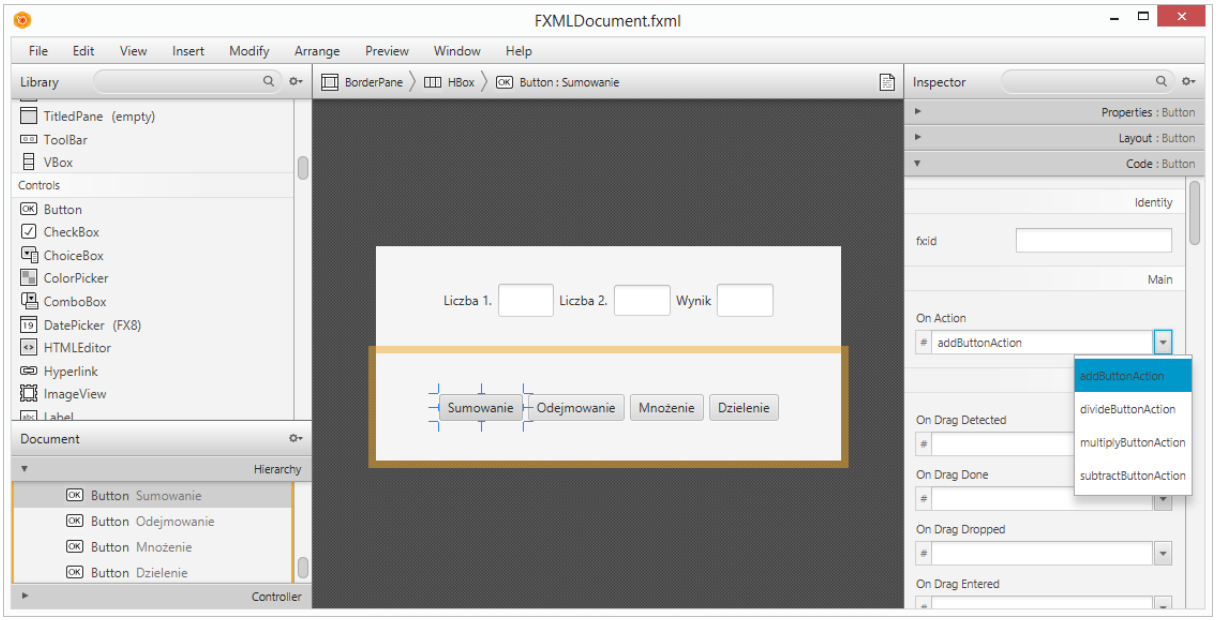
31.11.5. Wiązanie widoku z kontrolerem

Teraz możesz powiązać operacje komponentów z widoku z metodami z kontrolera służącymi do przetwarzania tych operacji. Oto potrzebne kroki:

1. Dodaj poniższy atrybut w znaczniku `<BorderPane>`, aby użyć kontrolera do widoku:

   ```
   fx:controller="calculator.FXMLDocumentController"
   ```

2. Kliknij dwukrotnie plik *.fxml* projektu, aby wyświetlić okno graficzne. Dla przycisku *Sumowanie* wybierz w panelu *Inspector* opcję *addButtonAction* z listy metod (rysunek 31.47). Kompletny kod pliku *.fxml* znajdziesz na listingu 31.19.



RYSUNEK 31.47. Wybierz opcję addButtonAction, aby wygenerować kod do obsługi kliknięcia przycisku Sumowanie

LISTING 31.19. FXMLDocument.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane maxHeight="200" maxWidth="600" minHeight="200"
            minWidth="600" prefHeight="400.0" prefWidth="600.0"
            xmlns="http://javafx.com/javafx/8"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
  <bottom>
    <HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0"
          spacing="5.0" BorderPane.alignment="CENTER">
      <children>
        <Button mnemonicParsing="false"
          onAction="#addButtonAction" text="Sumowanie" />
        <Button mnemonicParsing="false"
          onAction="#subtractButtonAction" text="Odejmowanie" />
        <Button mnemonicParsing="false"
          onAction="#multiplyButtonAction" text="Mnożenie" />
        <Button mnemonicParsing="false"
          onAction="#divideButtonAction" text="Dzielenie" />
      </children>
    </HBox>
```

```
  </bottom>
  <center>
    <HBox alignment="CENTER" prefHeight="232.0" prefWidth="572.0"
          spacing="5.0" BorderPane.alignment="CENTER">
      <children>
        <Label text="Liczba 1." />
        <TextField fx:id="tfNumber1" prefColumnCount="2"
          prefHeight="51.0" prefWidth="74.0" />
        <Label text="Liczba 2." />
        <TextField fx:id="tfNumber2" prefColumnCount="2"
          prefHeight="51.0" prefWidth="70.0" />
        <Label text="Wynik" />
        <TextField fx:id="tfResult" prefColumnCount="2" />
      </children>
    </HBox>
  </center>
</BorderPane>
```

31.11.6. Uruchamianie projektu

Kod tego modelu jest generowany automatycznie; znajdziesz go na listingu 31.15. Jest to główny program, który wczytuje kod w FXML-u, aby utworzyć interfejs użytkownika w obiekcie nadrzędnym (wiersze 12. i 13.). Następnie obiekt nadrzędny jest dodawany do sceny (wiersz 14.). W wierszu 15. scena jest przypisywana do okna, a w wierszu 16. program wyświetla okno.

PODSUMOWANIE ROZDZIAŁU

1. JavaFX udostępnia arkusze stylów podobne do stylów CSS. Za pomocą metody `getStylesheets` można wczytać arkusz stylów, a metody `setStyle`, `setStyleClass` i `setId` przypisują style JavaFX CSS do węzłów.
2. JavaFX udostępnia klasy `QuadCurve`, `CubicCurve` i `Path` do tworzenia zaawansowanych kształtów.
3. JavaFX umożliwia przekształcanie współrzędnych za pomocą przesuwania, rotacji i skalowania.
4. Możesz ustawić wzorzec przerywanej linii, sposób łączenia kresek, szerokość i typ pędzla.
5. Za pomocą klas `Menu`, `MenuItem`, `CheckMenuItem` i `RadioMenuItem` możesz tworzyć menu.
6. Za pomocą klasy `ContextMenu` możesz tworzyć menu kontekstowe.
7. Za pomocą klasy `SplitPane` możesz wyświetlać w poziomie i pionie wiele paneli, których wielkość może być zmieniana przez użytkownika.
8. Za pomocą panelu `TabPane` można wyświetlić wiele paneli z zakładkami do ich wyboru.
9. Za pomocą klas `TableView` i `TableColumn` możesz tworzyć i wyświetlać tabele.
10. Za pomocą języka FXML możesz tworzyć interfejsy użytkownika w architekturze JavaFX. FXML to oparty na XML-u język skryptowy do opisywania interfejsów użytkownika. Użycie FXML-a pozwala oddzielić interfejs użytkownika od napisanej w Javie logiki.
11. JavaFX Scene Builder to narzędzie graficzne do tworzenia interfejsów użytkownika bez ręcznego pisania skryptów w FXML-u.

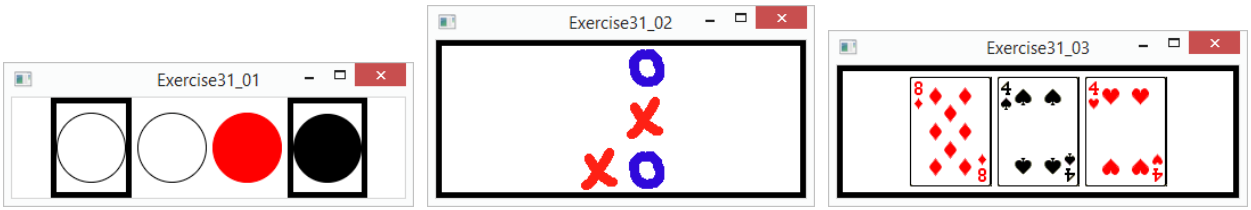
Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

ĆWICZENIA PROGRAMISTYCZNE

Podrozdział 31.2

31.1. *Używanie stylów JavaFX CSS.* Utwórz arkusz stylów CSS, który definiuje klasę z białym wypełnieniem i czarnym pędzlem oraz identyfikator z zielonym wypełnieniem i czerwonym pędzlem. Napisz program, który wyświetla cztery koła z wykorzystaniem klasy i interfejsu z arkusza. Przykładowy przebieg programu jest pokazany na rysunku 31.48a.



RYSUNEK 31.48. (a) Obramowanie i kolor kształtów są zdefiniowane w klasie stylu; (b) Rozwiązanie ćwiczenia 31.2 wyświetla planszę do gry w kółko i krzyżyk z obrazkami; obramowanie jest określone za pomocą stylów; (c) Program wybiera trzy losowe karty

***31.2.** *Plansza do gry w kółko i krzyżyk.* Napisz program, który wyświetla planszę do gry w kółko i krzyżyk (rysunek 31.48b). Komórka może zawierać X, O lub być pusta. Zawartość każdej komórki jest ustalana losowo. X i O to grafiki z plików *x.gif* i *o.gif*. Użyj arkusza stylów do zdefiniowania obramowania.

***31.3.** *Wyświetlanie trzech kart.* Napisz program, który wyświetla trzy losowe karty z talii 52 kart (rysunek 31.48c). Pliki z kartami noszą nazwy *1.png*, *2.png*, …, *52.png* i są zapisane w katalogu *image/card*. Wszystkie trzy karty mają być różne i program ma je wybierać losowo. Wskazówka: aby wybrać losowe karty, zapisz liczby od 1 do 52 w tablicy, przeprowadź losowe tasowanie (zobacz punkt 7.2.6) i wykorzystaj pierwsze 3 liczby z tablicy w nazwach plików. Obramowanie zdefiniuj w arkuszu stylów.

Podrozdział 31.3

31.4. *Kolor i czcionka.* Napisz program, który wyświetla w pionie pięć napisów (rysunek 31.49a). Dla każdego napisu ustaw losowy kolor i poziom przezroczystości. Tekst wyświetlaj czcionką Times Roman wielkości 22 pikseli z pogrubieniem i kursywą.

RYSUNEK 31.49. (a) Pięć napisów wyświetlanych określoną czcionką w losowym kolorze; (b) Ścieżka wyświetlana w kole; (c) Dwa koła wyświetlane w owalu

***31.5.** *Krzywa sześcienna.* Napisz program, który tworzy dwa kształty: koło i ścieżkę składającą się z dwóch krzywych sześciennych (rysunek 31.49b).

***31.6.** *Oczy.* Napisz program, który wyświetla parę oczu w owalu (rysunek 31.49c).

Podrozdział 31.4

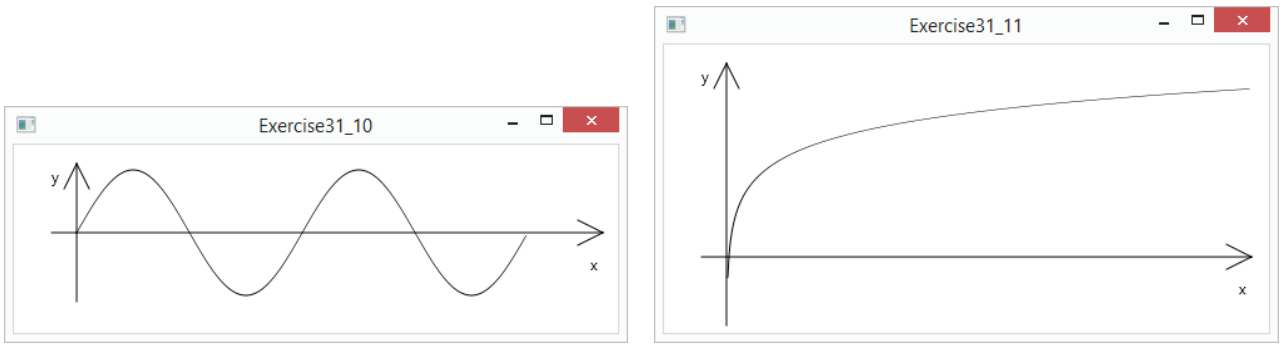
***31.7.** *Przesunięcie*. Napisz program, który wyświetla prostokąt z lewym górnym rogiem w punkcie (40, 40), o szerokości 50 i wysokości 40 jednostek. Wpisz wartości w polach tekstowych *x* i *y* oraz wciśnij przycisk *Przesuń*, aby przesunąć prostokąt w nową lokalizację (rysunek 31.50a).

Rysunek 31.50. (a) Rozwiązanie ćwiczenia 31.7 przesuwa figurę; (b) Rozwiązanie ćwiczenia 31.8 rotuje figurę; (c) Rozwiązanie ćwiczenia 31.9 skaluje figurę

***31.8.** *Rotacja*. Napisz program, który wyświetla wyśrodkowaną w panelu elipsę o szerokości 60 i wysokości 40 jednostek. Program ma przyjmować wartość z pola tekstowego *Kąt* i rotować elipsę po kliknięciu przycisku *Rotuj* (rysunek 31.50b).

***31.9.** *Skalowanie grafiki*. Napisz program, który wyświetla wyśrodkowaną w panelu elipsę o szerokości 60 i wysokości 40 jednostek. Program ma przyjmować poziom skalowania z pól tekstowych i skalować elipsę po kliknięciu przycisku *Skaluj* (rysunek 31.50c).

***31.10.** *Wykres funkcji sinus*. Napisz program, który wyświetla wykres funkcji sinus (rysunek 31.51a).



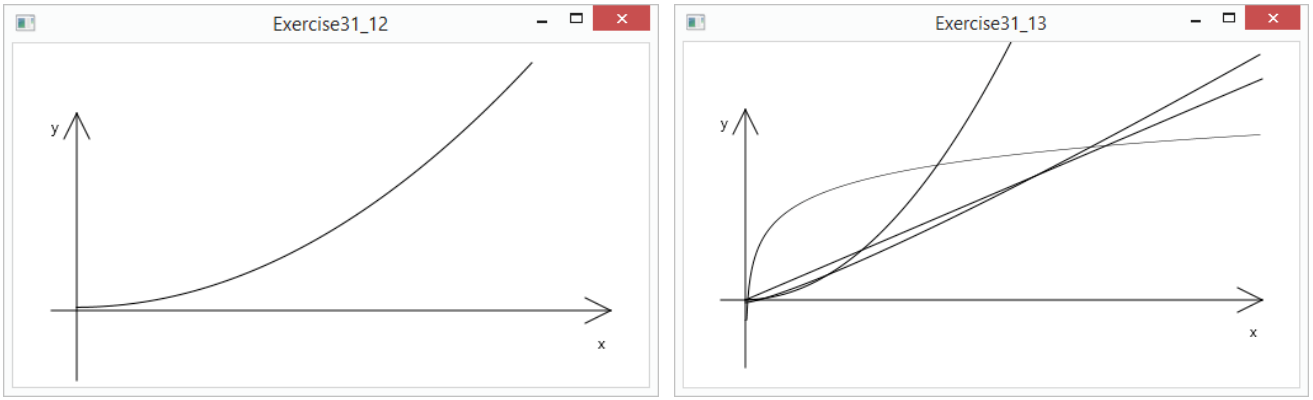
Rysunek 31.51. (a) Rozwiązanie ćwiczenia 31.10 wyświetla wykres funkcji sinus; (b) Rozwiązanie ćwiczenia 31.11 wyświetla wykres funkcji logarytmicznej

***31.11.** *Rysowanie wykresu funkcji logarytmicznej*. Napisz program do wyświetlania wykresu funkcji logarytmicznej (rysunek 31.51b).

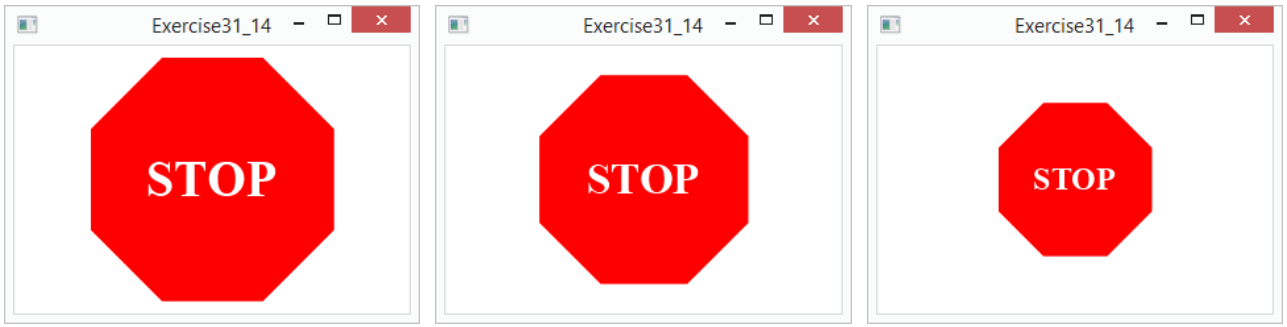
***31.12.** *Rysowanie wykresu funkcji n^2*. Napisz program do wyświetlania wykresu funkcji n^2 (rysunek 31.52a).

***31.13.** *Wykresy funkcji logarytmicznej, n, nlogn i n^2*. Napisz program wyświetlający wykresy funkcji logarytmicznej, n, nlogn i n^2 (rysunek 31.52b).

***31.14.** *Skalowanie i rotowanie grafiki*. Napisz program, który umożliwia użytkownikom skalowanie i rotowanie znaku STOP (rysunek 31.53). Użytkownik może wcisnąć klawisz strzałki w górę lub w dół, aby zwiększyć lub zmniejszyć wielkość grafiki; strzałki w prawo i lewo powodują rotację grafiki w prawo lub lewo.



RYSUNEK 31.52. (a) Rozwiązanie ćwiczenia 31.12 wyświetla wykres funkcji n^2; (b) Rozwiązanie ćwiczenia 31.13 wyświetla wykresy kilku funkcji



RYSUNEK 31.53. Program pozwala rotować i skalować grafikę

Podrozdział 31.5

***31.15.** *Słońce*. Napisz program, który wyświetla koło wypełnione gradientem (słońce) i promienie w postaci przerywanych linii (rysunek 31.54a).

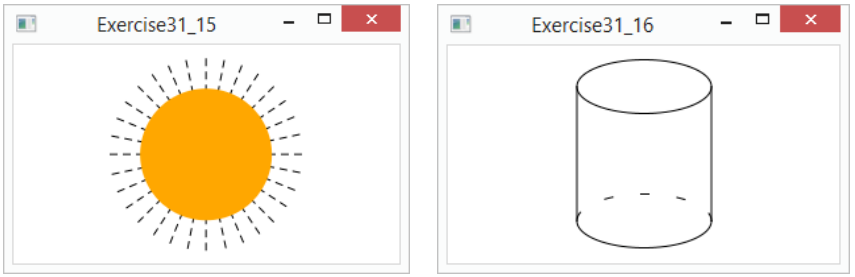
***31.16.** *Wyświetlanie walca*. Napisz program wyświetlający walec (rysunek 31.54b). Użyj przerywanej linii do narysowania ukrytego łuku.

Podrozdział 31.6

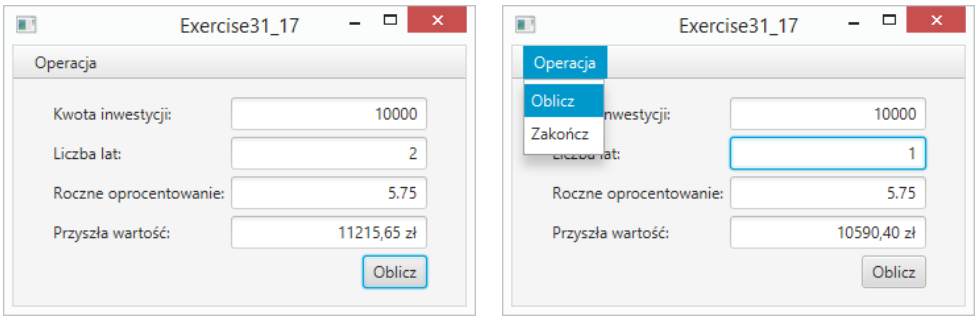
***31.17.** *Tworzenie kalkulatora wartości inwestycji*. Napisz program, który oblicza przyszłą wartość inwestycji przy określonym oprocentowaniu po podanej liczbie lat. Oto wzór:

$$\texttt{przyszłaWartość} = \texttt{kwotaInwestycji} * (1 + \texttt{miesięczneOprocentowanie})^{\texttt{lata*12}}$$

Użyj pól tekstowych do pobierania stopy oprocentowania, kwoty inwestycji i liczby lat. Gdy użytkownik kliknie przycisk *Oblicz* lub wybierze opcję *Oblicz* z menu *Operacja*, program ma wyświetlić przyszłą wartość inwestycji (rysunek 31.55). Opcja *Zakończ* ma kończyć pracę programu.



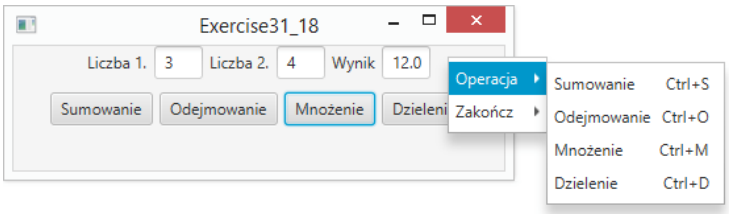
Rysunek 31.54. (a) Rozwiązanie ćwiczenia 31.15 wyświetla słońce; (b) Rozwiązanie ćwiczenia 31.16 wyświetla walec



Rysunek 31.55. Użytkownik wprowadza kwotę inwestycji, liczbę lat i stopę oprocentowania, aby obliczyć przyszłą wartość inwestycji

Podrozdział 31.8

***31.18.** *Menu kontekstowe.* Zmodyfikuj listing 31.9, *MenuDemo.java*, aby utworzyć menu kontekstowe zawierające podmenu *Operacja* i *Zakończ* (rysunek 31.56). To menu ma być wyświetlane po kliknięciu prawym przyciskiem myszy panelu zawierającego etykiety i pola tekstowe.

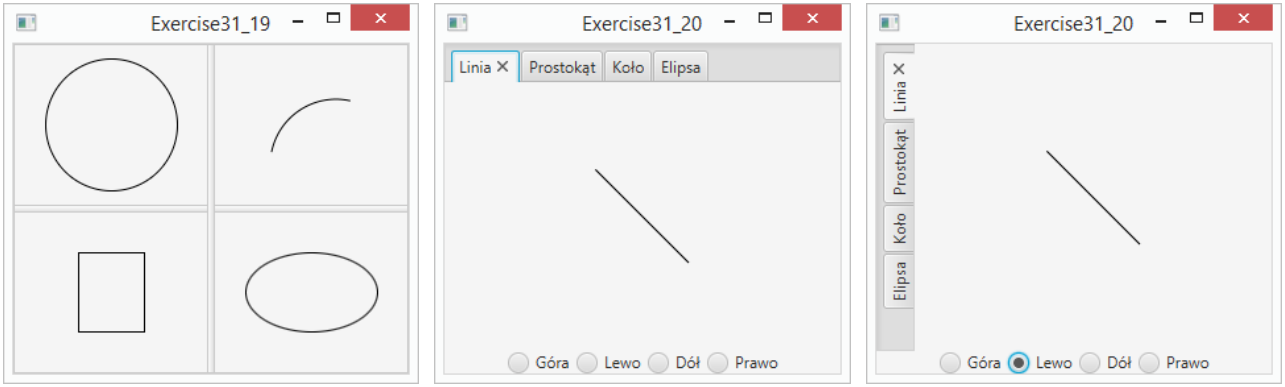


Rysunek 31.56. Menu kontekstowe zawiera polecenia do wykonywania operacji matematycznych

***31.19.** *Używanie panelu `SplitPane`.* Utwórz program wyświetlający cztery figury w panelach `SplitPane` (rysunek 31.57a).

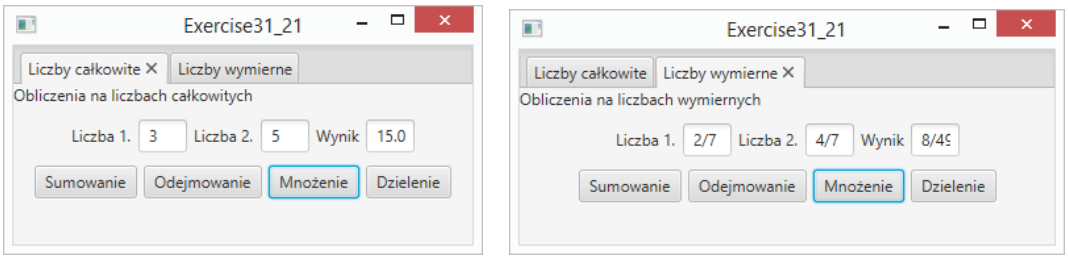
Podrozdział 31.9

***31.20.** *Używanie paneli `TabPane`.* Zmodyfikuj listing 31.12, *TabPaneDemo.java*, dodając panel z przyciskami opcji do określania lokalizacji zakładek (rysunki 31.57b i c).



RYSUNEK 31.57. (a) Cztery figury wyświetlane w panelach SplitPane; (b i c) Przyciski opcji umożliwiają wybór lokalizacji zakładek

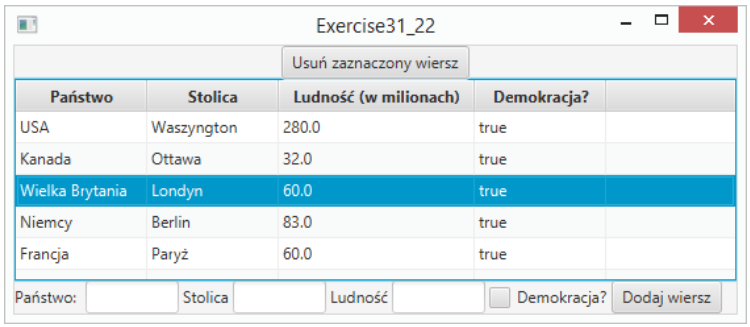
***31.21.** *Używanie paneli `TabPane`*. Użyj paneli `TabPane` do napisania programu wykonującego obliczenia na liczbach całkowitych i rzeczywistych (rysunek 31.58).



RYSUNEK 31.58. Panel TabPane służy do wybierania paneli do wykonywania operacji na liczbach całkowitych i rzeczywistych

Podrozdział 31.10

***31.22.** *Używanie kontrolki `TableView`*. Zmodyfikuj listing 31.14, dodając przycisk, który umożliwia usunięcie z tabeli wybranego wiersza (rysunek 31.59).



RYSUNEK 31.59. Kliknięcie przycisku „Usuń zaznaczony wiersz” usuwa wybrany wiersz z tabeli

ROZDZIAŁ

32

WIELOWĄTKOWOŚĆ I PROGRAMOWANIE RÓWNOLEGŁE

Cele

- Omówienie wielowątkowości (podrozdział 32.2).
- Tworzenie klas zadań z implementacją interfejsu `Runnable` (podrozdział 32.3).
- Używanie klasy `Thread` do tworzenia wątków do uruchamiania zadań (podrozdział 32.3).
- Kontrolowanie wątków za pomocą metod z klasy `Thread` (podrozdział 32.4).
- Sterowanie animacją za pomocą wątków i używanie wywołania `Platform.runLater` do uruchamiania kodu w wątku aplikacji (podrozdział 32.5).
- Wykonywanie zadań w puli wątków (podrozdział 32.6).
- Używanie zsynchronizowanych metod i bloków w celu zsynchronizowania wątków i uniknięcia sytuacji wyścigu (podrozdział 32.7).
- Synchronizowanie wątków z wykorzystaniem blokad (podrozdział 32.8).
- Wspomaganie komunikacji między wątkami z zastosowaniem warunków dla blokad (podrozdziały 32.9 i 32.10).
- Używanie kolejek z blokowaniem (`ArrayBlockingQueue`, `LinkedBlockingQueue` i `PriorityBlockingQueue`) do synchronizowania dostępu do nich (podrozdział 32.11).
- Ograniczanie jednoczesnego dostępu do współużytkowanych zasobów za pomocą semaforów (podrozdział 32.12).
- Używanie techniki cyklicznego oczekiwania na zasoby w celu uniknięcia zakleszczenia (podrozdział 32.13).
- Omówienie cyklu życia wątku (podrozdział 32.14).

- Tworzenie zsynchronizowanych kolekcji za pomocą metod statycznych z klasy `Collections` (podrozdział 32.15).
- Pisanie programów równoległych z wykorzystaniem platformy Fork/Join (podrozdział 32.16).

32.1. Wprowadzenie

Wielowątkowość umożliwia jednoczesne wykonywanie wielu zadań w programie.

wielowątkowość

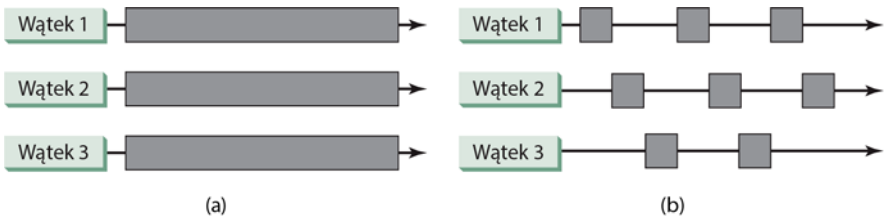
Jedną z wartościowych cech Javy jest wbudowana obsługa *wielowątkowości* — równoległego wykonywania wielu zadań w programie. W licznych językach programowania wielowątkowość wymaga wywoływania zależnych od systemu procedur i funkcji. W tym rozdziale poznasz wątki i dowiesz się, jak pisać programy wielowątkowe w Javie.

32.2. Zagadnienia związane z wątkami

Program może obejmować wiele równolegle wykonywanych zadań. Wątek wykonuje zadanie od początku do końca.

wątek
zadanie

Wątek to mechanizm do wykonywania zadań. W Javie można jednocześnie uruchomić wiele wątków programu. W systemach wieloprocesorowych te wątki można wykonywać jednocześnie (rysunek 32.1a).



RYSUNEK 32.1. (a) Wiele wątków działających w wielu procesorach; (b) Wiele wątków korzystających z jednego procesora

podział czasu

W systemach jednoprocesorowych (rysunek 32.1b) wiele wątków dzieli czas procesora. Jest to model *z podziałem czasu*, a za planowanie wykonywania wątków i przydzielanie im zasobów odpowiada system operacyjny. Ten model jest praktyczny, ponieważ przez większość czasu procesor pozostaje bezczynny (na przykład nie robi nic w trakcie oczekiwania na wprowadzenie danych).

Wielowątkowość może sprawić, że programy będą szybciej reagować i staną się bardziej interaktywne oraz wydajniejsze. Na przykład dobry edytor tekstu umożliwia drukowanie lub zapisywanie pliku w trakcie pisania. W niektórych scenariuszach programy wielowątkowe działają szybciej od jednowątkowych nawet w systemach jednoprocesorowych. Java zapewnia wyjątkowo dobre wsparcie tworzenia i uruchamiania wątków oraz blokowania zasobów w celu uniknięcia konfliktów.

zadanie
obiekt Runnable
wątek

Możesz tworzyć dodatkowe wątki, aby uruchamiać równolegle zadania w programie. W Javie każde zadanie to instancja interfejsu `Runnable` (nazywana *obiektem* `Runnable`). *Wątek* to obiekt wspomagający wykonywanie zadania.

32.2.1. Dlaczego wielowątkowość jest potrzebna? Jak można jednocześnie wykonywać wiele wątków w systemie jednoprocesorowym?

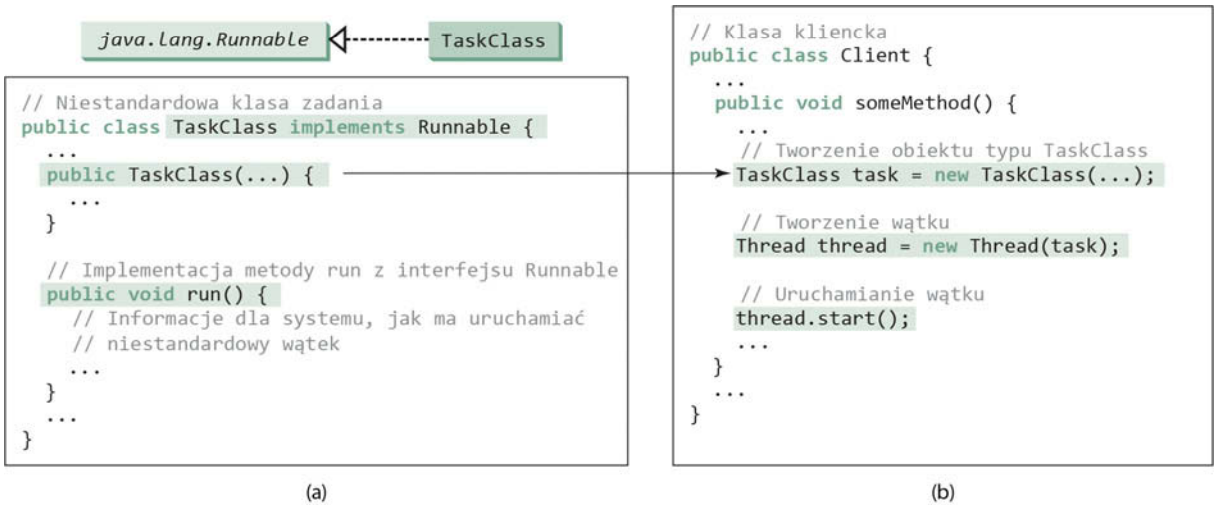
32.2.2. Czym jest obiekt `Runnable`? Czym jest wątek?

32.3. Tworzenie zadań i wątków

Klasa zadania musi implementować interfejs `Runnable`. Zadanie trzeba uruchamiać w wątku.

interfejs Runnable
metoda run()

Zadania są obiektami. Aby utworzyć zadanie, trzeba najpierw zdefiniować klasę zadania i zaimplementować w niej interfejs `Runnable`. Jest to prosty interfejs zawierający tylko metodę `run()`. Należy ją zaimplementować, aby poinformować system, jak wątek ma być wykonywany. Szablon do tworzenia klas zadań jest pokazany na rysunku 32.2a.



Rysunek 32.2a. Zdefiniuj klasę zadania i zaimplementuj interfejs Runnable

Po zdefiniowaniu klasy `TaskClass` możesz utworzyć zadanie za pomocą jej konstruktora:

```
TaskClass task = new TaskClass(...);
```

klasa Thread
tworzenie wątku

Zadanie trzeba wykonywać w wątku. Klasa `Thread` zawiera konstruktory do tworzenia wątków i wiele przydatnych metod do sterowania nimi. Wątek dla zadania możesz utworzyć tak:

```
Thread thread = new Thread(task);
```

Następnie możesz wywołać metodę `start()`, aby poinformować maszynę JVM, że wątek jest gotowy do uruchomienia:

```
thread.start();
```

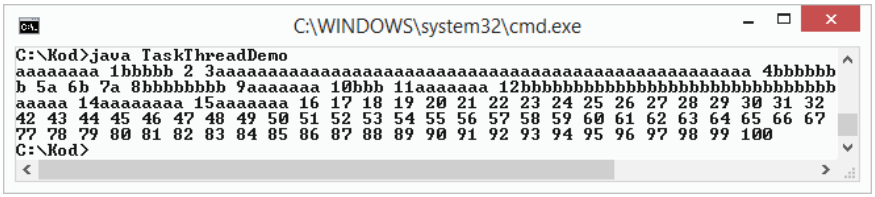
uruchamianie wątku

Maszyna JVM wykona to zadanie, wywołując jego metodę `run()`. Na rysunku 32.2b pokazane są główne kroki: tworzenie zadania, tworzenie wątku i uruchamianie wątku.

Na listingu 32.1 pokazany jest program, który tworzy trzy zadania i trzy wątki do ich wykonywania:

- pierwsze zadanie wyświetla 100 razy literę *a*,
- drugie zadanie wyświetla 100 razy literę *b*,
- trzecie zadanie wyświetla liczby całkowite od 1 do 100.

Gdy uruchomisz ten program, trzy wątki będą współdzielić procesor i na zmianę wyświetlać w konsoli litery i liczby. Przykładowy przebieg programu jest pokazany na rysunku 32.3.



Rysunek 32.3. Zadania printA, printB i print100 są wykonywane na zmianę i wyświetlają: 100 razy literę a, 100 razy literę b i liczby od 1 do 100

Listing 32.1. TaskThreadDemo.java

```
 1 public class TaskThreadDemo {
 2   public static void main(String[] args) {
 3     // Tworzenie zadań
 4     Runnable printA = new PrintChar('a', 100);
 5     Runnable printB = new PrintChar('b', 100);
 6     Runnable print100 = new PrintNum(100);
 7
 8     // Tworzenie wątków
 9     Thread thread1 = new Thread(printA);
10     Thread thread2 = new Thread(printB);
11     Thread thread3 = new Thread(print100);
12
13     // Uruchamianie wątków
14     thread1.start();
15     thread2.start();
16     thread3.start();
17   }
18 }
19
20 // Zadanie wyświetlające znak określoną liczbę razy
21 class PrintChar implements Runnable {
22   private char charToPrint; // Wyświetlany znak
23   private int times; // Liczba wyświetleń
24
25   /** Tworzy zadanie z określonym znakiem i liczbą
26     * wyświetleń tego znaku
27     */
28   public PrintChar(char c, int t) {
29     charToPrint = c;
30     times = t;
31   }
32
33   @Override /** Przesłanianie metody run() informującej
34                * system o tym, jakie zadanie należy wykonać
35                */
36   public void run() {
37     for (int i = 0; i < times; i++) {
38       System.out.print(charToPrint);
39     }
40   }
41 }
42
```

tworzenie zadań

tworzenie wątków

uruchamianie wątków

klasa zadania

metoda run

klasa zadania

```
43 // Klasa zadania wyświetlająca liczby od 1 do n dla danego n
44 class PrintNum implements Runnable {
45   private int lastNum;
46
47   /** Tworzy zadanie wyświetlające liczby 1, 2, ..., n */
48   public PrintNum(int n) {
49     lastNum = n;
50   }
51
52   @Override /** Informuje wątek, jak uruchamiać zadanie */
53   public void run() {
54     for (int i = 1; i <= lastNum; i++) {
55       System.out.print(" " + i);
56     }
57   }
58 }
```

metoda run

Ten program tworzy trzy zadania (wiersze 4. – 6.). Na potrzeby równoległego ich wykonywania tworzone są trzy wątki (wiersze 9. – 11.). Metoda `start()` (wiersze 14. – 16.) uruchamia wątek, co powoduje wykonanie metody `run()` zadania. Po jej wykonaniu wątek kończy pracę.

Ponieważ dwa pierwsze zadania, `printA` i `printB`, działają podobnie, można je zdefiniować w tej samej klasie `PrintChar` (wiersze 21. – 41.). Klasa `PrintChar` implementuje interfejs `Runnable` i przesłania metodę `run()` (wiersze 36. – 40.), wyświetlając znaki. Ta klasa służy do wyświetlania dowolnego znaku określoną liczbę razy. Obiekty `Runnable` `printA` i `printB` są instancjami klasy `PrintChar`.

Klasa `PrintNum` (wiersze 44. – 58.) implementuje interfejs `Runnable` i przesłania metodę `run()` (wiersze 53. – 57.), wyświetlając liczby. Ta klasa służy do wyświetlania liczb od 1 do *n* dla dowolnej liczby całkowitej *n*. Obiekt `Runnable` `print100` to instancja klasy `PrintNum`.

Uwaga

efekty współbieżności

Jeśli nie widzisz efektów jednoczesnego wykonywania trzech wątków, zwiększ liczbę wyświetleń znaków. Na przykład zmień wiersz 4. w następujący sposób:

```
Runnable printA = new PrintChar('a', 10000);
```

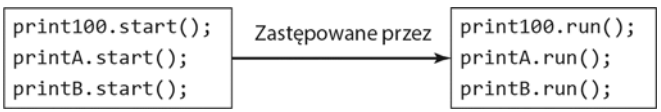
Ważna uwaga

metoda run()

Metoda `run()` w zadaniu określa, jak je wykonywać. Ta metoda jest automatycznie wywoływana przez maszynę JVM. Nie powinieneś uruchamiać jej samodzielnie. Bezpośrednie wywołanie metody `run()` powoduje wykonanie jej w tym samym wątku — nie są wtedy uruchamiane nowe wątki.

32.3.1. Jak zdefiniować klasę zadania? Jak utworzyć wątek dla zadania?

32.3.2. Co się stanie, jeśli w wierszach 14. – 16. listingu 32.1 zastąpisz metodę `start()` metodą `run()`?



32.3.3. Jakie błędy znajdują się w dwóch poniższych programach? Popraw usterki.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Test task = new Test();
new Thread(task).start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(a)

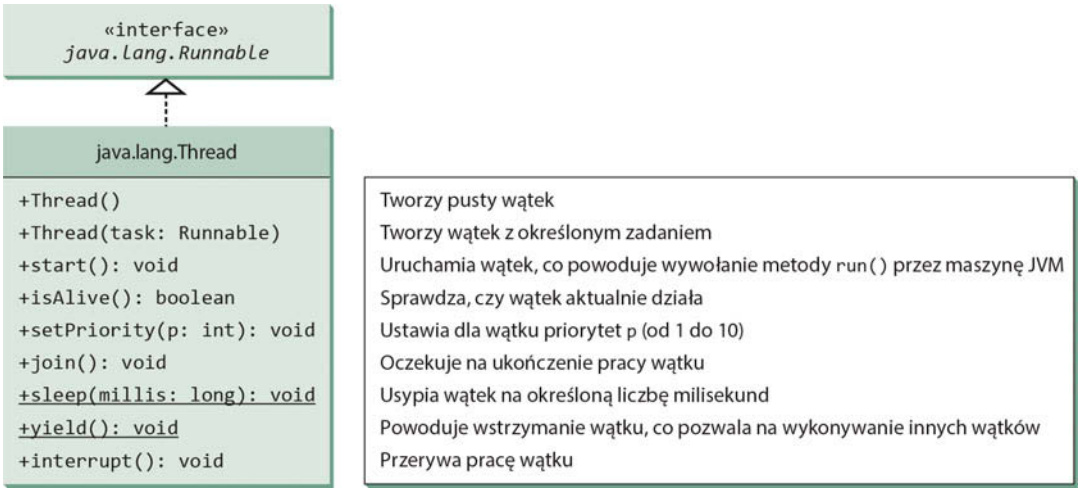
```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Thread t = new Thread(this);
    t.start();
    t.start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(b)

32.4. Klasa Thread

Klasa Thread zawiera konstruktory do tworzenia wątków z zadaniami i metody do sterowania wątkami.

Na rysunku 32.4 pokazany jest diagram klasy `Thread`.

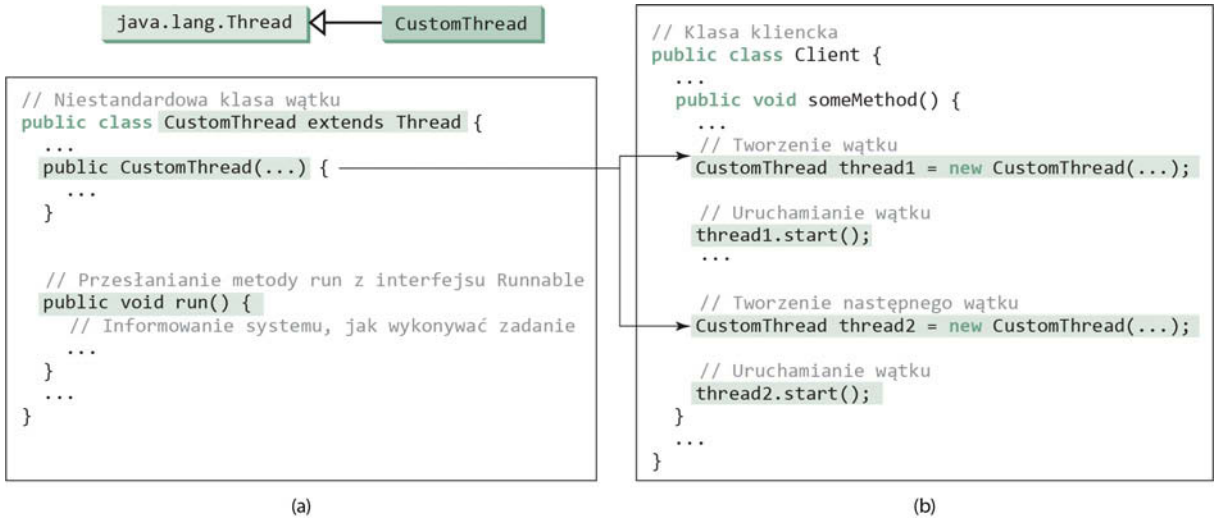


RYSUNEK 32.4. Klasa Thread zawiera metody do sterowania wątkami

Uwaga

oddzielanie zadania od wątku

Ponieważ klasa `Thread` implementuje interfejs `Runnable`, możesz zdefiniować klasę rozszerzającą klasę `Thread` i zaimplementować metodę `run` (rysunek 32.5a), a następnie utworzyć obiekt nowej klasy i wywołać w programie klienckim jego metodę `start`, aby uruchomić wątek (rysunek 32.5b).



RYSUNEK 32.5. Definiowanie klasy wątku przez rozszerzanie klasy Thread

Jednak to podejście nie jest zalecane, ponieważ łączy zadanie i mechanizm uruchamiania wątku. Preferowanym rozwiązaniem jest oddzielanie zadania od wątku.

Uwaga

niezalecane metody

Klasa `Thread` zawiera też metody `stop()`, `suspend()` i `resume()`. Od Javy 2 są one uznawane za *niezalecane* (lub *przestarzałe*), ponieważ wiadomo, że są z natury niebezpieczne. Zamiast używać metody `stop()`, należy przypisać `null` do zmiennej typu `Thread`, co oznacza zatrzymanie pracy wątku.

yield()

Metoda `yield()` pozwala tymczasowo zwolnić zasoby i przeznaczyć je dla innych wątków. Przyjmij, że w wierszach 53. – 57. listingu 32.1 zmodyfikowałeś kod metody `run()` z klasy `PrintNum`:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Teraz po każdym wyświetleniu liczby wątek z zadaniem `print100` przekazuje zasoby innym wątkom.

sleep(long)

Metoda `sleep(long millis)` usypia wątek na określoną liczbę milisekund, co pozwala na wykonywanie innych wątków. Przyjmij, że zmodyfikowałeś kod w wierszach 53. – 57. listingu 32.1 tak:

```
public void run() {
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " s+ i);
      if (i >= 50) Thread.sleep(1);
    }
  }
  catch (InterruptedException ex) {
  }
}
```

Po wyświetleniu każdej liczby >= 50 wątek z zadaniem print100 jest usypiany na milisekundę.

Metoda sleep może spowodować kontrolowany wyjątek InterruptedException. Występuje on po wywołaniu metody interrupt() dla uśpionego wątku. Metoda interrupt() jest wywoływana bardzo rzadko, dlatego wspomniany wyjątek jest mało prawdopodobny. Jednak ponieważ Java wymusza przechwytywania wyjątków kontrolowanych, musisz umieścić kod w bloku try-catch. Jeśli metoda sleep jest wywoływana w pętli, umieść pętlę w bloku try-catch tak jak w ramce (a). Jeżeli pętla znajduje się poza blokiem try-catch, tak jak w ramce (b), wątek może kontynuować pracę nawet po wywołaniu interrupt.

```
public void run() {
  try {
    while(...) {
      ...
      Thread.sleep(1000);
    }
  }
catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

(a) Poprawnie

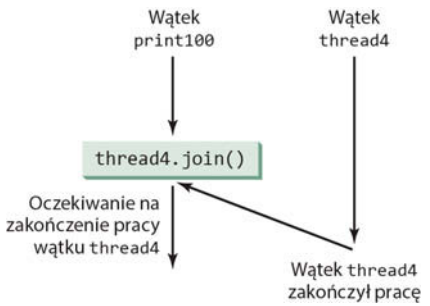
```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex) {
ex.printStackTrace();
    }
  }
}
```

(b) Niepoprawnie

join()

Za pomocą metody join() możesz sprawić, by dany wątek oczekiwał na ukończenie pracy przez inny wątek. Przyjmij, że zmodyfikowałeś kod z wierszy 53. – 57. z listingu 32.1 tak:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print (" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



Kod tworzy nowy wątek thread4 i wyświetla znak *c* 40 razy. Liczby od 50 do 100 są wyświetlane po zakończeniu pracy wątku thread4.

setPriority(int)

Java przypisuje każdemu wątkowi priorytet. Domyślnie nowy wątek dziedziczy priorytet wątku, w którym został utworzony. Możesz zwiększyć lub zmniejszyć priorytet dowolnego wątku, używając metody setPriority. Ponadto możesz pobrać priorytet wątku za pomocą metody getPriority. Priorytety to liczby od 1 do 10. Klasa Thread zawiera stałe MIN_PRIORITY, NORM_PRIORITY i MAX_PRIORITY typu int reprezentujące liczby 1, 5 i 10. Priorytet wątku z metodą main to Thread.NORM_PRIORITY.

szeregowanie cykliczne

Maszyna JVM zawsze wybiera gotowy do uruchomienia wątek o najwyższym priorytecie. Wątek o niższym priorytecie można uruchomić tylko wtedy, gdy nie działają wątki o wyższym priorytecie. Jeżeli wszystkie wykonywalne wątki mają ten sam priorytet, każdemu cyklicznie przydzielana jest równa część czasu procesora (jest to *szeregowanie cykliczne*). Przyjmij, że w wierszu 16. na listingu 32.1 wstawiłeś następujący kod:

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

Wtedy najpierw zakończy pracę wątek `print100`.

Wskazówka

W przyszłych wersjach Javy wartości priorytetów mogą się zmienić. Aby zminimalizować wpływ zmian, do podawania priorytetów wykorzystaj stałe z klasy `Thread`.

Wskazówka

Może się zdarzyć, że wątek w ogóle nie zostanie wykonany, ponieważ zawsze działają wątki o wyższych priorytetach lub wątki o tym samym priorytecie, które nie zwalniają zasobów. Taka sytuacja jest nazywana *rywalizacją* lub *zagłodzeniem*. Aby uniknąć zagłodzenia, wątki o wyższym priorytecie muszą okresowo wywoływać metodę `sleep` lub `yield`, aby dać wątkowi o niższym lub tym samym priorytecie możliwość wykonania zadania.

rywalizacja lub zagłodzenie

32.4.1. Które z poniższych metod są metodami instancji w klasie `java.lang.Thread`? Która metoda może zgłaszać wyjątek `InterruptedException`? Które z tych metod są niezalecane w Javie?

```
run, start, stop, suspend, resume, sleep, interrupt, yield, join
```

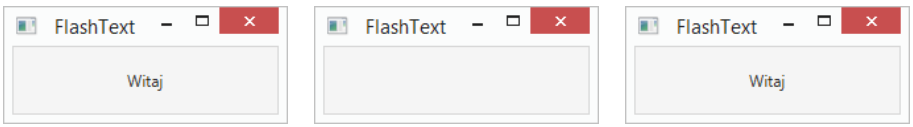
32.4.2. Dlaczego pętlę zawierającą metodę zgłaszającą wyjątki `InterruptedException` należy umieścić w bloku `try-catch`?

32.4.3. Jak ustawić priorytet wątku? Jaki jest priorytet domyślny?

32.5. Animacja z użyciem wątków i metody Platform.runLater

Za pomocą metody `Platform.runLater` możesz używać wątku do sterowania animacją i uruchamiania kodu wątku GUI z JavaFX.

W podrozdziale 15.11, „Animacje”, zobaczyłeś, jak używać obiektu typu `Timeline` do sterowania animacjami. Do zarządzania animacją możesz też użyć wątku. Listing 32.2 ilustruje wyświetlanie migającego tekstu w etykiecie (rysunek 32.6).



Rysunek 32.6. Tekst „Witaj” mruga

Listing 32.2. FlashText.java

```
 1 import javafx.application.Application;
 2 import javafx.application.Platform;
 3 import javafx.scene.Scene;
 4 import javafx.scene.control.Label;
 5 import javafx.scene.layout.StackPane;
 6 import javafx.stage.Stage;
 7
 8 public class FlashText extends Application {
 9   private String text = "";
10
11   @Override // Przesłanianie metody start z klasy Application
```

```
12   public void start(Stage primaryStage) {
13     StackPane pane = new StackPane();
14     Label lblText = new Label("Programowanie to przyjemność");
15     pane.getChildren().add(lblText);
16
17     new Thread(new Runnable() {
18       @Override
19       public void run() {
20         try {
21           while (true) {
22             if (lblText.getText().trim().length() == 0)
23               text = "Witaj";
24             else
25               text = "";
26
27             Platform.runLater(new Runnable() { // Uruchamianie w GUI z JavaFX
28               @Override
29               public void run() {
30                 lblText.setText(text);
31               }
32             });
33
34             Thread.sleep(200);
35           }
36         }
37         catch (InterruptedException ex) {
38         }
39       }
40     }).start();
41
42     // Tworzenie sceny i umieszczanie jej w oknie
43     Scene scene = new Scene(pane, 200, 50);
44     primaryStage.setTitle("FlashText"); // Ustawianie nagłówka okna
45     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
46     primaryStage.show(); // Wyświetlanie okna
47   }
48 }
```

tworzenie etykiety
etykieta w panelu
tworzenie wątku
uruchamianie wątku
modyfikacja tekstu
Platform.runLater
aktualizowanie GUI
usypianie wątku

Ten program tworzy w anonimowej klasie wewnętrznej obiekt typu `Runnable` (wiersze 17. – 40.). Ten obiekt jest uruchamiany w wierszu 40. i działa stale, zmieniając tekst w etykiecie. Jeśli etykieta jest pusta, obiekt dodaje do niej tekst (wiersz 23.), a gdy etykieta wyświetla tekst, obiekt go usuwa (wiersz 25.). Tekst jest dodawany i usuwany, aby zasymulować efekt migotania.

wątek aplikacji opartej na JavaFX

GUI z JavaFX jest uruchamiany w *wątku aplikacji opartej na JavaFX*. Migotanie kontrolki jest przetwarzane w odrębnym wątku. Kod z wątków innych niż wątek aplikacji nie może aktualizować GUI w wątku aplikacji. Aby zaktualizować tekst w etykiecie, w wierszach 27. – 32. należy utworzyć nowy obiekt typu `Runnable`. Wywołanie `Platform.runLater(Runnable r)` informuje system, że należy uruchomić obiekt typu `Runnable` w wątku aplikacji.

Platform.runLater

Anonimowe klasy wewnętrzne w tym programie można uprościć, używając wyrażeń lambda:

```
new Thread(() -> { // Wyrażenie lambda
  try {
    while (true) {
      if (lblText.getText().trim().length() == 0)
        text = "Witaj";
      else
```

```
        text = "";
        Platform.runLater(() -> lblText.setText(text)); // Wyrażenie lambda
        Thread.sleep(200);
      }
    }
    catch (InterruptedException ex) {
    }
}).start();
```

32.5.1. Co powoduje, że tekst migocze?

32.5.2. Czy instancja klasy `FlashText` jest obiektem `Runnable`?

32.5.3. Do czego służy metoda `Platform.runLater`?

32.5.4. Czy możesz zastąpić kod z wierszy 27. – 32. poniższym kodem?

```
Platform.runLater(e -> lblText.setText(text));
```

32.5.5. Co się stanie, jeśli pominiesz wiersz 34. (`Thread.sleep(200)`)?

32.5.6. Na listingu 16.9, *ListViewDemo*, występuje problem. Jeśli wciśniesz klawisz *Ctrl* i zaznaczysz kraje Kanada, Dania, Chiny w tej kolejności, wystąpi wyjątek `ArrayIndexOutBoundsException`. Co jest tego powodem i jak rozwiązać problem? Dziękuję Henriemu Heimonenowi z Finlandii za wkład w to pytanie.

32.6. Pule wątków

Do wydajnego wykonywania zadań można wykorzystać pulę wątków.

W podrozdziale 32.3, „Tworzenie zadań i wątków”, dowiedziałeś się, jak zdefiniować klasę zadania, implementując interfejs `java.lang.Runnable`, a także jak utworzyć wątek do wykonywania zadania:

```
Runnable task = new TaskClass(...);
new Thread(task).start();
```

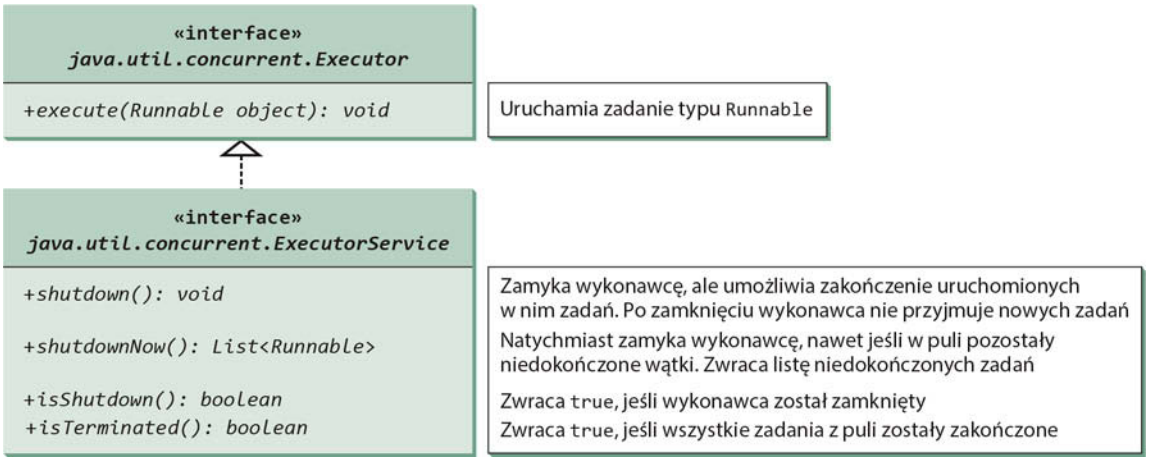
Ta technika jest wygodna do wykonywania jednego zadania. Jest jednak niewydajna, gdy liczba zadań jest duża, ponieważ wymaga to utworzenia wątku dla każdego zadania. Uruchamianie nowego wątku dla wszystkich zadań może zmniejszać przepustowość i prowadzić do spadku wydajności. Użycie *puli wątków* to doskonały sposób na zarządzanie wieloma jednocześnie wykonywanymi zadaniami. Java udostępnia interfejs `Executor` (wykonawca) do wykonywania zadań w puli wątków i interfejs `ExecutorService` do zarządzania zadaniami i kontrolowania ich. `ExecutorService` jest podinterfejsem interfejsu `Executor` (rysunek 32.7).

Aby utworzyć obiekt typu `Executor`, użyj metod statycznych z klasy `Executors` (rysunek 32.8). Metoda `new FixedThreadPool(int)` tworzy pulę ze stałą liczbą wątków. Jeśli wątek zakończy wykonywanie zadania, można go ponownie wykorzystać do uruchomienia następnego zadania. Jeżeli wątek zakończył pracę z powodu awarii, a istnieją zadania oczekujące na wykonanie i nie ma bezczynnych wątków, tworzony jest nowy wątek zastępczy. Metoda `newCachedThreadPool()` tworzy nowy wątek, jeśli w puli nie ma bezczynnych wątków i istnieją zadania oczekujące na wykonanie. Wątek z puli jest zamykany, gdy nie jest używany od 60 sekund. Pula z buforem umożliwia wydajne wykonywanie wielu krótkich zadań.

Listing 32.3 to wersja listingu 32.1 wykorzystująca pulę wątków.

LISTING 32.3. ExecutorDemo.java

```
1 import java.util.concurrent.*;
2
3 public class ExecutorDemo {
```



Rysunek 32.7. Interfejs Executor wykonuje wątki, a podinterfejs ExecutorService zarządza nimi

Rysunek 32.8. Klasa Executors udostępnia metody statyczne do tworzenia obiektów typu Executor

```
 4   public static void main(String[] args) {
 5     // Tworzenie puli z maksymalną liczbą trzech wątków
 6     ExecutorService executor = Executors.newFixedThreadPool(3);
 7
 8     // Przekazywanie zadań typu Runnable do wykonawcy
 9     executor.execute(new PrintChar('a', 100));
10     executor.execute(new PrintChar('b', 100));
11     executor.execute(new PrintNum(100));
12
13     // Zamykanie wykonawcy
14     executor.shutdown();
15   }
16 }
```

tworzenie wykonawcy

przekazywanie zadania

zamykanie wykonawcy

W wierszu 6. kod tworzy pulę obejmującą maksymalnie trzy wątki. Klasy `PrintChar` i `PrintNum` są zdefiniowane na listingu 32.1. W wierszu 9. program tworzy zadanie, `new PrintChar('a', 100)`, i dodaje je do puli. W wierszach 10. i 11. tworzone i dodawane do tej samej puli są dwa inne zadania typu `Runnable`. Wykonawca tworzy trzy wątki do jednoczesnego wykonywania trzech zadań.

Przyjmij, że wiersz 6. zastąpiłeś następującym kodem:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

Co się wtedy stanie? Trzy zadania typu `Runnable` będą wykonywane sekwencyjnie, ponieważ w puli dostępny będzie tylko jeden wątek.

Przyjmij, że wiersz 6. zastąpiłeś następującym kodem:

```
ExecutorService executor = Executors.newCachedThreadPool();
```

Co się stanie teraz? Dla każdego oczekującego zadania utworzony zostanie nowy wątek, dlatego wszystkie zadania będą wykonywane równolegle.

Metoda `shutdown()` z wiersza 14. powoduje zamknięcie wykonawcy. Żadne nowe zadania nie są wtedy akceptowane, ale bieżące zadania zostają ukończone.

Wskazówka

Jeśli chcesz utworzyć wątek dla tylko jednego zadania, użyj klasy `Thread`. Jeżeli potrzebujesz wątków do wykonywania wielu zadań, lepiej jest użyć puli.

32.6.1. Jakie są korzyści używania puli wątków?

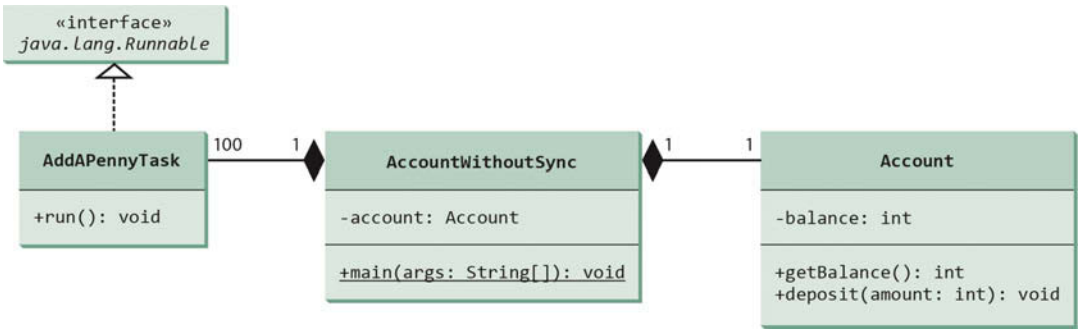
32.6.2. Jak utworzyć pulę o stałej liczbie trzech wątków? Jak przekazać zadanie do puli wątków? Skąd wiadomo, że wszystkie zadania ukończyły pracę?

32.7. Synchronizacja wątków

Synchronizacja wątków pozwala skoordynować wykonywanie zależnych od siebie wątków.

Współużytkowane zasoby mogą zostać uszkodzone, jeśli wiele wątków będzie jednocześnie z nich korzystać. Następny przykład ilustruje ten problem.

Wyobraź sobie, że utworzyłeś i uruchomiłeś 100 wątków, z których każdy może dodać grosz do konta. Zdefiniuj klasę `Account` do modelowania konta, klasę `AddAPennyTask` do dodawania grosza do konta oraz główną klasę do tworzenia i uruchamiania wątków. Zależności między tymi klasami są pokazane na rysunku 32.9, a kod programu znajdziesz na listingu 32.4.



Rysunek 32.9. Klasa AccountWithoutSync zawiera obiekt typu Account i 100 wątków z zadaniem AddAPennyTask

Listing 32.4. AccountWithoutSync.java

```
1 import java.util.concurrent.*;
2
3 public class AccountWithoutSync {
4   private static Account account = new Account();
```

tworzenie wykonawcy

przekazywanie zadania

zamykanie wykonawcy

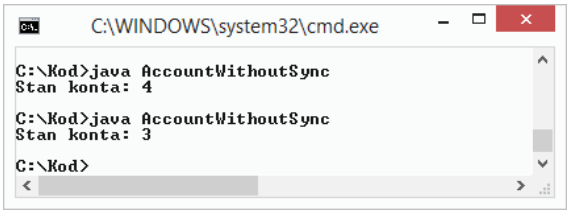
oczekiwanie na ukończenie wszystkich zadań

```
 5
 6   public static void main(String[] args) {
 7     ExecutorService executor = Executors.newCachedThreadPool();
 8
 9     // Tworzenie i uruchamianie 100 wątków
10     for (int i = 0; i < 100; i++) {
11       executor.execute(new AddAPennyTask());
12     }
13
14     executor.shutdown();
15
16     // Oczekiwanie na ukończenie wszystkich zadań
17     while (!executor.isTerminated()) {
18     }
19
20     System.out.println("Stan konta: " + account.getBalance());
21   }
22
23   // Wątek dodający grosz do konta
24   private static class AddAPennyTask implements Runnable {
25     public void run() {
26       account.deposit(1);
27     }
28   }
29
30   // Klasa wewnętrzna reprezentująca konto
31   private static class Account {
32     private int balance = 0;
33
34     public int getBalance() {
35       return balance;
36     }
37
38     public void deposit(int amount) {
39       int newBalance = balance + amount;
40
41       // Opóźnienie jest dodane celowo, aby nasilić i
42       // uwidocznić problem uszkadzania danych
43       try {
44         Thread.sleep(5);
45       }
46       catch (InterruptedException ex) {
47       }
48
49       balance = newBalance;
50     }
51   }
52 }
```

Klasy `AddAPennyTask` i `Account` z wierszy 24. – 51. to klasy wewnętrzne. W wierszu 4. tworzone jest konto z początkowym stanem 0. W wierszu 11. program tworzy zadanie dodające grosz do konta i przekazuje to zadanie do wykonawcy. Wiersz 11. jest w wierszach 10. – 12. wykonywany 100 razy. W wierszach 17. i 18. program wielokrotnie sprawdza, czy wszystkie zadania zakończyły pracę. W wierszu 20., po ukończeniu wszystkich zadań, wyświetlany jest stan konta.

Program tworzy 100 wątków wykonywanych w puli `executor` (wiersze 10. – 12.). Metoda `isTerminated()` (wiersz 17.) służy do sprawdzania, czy wszystkie wątki z puli zakończyły pracę.

Stan konta początkowo wynosi 0 (wiersz 32.). Po zakończeniu pracy wszystkich wątków powinien wynosić 100, ale dane wejściowe są nieprzewidywalne. Na rysunku 32.10 widać, że w przykładowych przebiegach wyniki są błędne. To ilustruje problem uszkodzenia danych występujący, gdy wszystkie wątki mają jednoczesny dostęp do tego samego źródła danych.



Rysunek 32.10. Dane w programie AccountWithoutSync są niespójne

Wiersze 39. – 49. można zastąpić jedną instrukcją:

```
balance = balance + amount;
```

Wystąpienie problemu, gdy używana jest ta pojedyncza instrukcja, jest mało prawdopodobne (choć możliwe). Instrukcje z wierszy 39. – 49. zostały celowo tak dobrane, by nasilać problem uszkodzenia danych i ułatwiać dostrzeżenie go. Jeśli uruchomisz program kilkakrotnie i nadal nie dostrzeżesz problemu, wydłuż czas uśpienia w wierszu 44. Zwiększy to prawdopodobieństwo wykrycia problemu niespójności w danych.

Co jest powodem błędu w programie? Rysunek 32.11 obrazuje możliwy scenariusz.

Krok	Stan konta	Zadanie 1.	Zadanie 2.
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

Rysunek 32.11. Zadania 1 i 2 dodają 1 do tego samego stanu konta

W kroku 1. zadanie 1. pobiera stan konta. W kroku 2. zadanie 2. pobiera ten sam stan konta. W kroku 3. zadanie 1. zapisuje nowy stan konta. W kroku 4. zadanie 2. zapisuje nowy stan konta.

sytuacja wyścigu bezpieczna ze względu na wątki

Skutek tego jest taki, że zadanie 1. nie wykonuje żadnej operacji, ponieważ w kroku 4. zadanie 2. nadpisuje wartość zapisaną przez zadanie 1. Problemem jest tu oczywiście to, że zadania 1. i 2. używają wspólnego zasobu w sposób powodujący konflikt. Jest to problem często występujący w programach wielowątkowych. Nazywa się go *sytuacją wyścigu*. Klasa jest *bezpieczna ze względu na wątki*, jeśli obiekt tej klasy nie powoduje w kodzie wielowątkowym sytuacji wyścigu. Pokazany przykład obrazuje, że klasa `Account` nie jest bezpieczna ze względu na wątki.

32.7.1. Słowo kluczowe synchronized

sekcja krytyczna

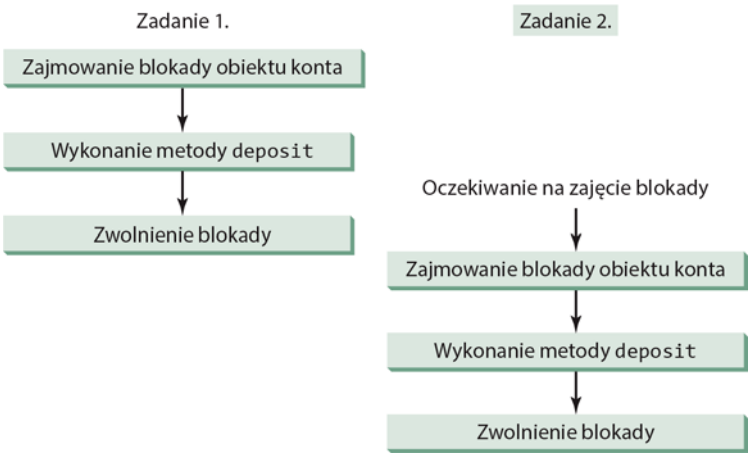
Aby uniknąć sytuacji wyścigu, trzeba zapobiec jednoczesnemu wkroczeniu przez więcej niż jeden wątek do określonej części programu nazywanej *sekcją krytyczną*. Na listingu 32.4 sekcją krytyczną jest cała metoda `deposit`. Możesz użyć słowa kluczowego `synchronized`, aby zsynchronizować metodę, dzięki czemu w danym momencie tylko jeden

wątek będzie miał do niej dostęp. Istnieje kilka sposobów na poprawienie kodu z listingu 32.4. Jedną z możliwości jest utworzenie klasy `Account` jako bezpiecznej ze względu na wątki. W tym celu dodaj słowo kluczowe `synchro ↪nized` w metodzie `deposit` w wierszu 38.:

```
public synchronized void deposit(double amount)
```

Metoda synchronizowana zajmuje blokadę przed rozpoczęciem pracy. Blokada jest mechanizmem pozwalającym na wyłączne używanie zasobów. W przypadku metody instancji blokada dotyczy obiektu, dla którego metoda jest wywoływana. Gdy używana jest metoda statyczna, blokada dotyczy klasy. Jeśli wątek wywoła synchronizowaną metodę instancji (lub statyczną) dla obiektu, najpierw zajmowana jest blokada tego obiektu (lub klasy), potem metoda jest wywoływana, a na końcu blokada jest zwalniana. Inny wątek chcący wywołać tę samą metodę danego obiektu (lub klasy) zostaje zablokowany do czasu zwolnienia blokady.

Gdy metoda `deposit` jest synchronizowana, opisany wcześniej scenariusz jest niemożliwy. Jeśli zadanie 1. uruchomi metodę, zadanie 2. zostanie zablokowane do czasu ukończenia pracy przez zadanie 1. (rysunek 32.12).



Rysunek 32.12. Zadania 1. i 2. są synchronizowane

32.7.2. Instrukcje synchronizacji

Wywołanie synchronizowanej metody instancji dla obiektu powoduje zajęcie jego blokady, a wywołanie synchronizowanej metody statycznej blokuje daną klasę. Synchronizowana instrukcja pozwala zająć blokadę dowolnego obiektu (nie tylko *bieżącego* obiektu) na czas wykonywania bloku kodu w metodzie. Taki blok jest nazywany *synchronizowanym*. Ogólna postać instrukcji synchronizowanych wygląda tak:

blok synchronizowany

```
synchronized (wyrażenie) {
  instrukcje;
}
```

Wartością wyrażenia musi tu być referencja do obiektu. Jeśli dany obiekt jest już zablokowany przez inny wątek, nowy wątek zostanie zablokowany do czasu zwolnienia blokady. Po zajęciu blokady instrukcje z synchronizowanego bloku są wykonywane, po czym blokada jest zwalniana.

Synchronizowane instrukcje umożliwiają synchronizowanie fragmentu kodu metody zamiast całej metody, dzięki czemu rośnie współbieżność. Możesz sprawić, by kod z listingu 32.4 był bezpieczny ze względu na wątki, umieszczając tę instrukcję w wierszu 26. synchronizowanego bloku:

```
synchronized (account) {
  account.deposit(1);
}
```

Uwaga

Każdą synchronizowaną metodę instancji można przekształcić w instrukcję synchronizowaną. Synchronizowana metoda instancji z ramki (a) to odpowiednik kodu z ramki (b):

```
public synchronized void xMethod() {
  // Ciało metody
}
```

(a)

```
public void xMethod() {
  synchronized (this) {
    // Ciało metody
  }
}
```

(b)

32.7.1. Podaj przykład ilustrujący możliwe uszkodzenie zasobów, gdy uruchamianych jest wiele wątków. Jak zsynchronizować wątki powodujące konflikty?

32.7.2. Przyjmij, że instrukcję z wiersza 26. listingu 32.4 umieściłeś w bloku synchronizowanym, aby uniknąć sytuacji wyścigu:

```
synchronized (this) {
  account.deposit(1);
}
```

Czy to rozwiązanie zadziała?

32.8. Synchronizacja z użyciem blokad

Do synchronizowania wątków można bezpośrednio zastosować blokady i warunki.

Na listingu 32.4 100 zadań równolegle deponowało grosz na tym samym koncie, co powodowało konflikty. Aby uniknąć tego problemu, możesz użyć w metodzie `deposit` słowa kluczowego `synchronized`:

```
public synchronized void deposit(double amount)
```

blokada

Synchronizowana metoda instancji automatycznie zajmuje *blokadę* instancji przed wykonaniem kodu.

Java umożliwia też bezpośrednie zajmowanie blokad, co zapewnia większą kontrolę w zakresie koordynowania wątków. Blokada jest instancją interfejsu `Lock`, który definiuje metody do zajmowania i zwalniania blokad (rysunek 32.13). Blokada pozwala też użyć metody `newCondition()` do utworzenia dowolnej liczby obiektów typu `Condition`, które umożliwiają komunikację między wątkami.

polityka uczciwości

`ReentrantLock` to konkretna implementacja interfejsu `Lock` służąca do tworzenia blokad ze wzajemnym wykluczaniem. Ta klasa pozwala utworzyć blokadę z określoną *polityką uczciwości*. Wartość `true` gwarantuje, że jako pierwszy otrzyma blokadę wątek o najdłuższym czasie oczekiwania. Wartość `false` powoduje przyznawanie blokad dowolnym wątkom. Programy z „uczciwymi” blokadami używane przez wiele wątków mogą mieć niższą ogólną wydajność w porównaniu z ustawieniami domyślnymi, ale cechują się niższą wariancją czasu oczekiwania na blokady i chronią wątki przed zagłodzeniem.

Listing 32.5 to zmodyfikowana wersja programu z listingu 32.4. Modyfikowanie konta jest tu synchronizowane z użyciem bezpośrednio dodawanych blokad.

RYSUNEK 32.13. Klasa ReentrantLock implementuje interfejs Lock i reprezentuje blokadę

LISTING 32.5. AccountWithSyncUsingLock.java

polityka uczciwości

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;
3
4  public class AccountWithSyncUsingLock {
5    private static Account account = new Account();
6
7    public static void main(String[] args) {
8      ExecutorService executor = Executors.newCachedThreadPool();
9
10     // Tworzenie i uruchamianie 100 wątków
11     for (int i = 0; i < 100; i++) {
12       executor.execute(new AddAPennyTask());
13     }
14
15     executor.shutdown();
16
17     // Oczekiwanie na ukończenie wszystkich zadań
18     while (!executor.isTerminated()) {
19     }
20
21     System.out.println("Stan konta: " + account.getBalance());
22   }
23
24   // Wątek dodający grosz do konta
25   public static class AddAPennyTask implements Runnable {
26     public void run() {
27       account.deposit(1);
28     }
29   }
30
31   // Klasa wewnętrzna Account
32   public static class Account {
33     private static Lock lock = new ReentrantLock(); // Tworzenie blokady
34     private int balance = 0;
```

tworzenie blokady

```
35
36      public int getBalance() {
37        return balance;
38      }
39
40      public void deposit(int amount) {
41        lock.lock(); // Zajmowanie blokady
42
43        try {
44          int newBalance = balance + amount;
45
46          // Opóźnienie jest dodane celowo, aby nasilić i uwidocznić
47          // problemy z uszkadzaniem danych
48          Thread.sleep(5);
49
50          balance = newBalance;
51        }
52        catch (InterruptedException ex) {
53        }
54        finally {
55          lock.unlock(); // Zwalnianie blokady
56        }
57      }
58    }
59 }
```

zajmowanie blokady

zwalnianie blokady

W wierszu 33. program tworzy blokadę, w wierszu 41. zajmuje ją, a w wierszu 55. zwalnia.

Wskazówka

Dobrą praktyką jest zawsze dodawać blok `try-catch` po wywołaniu `lock()` i zwalniać blokadę w klauzuli `finally` (wiersze 41. – 56.), aby mieć pewność, że blokada zawsze zostanie zwolniona.

Kod z listingu 32.5 można zaimplementować z użyciem synchronizowanej metody `deposit`, zamiast bezpośrednio dodając blokadę. Zwykle używanie metod i instrukcji synchronizowanych jest prostsze niż bezpośrednie stosowanie blokad. Jednak bezpośrednie blokady są bardziej intuicyjne i dają większą swobodę w zakresie synchronizowania wątków z wykorzystaniem warunków. Przekonasz się o tym w następnym podrozdziale.

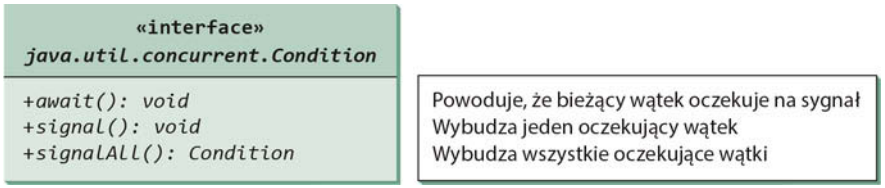
32.8.1. Jak utworzyć obiekt blokady? Jak zająć i zwolnić blokadę?

32.9. Współdziałanie między wątkami

Do koordynowania interakcji między wątkami można wykorzystać warunki dotyczące blokad.

Synchronizowanie wątków wystarcza do uniknięcia sytuacji wyścigu, ponieważ wyklucza jednoczesne działanie wielu wątków w sekcji krytycznej. Jednak czasem potrzebny jest mechanizm współdziałania wątków. Do ułatwienia komunikacji między wątkami można wykorzystać *warunki*. Wątek może określać, co należy robić w ustalonych warunkach. Warunki są obiektami tworzonymi w wyniku wywołania metody `newCondition()` obiektu typu `Lock`. Po utworzeniu warunku można używać metod `await()`, `signal()` i `signalAll()` do komunikacji między wątkami (rysunek 32.14). Metoda `await()` powoduje, że bieżący wątek oczekuje na sygnał. Metoda `signal()` wybudza jeden oczekujący wątek, a metoda `signalAll()` wybudza wszystkie oczekujące wątki.

warunek

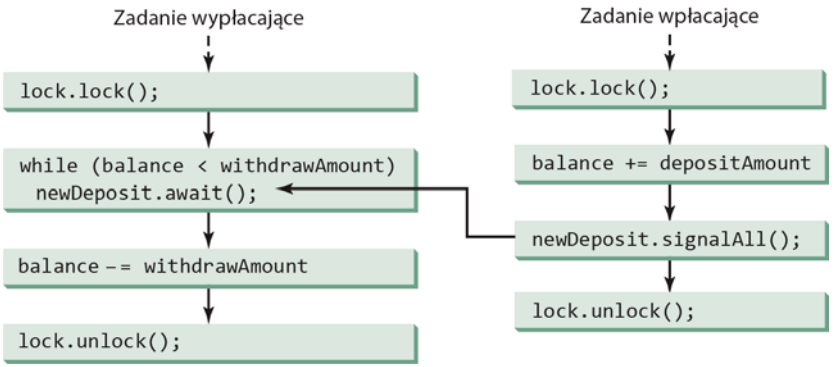


Rysunek 32.14. Interfejs Condition definiuje metody do przeprowadzania synchronizacji

przykład współdziałania wątków

Oto przykład ilustrujący komunikację między wątkami: przyjmij, że tworzysz i uruchamiasz dwa zadania — jedno powoduje zdeponowanie kwoty na rachunku, drugie wypłacenie środków z tego samego rachunku. Zadanie wypłacające środki musi oczekiwać, jeśli wypłacana kwota przekracza aktualny stan konta. Po zdeponowaniu nowych środków zadanie powodujące wpłacenie pieniądze powiadamia wątek wypłacający, co wznawia pracę tego ostatniego. Jeżeli stan konta nadal nie pozwala na dokonanie wypłaty, wątek wypłacający ponownie zaczyna oczekiwać na nową wpłatę.

Aby zsynchronizować te operacje, użyj blokady z warunkiem `newDeposit` (dotyczy on nowej wpłaty na konto). Jeśli stan konta jest niższy niż wypłacane środki, zadanie wypłacające oczekuje na warunek `newDeposit`. Gdy zadanie wpłacające doda środki do konta, sygnalizuje to oczekującemu zadaniu wypłacającemu, które może wtedy podjąć kolejną próbę. Interakcje między tymi dwoma zadaniami są przedstawione na rysunku 32.15.



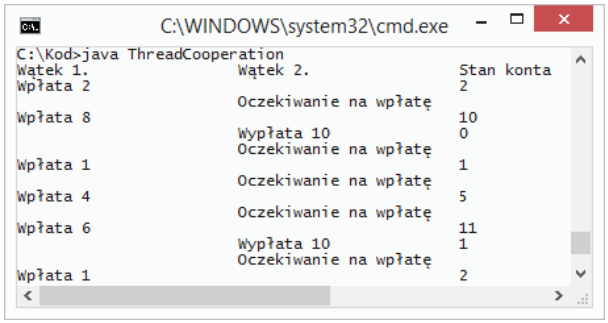
Rysunek 32.15. Warunek newDeposit służy do komunikacji między dwoma wątkami

Warunek można utworzyć za pomocą obiektu typu `Lock`. Aby użyć warunku, najpierw należy zająć blokadę. Metoda `await()` powoduje, że wątek oczekuje na sygnał i automatycznie zwalnia blokadę powiązaną z warunkiem. Gdy warunek jest spełniony, wątek ponownie zajmuje blokadę i kontynuuje działanie.

Przyjmij, że początkowy stan konta to 0, a deponowane i wypłacane kwoty są generowane losowo. Program pokazany jest na listingu 32.6, a jego przykładowy przebieg — na rysunku 32.16.

Listing 32.6. ThreadCooperation.java

```
1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 public class ThreadCooperation {
5   private static Account account = new Account();
6
7   public static void main(String[] args) {
```



RYSUNEK 32.16. Jeśli stan konta nie pozwala na wypłatę, zadanie wypłacające oczekuje na sygnał

```
8     // Tworzenie puli z dwoma wątkami
9     ExecutorService executor = Executors.newFixedThreadPool(2);
10    executor.execute(new DepositTask());
11    executor.execute(new WithdrawTask());
12    executor.shutdown();
13
14    System.out.println("Wątek 1.\t\tWątek 2.\t\tStan konta");
15  }
16
17  public static class DepositTask implements Runnable {
18    @Override // Dodawanie środków do konta
19    public void run() {
20      try {  // Celowe opóźnienie, aby umożliwić pracę metodzie wypłacającej środki
21        while (true) {
22          account.deposit((int)(Math.random() * 10) + 1);
23          Thread.sleep(1000);
24        }
25      }
26      catch (InterruptedException ex) {
27        ex.printStackTrace();
28      }
29    }
30  }
31
32  public static class WithdrawTask implements Runnable {
33    @Override // Wypłacanie środków z konta
34    public void run() {
35      while (true) {
36        account.withdraw((int)(Math.random() * 10) + 1);
37      }
38    }
39  }
40
41  // Klasa wewnętrzna Account
42  private static class Account {
43    // Tworzenie nowej blokady
44    private static Lock lock = new ReentrantLock();
45
46    // Tworzenie warunku
47    private static Condition newDeposit = lock.newCondition();
```

tworzenie 2 wątków

tworzenie blokady

tworzenie warunku

```
48
49      private int balance = 0;
50
51      public int getBalance() {
52        return balance;
53      }
54
55      public void withdraw(int amount) {
56        lock.lock(); // Zajmowanie blokady
57        try {
58          while (balance < amount) {
59            System.out.println("\t\t\tOczekiwanie na wpłatę");
60            newDeposit.await();
61          }
62
63          balance -= amount;
64          System.out.println("\t\t\tWypłata " + amount +
65            "\t\t" + getBalance());
66        }
67        catch (InterruptedException ex) {
68          ex.printStackTrace();
69        }
70        finally {
71          lock.unlock(); // Zwalnianie blokady
72        }
73      }
74
75      public void deposit(int amount) {
76        lock.lock(); // Zajmowanie blokady
77        try {
78          balance += amount;
79          System.out.println("Wpłata " + amount +
80            "\t\t\t\t\t" + getBalance());
81
82          // Sygnał dla wątku oczekującego na warunek
83          newDeposit.signalAll();
84        }
85        finally {
86          lock.unlock(); // Zwalnianie blokady
87        }
88      }
89    }
90 }
```

zajmowanie blokady

oczekiwanie na warunek

zwalnianie blokady

zajmowanie blokady

sygnał dla wątków

zwalnianie blokady

Ten kod tworzy nową klasę wewnętrzną, `Account`, reprezentującą konto i udostępniającą dwie metody: `depo➥sit(int)` i `withdraw(int)`. Program tworzy też klasę `DepositTask`, która dodaje środki do konta, i klasę `Withdraw➥Task` do wypłacania środków. Klasa główna tworzy i uruchamia dwa wątki.

Program tworzy i przesyła zadania `DepositTask` (wiersz 10.) i `WithdrawTask` (wiersz 11.). Zadanie `DepositTask` jest celowo usypiane (wiersz 23.), aby umożliwić uruchomienie zadania `WithdrawTask`. Gdy środki nie wystarczają na wypłatę, zadanie `WithdrawTask` oczekuje (wiersz 59.) na sygnał zmiany stanu konta od zadania `DepositTask` (wiersz 83.).

W wierszu 44. zajmowana jest blokada. W wierszu 47. tworzony jest warunek `newDeposit` powiązany z tą blokadą. Wątek przed rozpoczęciem oczekiwania lub przesłaniem sygnału musi najpierw zająć blokadę. Zadanie `WithdrawTask`

zajmuje blokadę w wierszu 56., oczekuje na warunek `newDeposit` (wiersz 60.), jeśli na koncie nie ma wystarczających środków, i zwalnia blokadę (wiersz 71.). Zadanie `DepositTask` zajmuje blokadę w wierszu 76., a po dokonaniu wpłaty przekazuje w wierszu 83. wszystkim wątkom oczekującym sygnał powiązany z warunkiem `newDeposit`.

Co się stanie, jeśli zastąpisz pętlę `while` (wiersze 58. – 61.) poniższą instrukcją `if`?

```
if (balance < amount) {
  System.out.println("\t\t\tOczekiwanie na wpłatę");
  newDeposit.await();
}
```

Zadanie wpłacające każdorazowo po zmianie stanu konta powiadamia o tym zadanie wypłacające. Po wybudzeniu zadania wypłacającego warunek (`balance < amount`) nadal może mieć wartość `true`. Użycie instrukcji `if` spowoduje wtedy niedozwoloną wypłatę. Dzięki pętli zadanie wypłacające może ponownie sprawdzić warunek.

oczekiwanie wątku w nieskończoność

Ostrzeżenie

Gdy zadanie wywoła metodę `await()` warunku, wątek oczekuje na sygnał, aby wznowić pracę. Jeśli zapomnisz wywołać metodę `signal()` lub `signalAll()` warunku, taki wątek będzie czekał w nieskończoność.

IllegalMonitorStateException

Ostrzeżenie

Warunek jest tworzony na podstawie obiektu typu `Lock`. Aby wywołać metody warunku (`await()`, `signal()`, `signalAll()`), trzeba najpierw zająć blokadę. Jeśli wywołasz te metody bez zajęcia warunku, zgłoszony zostanie wyjątek `IllegalMonitorStateException`.

wbudowane monitory z Javy

Blokady i warunki zostały wprowadzone w Javie 5. W starszych wersjach komunikacja między wątkami była programowana za pomocą wbudowanych monitorów. Blokady i warunki dają więcej możliwości niż wbudowane monitory, dlatego te ostatnie stały się zbędne. Jeśli jednak pracujesz nad starszym kodem w Javie, możesz natrafić na wbudowane monitory.

monitor

Monitor jest obiektem do obsługi wzajemnego wykluczania i synchronizacji. W danym momencie tylko jeden wątek może wykonywać metodę z monitora. Wątek wchodzi do monitora, zajmując blokadę powiązaną z tym monitorem. Wyjście z monitora następuje po zwolnieniu blokady. *Monitorem może być dowolny obiekt*. Obiekt staje się monitorem po zablokowaniu go przez wątek. Blokowanie jest implementowane poprzez dodanie słowa kluczowego `synchronized` do metody lub bloku. Wątek musi zająć blokadę przed wykonaniem synchronizowanej metody lub synchronizowanego bloku. Wątek może oczekiwać w monitorze, jeśli warunek nie pozwala na kontynuowanie pracy. Metoda `wait()` monitora pozwala zwolnić blokadę, dzięki czemu inny wątek może wejść do monitora i ewentualnie zmienić jego stan. Gdy warunek umożliwia kontynuowanie pracy, ten inny wątek może wywołać metodę `notify()` albo `notifyAll()`, by przesłać sygnał jednemu lub wszystkim oczekującym wątkom, które mogą następnie ponownie zająć blokadę i wznowić pracę. Szablon do wykonywania takich metod jest pokazany na rysunku 32.17.

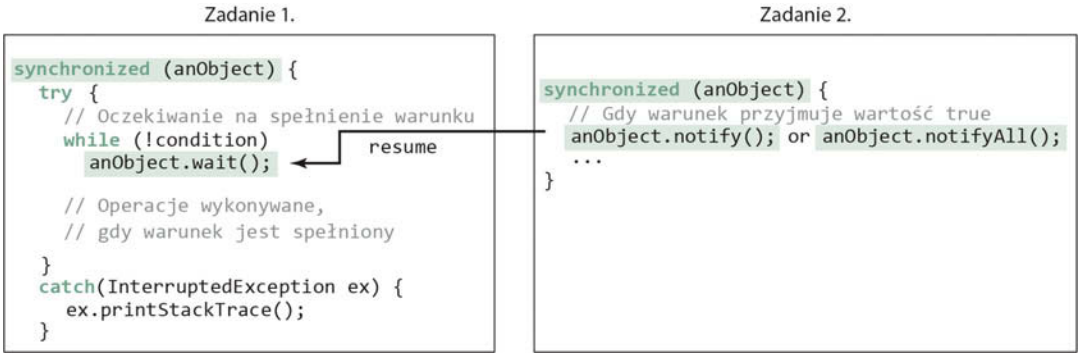
Metody `wait()`, `notify()` i `notifyAll()` trzeba wywołać w synchronizowanej metodzie lub synchronizowanym bloku dla obiektu obsługującego te metody. W przeciwnym razie wystąpi wyjątek `IllegalMonitorStateException`.

Wywołanie metody `wait()` powoduje wstrzymanie wątku i zwolnienie blokady obiektu. Gdy wątek wznawia pracę po otrzymaniu sygnału, blokada jest automatycznie ponownie zajmowana.

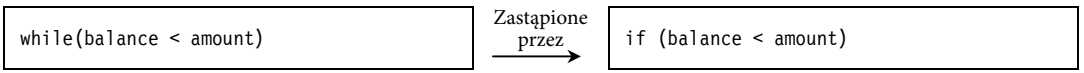
Metody `wait()`, `notify()` i `notifyAll()` dla obiektu są odpowiednikiem metod `await()`, `signal()` i `signalAll()` wywoływanych dla warunku.

32.9.1. Jak utworzyć warunek powiązany z blokadą? Do czego służą metody `await()`, `signal()` i `signalAll()`?

32.9.2. Co się stanie, jeśli pętlę `while` z wiersza 58. listingu 32.6 zastąpisz instrukcją `if`?



RYSUNEK 32.17. Metody wait(), notify() i notifyAll() służą do koordynowania komunikacji między wątkami



32.9.3. Dlaczego w poniższej klasie znajduje się błąd składni?

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() throws InterruptedException {
    Thread thread = new Thread(this);
    thread.sleep(1000);
  }

  public synchronized void run() {
  }
}
```

32.9.4. Z jakiego powodu może zostać zgłoszony wyjątek `IllegalMonitorStateException`?

32.9.5. Czy metody `wait()`, `notify()` i `notifyAll()` można wywołać dla dowolnego obiektu? Do czego służą te metody?

32.9.6. Jaki błąd znajduje się w tym kodzie?

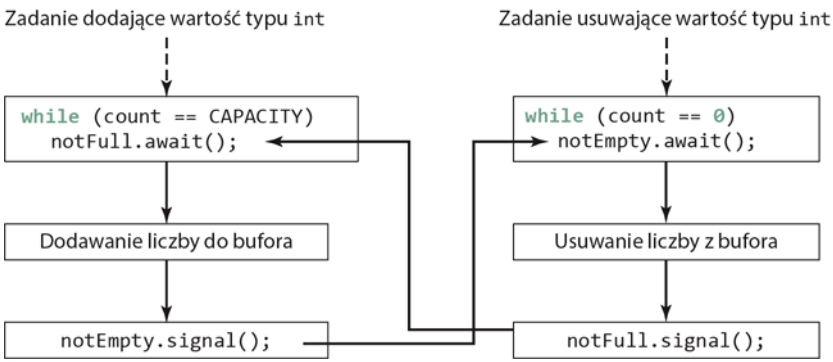
```
synchronized (object1) {
  try {
    while(!condition) object2.wait();
  }
  catch (InterruptedException ex) {
  }
}
```

32.10. Studium przypadku: wzorzec producent/konsument

W tym podrozdziale do zilustrowania koordynacji wątków posłuży klasyczny wzorzec producent/konsument.

Przyjmij, że do przechowywania liczb całkowitych używasz bufora o ograniczonej wielkości. Bufor udostępnia metodę `write(int)`, która pozwala dodać wartość typu `int` do bufora, i metodę `read()`, która wczytuje i usuwa

wartość typu `int`. Aby zsynchronizować operacje, należy użyć dwóch warunków: `notEmpty` (bufor nie jest pusty) i `notFull` (bufor nie jest pełny). Gdy zadanie dodaje wartość typu `int` do pełnego bufora, należy czekać na warunek `notFull`. Gdy zadanie wczytuje wartość z pustego bufora, należy poczekać na warunek `notEmpty`. Interakcję między zadaniami ilustruje rysunek 32.18.



RYSUNEK 32.18. Do koordynowania interakcji między zadaniami służą warunki notFull i notEmpty

Na listingu 32.7 pokazany jest kompletny program. Obejmuje on klasę `Buffer` (wiersze 50. – 101.) i dwa zadania, które wielokrotnie dodają i pobierają liczby (wiersze 16. – 47.). Metoda `write(int)` (wiersze 62. – 79.) dodaje liczbę całkowitą do bufora. Metoda `read()` (wiersze 81. – 100.) usuwa liczbę z bufora i zwraca ją.

Bufor działa tu jak kolejka FIFO (wiersze 52. i 53.). W wierszach 59. i 60. tworzone są warunki `notEmpty` i `notFull` powiązane z blokadą. Przed użyciem warunku trzeba zająć blokadę. Jeśli zmodyfikujesz przykład, używając metod `wait()` i `notify()`, będziesz musiał zastosować dwa obiekty jako monitory.

LISTING 32.7. ConsumerProducer.java

```
 1 import java.util.concurrent.*;
 2 import java.util.concurrent.locks.*;
 3
 4 public class ConsumerProducer {
tworzenie bufora
 5   private static Buffer buffer = new Buffer();
 6
 7   public static void main(String[] args) {
 8     // Tworzenie puli z dwoma wątkami
tworzenie 2 wątków
 9     ExecutorService executor = Executors.newFixedThreadPool(2);
10     executor.execute(new ProducerTask());
11     executor.execute(new ConsumerTask());
12     executor.shutdown();
13   }
14
15   // Zadanie dodające liczbę do bufora
zadanie producenta
16   private static class ProducerTask implements Runnable {
17     public void run() {
18       try {
19         int i = 1;
20         while (true) {
21           System.out.println("Producent zapisał " + i);
22           buffer.write(i++); // Dodawanie wartości do bufora
23           // Usypianie wątku
```

```
24            Thread.sleep((int)(Math.random() * 10000));
25          }
26        }
27        catch (InterruptedException ex) {
28          ex.printStackTrace();
29        }
30      }
31    }
32
33    // Zadanie wczytujące i usuwające liczbę z bufora
34    private static class ConsumerTask implements Runnable {
35      public void run() {
36        try {
37          while (true) {
38            System.out.println("\t\t\tKonsument wczytał " + buffer.read());
39            // Usypianie wątku
40            Thread.sleep((int)(Math.random() * 10000));
41          }
42        }
43        catch (InterruptedException ex) {
44          ex.printStackTrace();
45        }
46      }
47    }
48
49    // Klasa wewnętrzna Buffer
50    private static class Buffer {
51      private static final int CAPACITY = 1; // Wielkość bufora
52      private java.util.LinkedList<Integer> queue =
53        new java.util.LinkedList<>();
54
55      // Tworzenie nowej blokady
56      private static Lock lock = new ReentrantLock();
57
58      // Tworzenie dwóch warunków
59      private static Condition notEmpty = lock.newCondition();
60      private static Condition notFull = lock.newCondition();
61
62      public void write(int value) {
63        lock.lock(); // Zajmowanie blokady
64        try {
65          while (queue.size() == CAPACITY) {
66            System.out.println("Oczekiwanie na warunek notFull");
67            notFull.await();
68          }
69
70          queue.offer(value);
71          notEmpty.signal(); // Sygnalizowanie warunku notEmpty
72        }
73        catch (InterruptedException ex) {
74          ex.printStackTrace();
75        }
76        finally {
77          lock.unlock(); // Zwalnianie blokady
78        }
```

zadanie konsumenta

tworzenie blokady

tworzenie warunku

tworzenie warunku

zajmowanie blokady

oczekiwanie na notFull

sygnalizowanie warunku notEmpty

zwalnianie blokady

```
79      }
80
81      public int read() {
82        int value = 0;
83        lock.lock(); // Zajmowanie blokady
84        try {
85          while (queue.isEmpty()) {
86            System.out.println("\t\t\tOczekiwanie na warunek notEmpty");
87            notEmpty.await();
88          }
89
90          value = queue.remove();
91          notFull.signal(); // Sygnalizowanie warunku notFull
92        }
93        catch (InterruptedException ex) {
94          ex.printStackTrace();
95        }
96        finally {
97          lock.unlock(); // Zwalnianie blokady
98          return value;
99        }
100     }
101   }
102 }
```

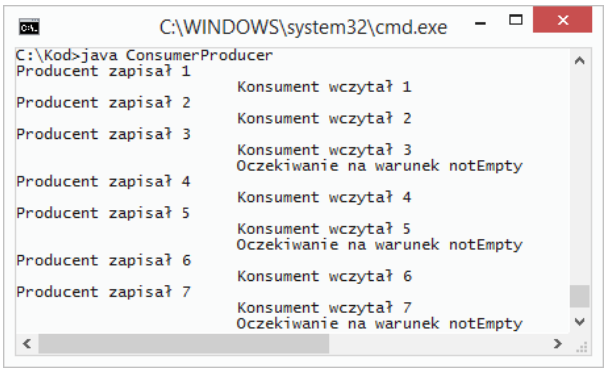
zajmowanie blokady

oczekiwanie na notEmpty

sygnalizowanie warunku notFull

zwalnianie blokady

Przykładowy przebieg tego programu jest pokazany na rysunku 32.19.



RYSUNEK 32.19. Blokady i warunki są używane do komunikacji między wątkami producenta i konsumenta

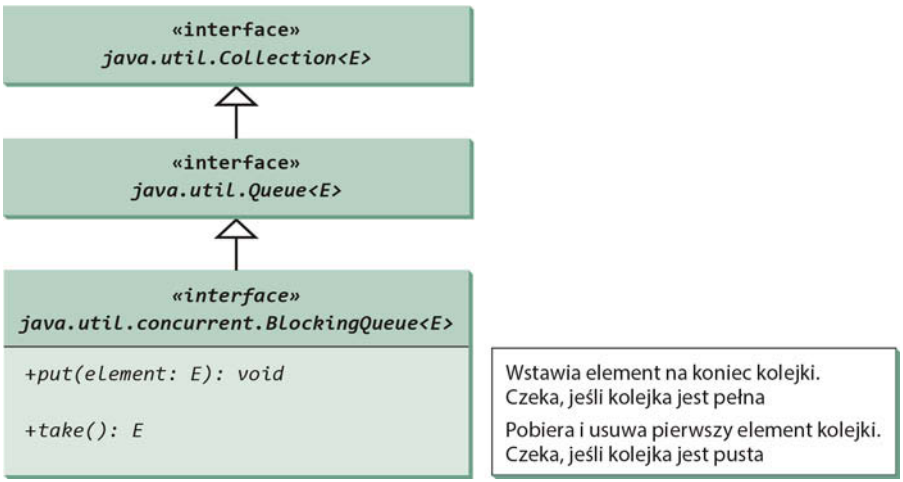
32.10.1. Czy metody `read` i `write` z klasy `Buffer` mogą być wykonywane współbieżnie?
32.10.2. Co się stanie, jeśli wywołasz metodę `read`, a kolejka będzie pusta?
32.10.3. Co się stanie, jeśli wywołasz metodę `write`, a kolejka będzie pełna?

32.11. Kolejki z blokowaniem

Java Collections Framework udostępnia klasy `ArrayBlockingQueue`, `LinkedBlockingQueue` i `PriorityBlockingQueue` reprezentujące kolejki z blokowaniem.

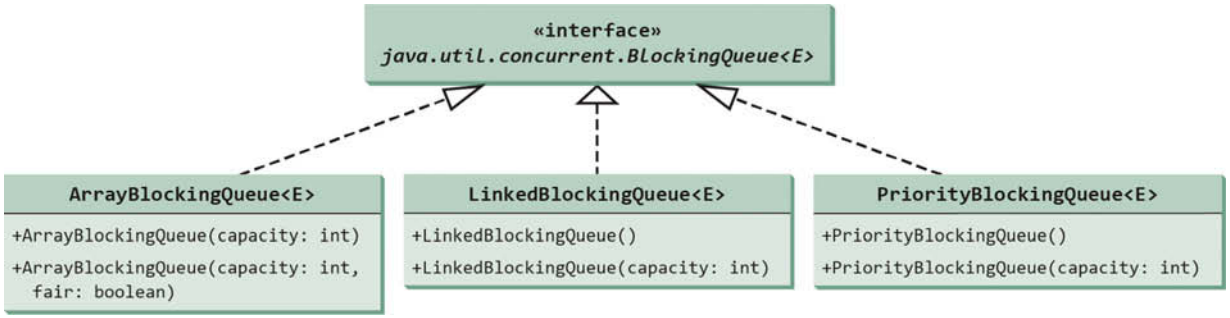
kolejka z blokowaniem

Kolejki i kolejki priorytetowe zostały opisane w podrozdziale 20.9. *Kolejki z blokowaniem* powodują zablokowanie wątku przy próbie dodania elementu do pełnej kolejki lub usunięcia elementu z pustej kolejki. Interfejs `Blocking➥Queue` rozszerza interfejs `java.util.Queue` i udostępnia zsynchronizowane metody `put` i `take`, które dodają element na koniec kolejki lub usuwają element z początku kolejki (rysunek 32.20).



Rysunek 32.20. BlockingQueue rozszerza interfejs Queue

Java udostępnia trzy konkretne klasy kolejek z blokowaniem: `ArrayBlockingQueue`, `LinkedBlockingQueue` i `Prio➥rityBlockingQueue` (rysunek 32.21). Wszystkie one znajdują się w pakiecie `java.util.concurrent`. Klasa `Array➥BlockingQueue` implementuje kolejkę za pomocą tablicy. Aby utworzyć obiekt tego typu, należy podać pojemność tablicy i opcjonalnie politykę uczciwości. Klasa `LinkedBlockingQueue` implementuje kolejkę za pomocą listy powiązanej z nieograniczoną lub ograniczoną liczbą elementów. Klasa `PriorityBlockingQueue` implementuje kolejkę za pomocą kolejki priorytetowej z nieograniczoną lub ograniczoną liczbą elementów.



Rysunek 32.21. ArrayBlockingQueue, LinkedBlockingQueue i PriorityBlockingQueue to konkretne klasy kolejek z blokowaniem

kolejki bez ograniczenia liczby elementów

Uwaga

W klasach `LinkedBlockingQueue` i `PriorityBlockingQueue` bez ograniczenia liczby elementów metoda `put` nigdy nie blokuje wątków.

Na listingu 32.8 pokazane jest, jak użyć klasy `ArrayBlockingQueue` do uproszczenia kodu wzorca producent/konsument z listingu 32.7. W wierszu 5. tworzona jest kolejka `ArrayBlockingQueue` do przechowywania liczb całkowitych. Wątek producenta umieszcza liczbę w tej kolejce (wiersz 22.), a wątek konsumenta pobiera liczbę z kolejki (wiersz 38.).

LISTING 32.8. ConsumerProducerUsingBlockingQueue.java

```
 1 import java.util.concurrent.*;
 2
 3 public class ConsumerProducerUsingBlockingQueue {
 4   private static ArrayBlockingQueue<Integer> buffer =
 5     new ArrayBlockingQueue<>(2);                                  tworzenie bufora
 6
 7   public static void main(String[] args) {
 8     // Tworzenie puli z dwoma wątkami
 9     ExecutorService executor = Executors.newFixedThreadPool(2);   tworzenie 2 wątków
10     executor.execute(new ProducerTask());
11     executor.execute(new ConsumerTask());
12     executor.shutdown();
13   }
14
15   // Zadanie dodające liczby całkowite do bufora
16   private static class ProducerTask implements Runnable {         zadanie producenta
17     public void run() {
18       try {
19         int i = 1;
20         while (true) {
21           System.out.println("Producent zapisuje " + i);
22           buffer.put(i++); // Dodawanie do bufora wartości (na przykład 1)   zapisywanie
23           // Usypianie wątku
24           Thread.sleep((int)(Math.random() * 10000));
25         }
26       }
27       catch (InterruptedException ex) {
28         ex.printStackTrace();
29       }
30     }
31   }
32
33   // Zadanie wczytujące i usuwające liczbę całkowitą z bufora
34   private static class ConsumerTask implements Runnable {         zadanie konsumenta
35     public void run() {
36       try {
37         while (true) {
38           System.out.println("\t\t\tKonsument wczytuje " + buffer.take());   wczytywanie
39           // Usypianie wątku
40           Thread.sleep((int)(Math.random() * 10000));
41         }
42       }
43       catch (InterruptedException ex) {
44         ex.printStackTrace();
45       }
46     }
47   }
48 }
```

Na listingu 32.7 używałeś blokad i warunków do synchronizowania wątków producenta i konsumenta. Nowy program nie używa blokad i warunków, ponieważ synchronizacja jest zaimplementowana w klasie `ArrayBlocking↪Queue`.

32.11.1. Czym jest kolejka z blokowaniem? Jakie kolejki tego rodzaju są dostępne w Javie?

32.11.2. Jaka metoda służy do dodawania elementów do kolejek `ArrayBlockingQueue`? Co się dzieje, gdy kolejka jest pełna?

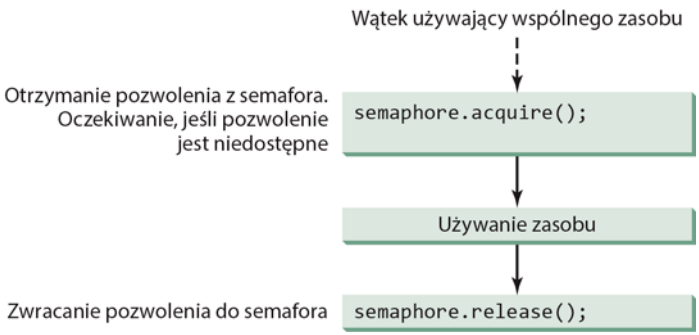
32.11.3. Jaka metoda służy do pobierania elementów z kolejek `ArrayBlockingQueue`? Co się dzieje, gdy kolejka jest pusta?

32.12. Semafory

Za pomocą semaforów możesz ograniczyć liczbę wątków mających dostęp do wspólnych zasobów.

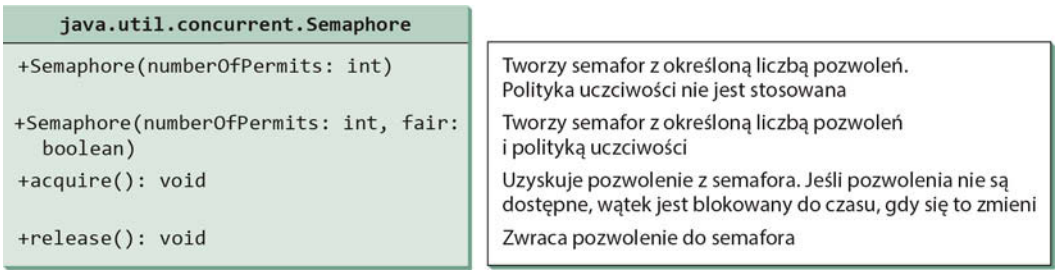
semafor

W informatyce *semafor* jest obiektem, który kontroluje dostęp do wspólnych zasobów. Przed dostępem do zasobu wątek musi otrzymać pozwolenie od semafora. Po zakończeniu korzystania z zasobu wątek musi zwrócić pozwolenie do semafora (rysunek 32.22).



Rysunek 32.22. Do wspólnego zasobu kontrolowanego przez semafor dostęp ma ograniczona liczba wątków

Aby utworzyć semafor, trzeba określić liczbę pozwoleń i opcjonalnie politykę uczciwości (rysunek 32.23). Zadanie może otrzymać pozwolenie, wywołując metodę `acquire()` semafora. Do zwracania pozwoleń służy metoda `release()` semafora. Po przyznaniu pozwolenia łączna liczba dostępnych pozwoleń w semaforze jest zmniejszana o 1. Zwrócenie pozwolenia skutkuje zwiększeniem liczby dostępnych pozwoleń o 1.



Rysunek 32.23. Klasa Semaphore zawiera metody dostępu do semafora

Semafor z jednym pozwoleniem pozwala zasymulować pracę blokady ze wzajemnym wykluczaniem. Na listingu 32.9 znajduje się zmodyfikowana klasa wewnętrzna `Account` z listingu 32.6. Nowa wersja używa semafora, aby zagwarantować, że w danym momencie do metody `deposit` dostęp ma tylko jeden wątek.

LISTING 32.9. Nowa klasa wewnętrzna Account

```
 1 // Klasa wewnętrzna Account
 2 private static class Account {
 3   // Tworzenie semafora
 4   private static Semaphore semaphore = new Semaphore(1);
 5   private int balance = 0;
 6
 7   public int getBalance() {
 8     return balance;
 9   }
10
11   public void deposit(int amount) {
12     try {
13       semaphore.acquire(); // Uzyskiwanie pozwolenia
14       int newBalance = balance + amount;
15
16       // Opóźnienie jest dodawane celowo, aby nasilić i uwidocznić
17       // problem uszkodzenia danych
18       Thread.sleep(5);
19
20       balance = newBalance;
21     }
22     catch (InterruptedException ex) {
23     }
24     finally {
25       semaphore.release(); // Zwracanie pozwolenia
26     }
27   }
28 }
```

tworzenie semafora

uzyskiwanie pozwolenia

zwracanie pozwolenia

W wierszu 4. tworzony jest semafor z jednym pozwoleniem. Wątek najpierw uzyskuje pozwolenie w trakcie wykonywania metody wpłacającej (wiersz 13.). Po zaktualizowaniu stanu konta wątek zwraca pozwolenie (wiersz 25.). Dobrym zwyczajem jest umieszczanie metody `release()` w klauzuli `finally`. Gwarantuje to, że pozwolenie zostanie zwrócone nawet po wystąpieniu wyjątku.

32.12.1. Opisz podobieństwa i różnice między blokadami i semaforami.

32.12.2. Jak utworzyć semafor umożliwiający jednoczesną pracę trzem wątkom? Jak uzyskać pozwolenie od semafora? Jak zwrócić pozwolenie?

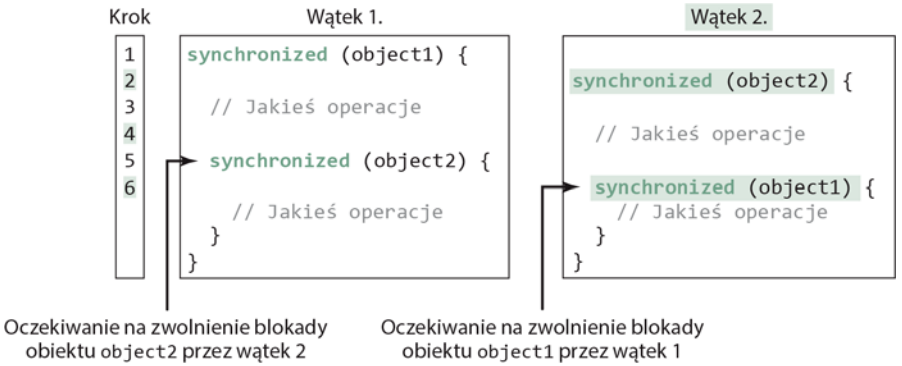
32.13. Unikanie zakleszczenia

Zakleszczenia można uniknąć za pomocą techniki cyklicznego oczekiwania na zasoby.

zakleszczenie

Czasem kilka wątków musi zająć blokady powiązane z kilkoma wspólnymi obiektami. Może to skutkować *zakleszczeniem*, co polega na tym, że każdy wątek zajmuje blokadę jednego obiektu i oczekuje na blokadę innego obiektu. Rozważ scenariusz z dwoma wątkami i dwoma obiektami (rysunek 32.24). Wątek 1 zajął blokadę obiektu `object1`, a wątek 2 zajął blokadę obiektu `object2`. Teraz wątek 1 oczekuje na blokadę obiektu `object2`, a wątek 2 oczekuje

na blokadę obiektu `object1`. Każdy wątek oczekuje na zwolnienie potrzebnej blokady przez drugi wątek; dopóki któraś z blokad nie zostanie zwolniona, żaden wątek nie będzie mógł wznowić pracy.



Rysunek 32.24. Zakleszczenie wątków 1 i 2

cykliczne oczekiwanie

Zakleszczenia można łatwo uniknąć za pomocą prostej techniki *cyklicznego oczekiwania na zasoby*. Polega to na określeniu i przestrzeganiu ustalonej kolejności zajmowania blokad obiektów. Przyjmij, że na rysunku 32.24 blokady są zajmowane w kolejności `object1`, `object2`. Wtedy wątek 2 musi najpierw zająć blokadę obiektu `object1`, a następnie obiektu `object2`. Gdy wątek 1 zajmie blokadę obiektu `object1`, wątek 2 będzie musiał czekać. Dzięki temu wątek 1 będzie mógł zająć także blokadę obiektu `object2` i zakleszczenie nie wystąpi.

32.13.1. Czym jest zakleszczenie? Jak można go uniknąć?

32.14. Stany wątków

Stan wątku określa status wątku.

Zadania są wykonywane w wątkach. Wątki przyjmują pięć stanów: nowy, gotowy, działający, zablokowany, ukończony (rysunek 32.25).

Po utworzeniu wątek przyjmuje stan *Nowy*. Po uruchomieniu za pomocą metody `start()` wątek przechodzi w stan *Gotowy*. Taki wątek jest gotowy do wykonywania, ale możliwe, że jeszcze nie działa. System operacyjny musi przydzielić mu czas procesora.

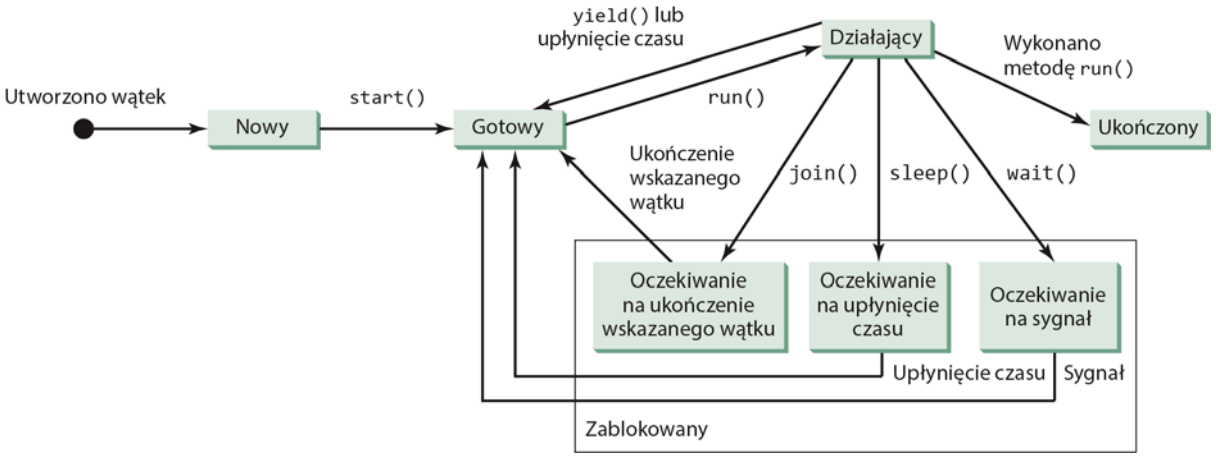
Gdy gotowy wątek rozpoczyna pracę, przechodzi w stan *Działający*. Działający wątek może przejść w stan *Gotowy*, jeśli upłynie przydzielony mu czas procesora lub wywołana zostanie metoda `yield()`.

Wątek z kilku przyczyn może przejść w stan *zablokowany* (czyli nieaktywny). Powodem może być na przykład wywołanie metody `join()`, `sleep()` lub `wait()` albo oczekiwanie na ukończenie operacji I/O. Zablokowany wątek może wznowić pracę po odwróceniu operacji, która doprowadziła do zablokowania go. Na przykład po upływie czasu uśpienia wątek ponownie staje się aktywny i przechodzi w stan *gotowy*.

Po zakończeniu wykonywania metody `run()` wątek przechodzi w stan *ukończony*.

Stan wątku można sprawdzić za pomocą metody `isAlive()`. Zwraca ona wartość `true`, jeśli wątek znajduje się w stanie **gotowy**, **zablokowany** lub **działający**. Wartość `false` oznacza, że wątek jest nowy i nie zaczął jeszcze pracy lub że wątek ukończył działanie.

Metoda `interrupt()` przerywa aktualne działania wątku. Jeśli wątek znajduje się w stanie **gotowy** lub **działający**, ustawiana jest flaga przerwania. Gdy wątek jest zablokowany, zostaje wybudzony, przechodzi w stan **gotowy** i zgłaszany jest wyjątek `java.lang.InterruptedException`.



RYSUNEK 32.25. Wątek przyjmuje pięć stanów: nowy, gotowy, działający, zablokowany, ukończony

32.14.1. Czym jest stan wątku? Opisz stany wątku.

32.15. Synchronizowane kolekcje

Java Collections Framework udostępnia synchronizowane listy, zbiory i odwzorowania.

Klasy z Java Collections Framework nie są bezpieczne ze względu na wątki. Ich zawartość może zostać uszkodzona, jeśli wiele wątków będzie z nich jednocześnie korzystać i aktualizować ich dane. Możesz chronić dane w kolekcji, blokując kolekcję lub używając kolekcji synchronizowanych.

kolekcje synchronizowane

Klasa `Collections` udostępnia sześć metod statycznych do opakowywania kolekcji w synchronizowane wersje (rysunek 32.26). Kolekcje tworzone za pomocą tych metod są *nakładkami synchronizującymi*.

nakładki synchronizujące

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Zwraca synchronizowaną kolekcję
+synchronizedList(list: List): List	Zwraca synchronizowaną listę na podstawie podanej listy
+synchronizedMap(m: Map): Map	Zwraca synchronizowane odwzorowanie na podstawie podanego odwzorowania
+synchronizedSet(s: Set): Set	Zwraca synchronizowany zbiór na podstawie podanego zbioru
+synchronizedSortedMap(s: SortedMap): SortedMap	Zwraca synchronizowane posortowane odwzorowanie na podstawie podanego posortowanego odwzorowania
+synchronizedSortedSet(s: SortedSet): SortedSet	Zwraca synchronizowany posortowany zbiór

RYSUNEK 32.26. Za pomocą klasy Collections możesz otrzymać kolekcje synchronizowane

Wywołanie `synchronizedCollection(Collection c)` zwraca nowy obiekt typu `Collection`, w którym wszystkie metody używające pierwotnej kolekcji `c` i modyfikujące ją są synchronizowane. Te metody są implementowane z wykorzystaniem słowa kluczowego `synchronized`. Na przykład metoda `add` jest zaimplementowana tak:

```
public boolean add(E o) {
  synchronized (this) {
    return c.add(o);
  }
}
```

Kolekcje synchronizowane mogą być bezpiecznie używane i modyfikowane przez wiele równoległych wątków.

Uwaga

Metody z klas `java.util.Vector`, `java.util.Stack` i `java.util.Hashtable` są synchronizowane. Są to dawne klasy wprowadzone już w JDK 1.0. Od JDK 1.5 należy używać klasy `java.util.ArrayList` zamiast `Vector`, `java.util.LinkedList` zamiast `Stack` i `java.util.Map` zamiast `Hashtable`. Jeśli potrzebna jest synchronizacja, użyj nakładki synchronizującej.

szybkie zgłaszanie niepowodzenia

Synchronizujące klasy nakładkowe są bezpieczne ze względu na wątki, a iterator ma *szybko zgłaszać niepowodzenie*. To oznacza, że jeśli używasz iteratora do poruszania się po kolekcji, a zostanie ona zmodyfikowana przez inny wątek, iterator natychmiast zgłosi problem, zgłaszając wyjątek `java.util.ConcurrentModificationException` (rozszerzający klasę `RuntimeException`). Aby uniknąć tego błędu, trzeba utworzyć obiekt synchronizowanej kolekcji i zajmować blokadę obiektu w trakcie poruszania się po kolekcji. Na przykład na potrzeby przetwarzania zbioru napisz następujący kod:

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Potrzebna synchronizacja
  Iterator iterator = hashSet.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Jeśli tego nie zrobisz, działanie kodu będzie nieprzewidywalne (może na przykład wystąpić wyjątek `ConcurrentModificationException`).

32.15.1. Czym jest synchronizowana kolekcja? Czy klasa `ArrayList` jest synchronizowana? Jak sprawić, aby była synchronizowana?

32.15.2. Wyjaśnij, dlaczego iterator szybko zgłasza błąd.

32.16. Programowanie równoległe

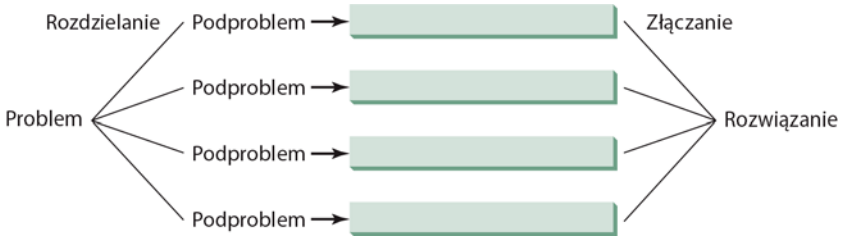
W programowaniu równoległym w Javie używany jest model fork/join.

W podrozdziale 7.12 opisane zostały metody `Arrays.sort` i `Arrays.parallelSort` służące do sortowania tablic. Metoda `parallelSort` wykorzystuje wiele procesorów, co skraca czas sortowania. W rozdziale 22. poznałeś równoległe strumienie i dowiedziałeś się, jak wykonywać na nich równoległe operacje, co przyspiesza pracę w środowiskach wieloprocesorowych. Przetwarzanie równoległe jest oparte na modelu fork/join. W tym podrozdziale poznasz ten model, co pozwoli Ci samodzielnie pisać kod równoległy.

model fork/join

Model fork/join (czyli rozdziel/złącz) jest pokazany na rysunku 32.27. Problem jest dzielony na niepokrywające się podproblemy, które można równolegle rozwiązywać niezależnie od pozostałych. Rozwiązania wszystkich podproblemów są następnie złączane, aby otrzymać ogólne rozwiązanie problemu. Jest to równoległa implementacja techniki dziel i rządź. W modelu fork/join w JDK 7 etap *rozdzielania* można traktować jak niezależne zadanie działające w wątku.

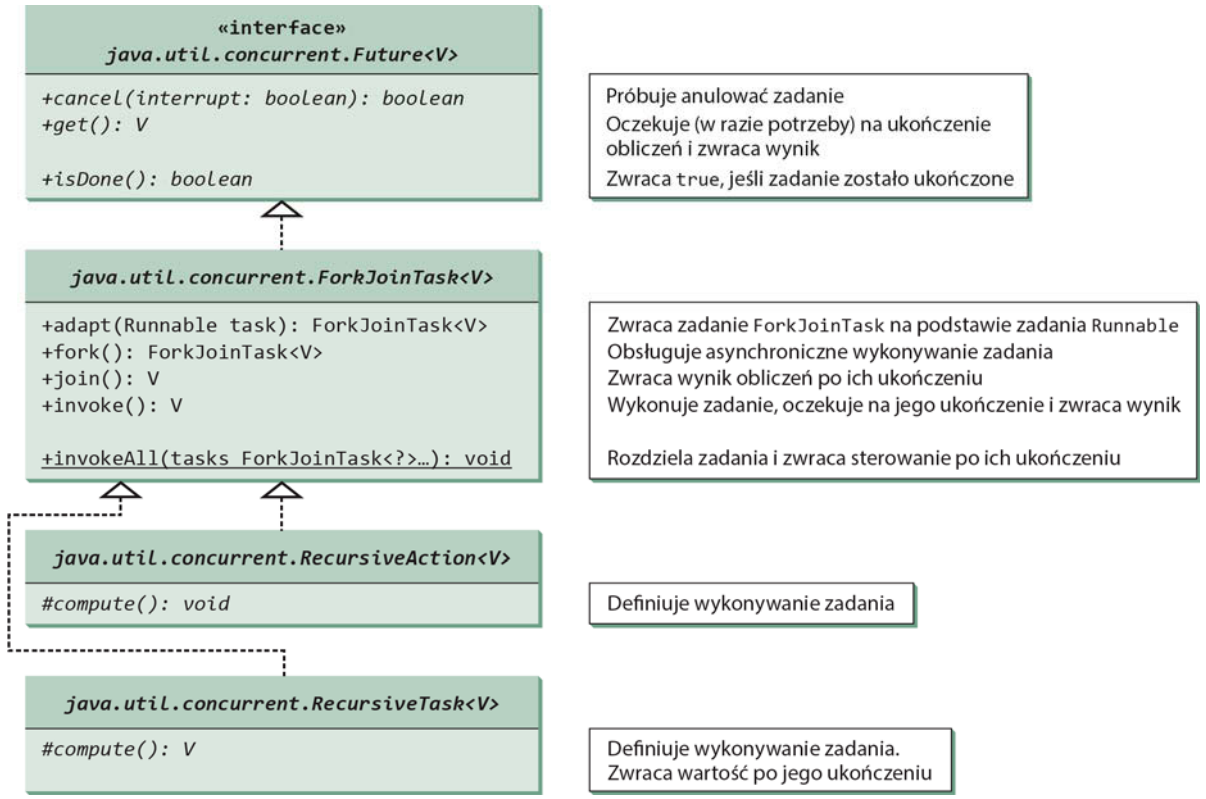
model w JDK 7



RYSUNEK 32.27. Niepokrywające się podproblemy są rozwiązywane równolegle

ForkJoinClass
ForkJoinPool

W tym modelu zadanie jest definiowane za pomocą klasy `ForkJoinClass` (rysunek 32.28). Zadania są wykonywane w instancji klasy `ForkJoinPool` (rysunek 32.29).



RYSUNEK 32.28. Klasa ForkJoinTask definiuje asynchronicznie wykonywane zadanie

`ForkJoinTask` jest abstrakcyjną klasą bazową do tworzenia zadań. Zadania tego typu są podobne do wątków, ale znacznie lżejsze, ponieważ można uruchomić bardzo dużą liczbę zadań i podzadań za pomocą niewielkiej liczby wątków z puli `ForkJoinPool`. Zadania są koordynowane głównie za pomocą wywołań `fork()` i `join()`. Wywołanie `fork()` obsługuje asynchroniczne wykonywanie zadania, a wywołanie `join()` powoduje oczekiwanie na ukończenie zadania. Metody `invoke()` i `invokeAll(tasks)` automatycznie wywołują metodę `fork()` w celu wykonania zadania

RYSUNEK 32.29. Klasa ForkJoinPool wykonuje zadania w modelu fork/join

oraz `join()` w celu oczekiwania na ukończenie zadań i zwrócenie wyników (jeśli kod zwraca wyniki). Zauważ, że statyczna metoda `invokeAll` przyjmuje zmienną liczbę argumentów typu `ForkJoinTask`; używana jest tu wprowadzona w podrozdziale 7.9 składnia `...`.

Model fork/join ma umożliwiać równoległe stosowanie techniki dziel i rządź, która z natury jest rekurencyjna. `RecursiveAction` i `RecursiveTask` są podklasami klasy `ForkJoinTask`. Aby zdefiniować konkretną klasę zadania, utwórz podklasę klasy `RecursiveAction` lub `RecursiveTask`. `RecursiveAction` służy do tworzenia zadań, które nie zwracają wartości. `RecursiveTask` tworzy zadania, które zwracają wartość. W klasie zadania należy przesłonić metodę `compute()`, aby określić sposób jego wykonywania.

RecursiveAction
RecursiveTask

Teraz na przykładzie sortowania przez scalanie zobaczysz, jak pisać programy równoległe w modelu fork/join. Algorytm sortowania przez scalanie (podrozdział 25.3) dzieli tablicę na dwie połowy i rekurencyjnie stosuje sortowanie przez scalanie do każdej z nich. Po posortowaniu obu połów algorytm scala je. Na listingu 32.10 pokazana jest równoległa implementacja tego algorytmu. Porównany został też czas wykonywania tego algorytmu i sortowania sekwencyjnego.

LISTING 32.10. ParallelMergeSort.java

```
 1 import java.util.concurrent.RecursiveAction;
 2 import java.util.concurrent.ForkJoinPool;
 3
 4 public class ParallelMergeSort {
 5   public static void main(String[] args) {
 6     final int SIZE = 7000000;
 7     int[] list1 = new int[SIZE];
 8     int[] list2 = new int[SIZE];
 9
10     for (int i = 0; i < list1.length; i++)
11       list1[i] = list2[i] = (int)(Math.random() * 10000000);
12
13     long startTime = System.currentTimeMillis();
14     parallelMergeSort(list1); // Wywołanie równoległego sortowania przez scalanie
15     long endTime = System.currentTimeMillis();
16     System.out.println("\nSortowanie równoległe z użyciem "
17       + Runtime.getRuntime().availableProcessors() +
18       " procesorów: " + (endTime - startTime) + " ms");
19
20     startTime = System.currentTimeMillis();
21     MergeSort.mergeSort(list2); // Klasa MergeSort z listingu 23.5
22     endTime = System.currentTimeMillis();
23     System.out.println("\nSortowanie sekwencyjne: " +
24       (endTime - startTime) + " ms");
```

wywołanie sortowania równoległego

wywołanie sortowania sekwencyjnego

```
25   }
26
27   public static void parallelMergeSort(int[] list) {
28     RecursiveAction mainTask = new SortTask(list);
29     ForkJoinPool pool = new ForkJoinPool();
30     pool.invoke(mainTask);
31   }
32
33   private static class SortTask extends RecursiveAction {
34     private final int THRESHOLD = 500;
35     private int[] list;
36
37     SortTask(int[] list) {
38       this.list = list;
39     }
40
41     @Override
42     protected void compute() {
43       if (list.length < THRESHOLD)
44         java.util.Arrays.sort(list);
45       else {
46         // Pobieranie najpierw pierwszej połowy
47         int[] firstHalf = new int[list.length / 2];
48         System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50         // Pobieranie drugiej połowy
51         int secondHalfLength = list.length - list.length / 2;
52         int[] secondHalf = new int[secondHalfLength];
53         System.arraycopy(list, list.length / 2,
54           secondHalf, 0, secondHalfLength);
55
56         // Rekurencyjne sortowanie obu połów
57         invokeAll(new SortTask(firstHalf),
58           new SortTask(secondHalf));
59
60         // Scalanie firstHalf z secondHalf na liście
61         MergeSort.merge(firstHalf, secondHalf, list);
62       }
63     }
64   }
65 }
```

tworzenie obiektu typu ForkJoinTask
tworzenie obiektu typu ForkJoinPool
wykonywanie zadania
definiowanie konkretnej klasy zadania
wykonywanie zadania
sortowanie krótkiej listy
podział na 2 części
sortowanie obu części
scalanie obu części

```
Sortowanie równoległe z użyciem 2 procesorów: 2829 ms
Sortowanie sekwencyjne: 4751 ms
```

Ponieważ ten algorytm sortowania nie zwraca wartości, definiowana jest klasa konkretna z rodziny `ForkJoinTask` rozszerzająca klasę `RecursiveAction` (wiersze 33. – 64.). Metoda `compute` jest przesłaniana, aby zaimplementować rekurencyjne sortowanie przez scalanie (wiersze 42. – 63.). Jeśli lista jest krótka, bardziej wydajne jest sortowanie sekwencyjne (wiersz 44.). Duże listy są dzielone na dwie połowy (wiersze 47. – 54.). Obie połowy są sortowane równolegle (wiersze 57. i 58.), a następnie scalane (wiersz 61.).

Ten program tworzy główne zadanie typu `ForkJoinTask` (wiersz 28.), pulę `ForkJoinPool` (wiersz 29.) i przekazuje główne zadanie do wykonania w tej puli (wiersz 30.). Metoda `invoke` zwraca sterowanie po zakończeniu wykonywania głównego zadania.

W trakcie wykonywania głównego zadania jest ono dzielone na podzadania, które zostają uruchomione za pomocą wywołania `invokeAll` (wiersze 57. i 58.). Metoda `invokeAll` zwraca sterowanie po ukończeniu wszystkich podzadań. Zauważ, że każde podzadanie jest dodatkowo rekurencyjnie dzielone na mniejsze zadania. Może to skutkować tworzeniem i uruchamianiem w puli dużej liczby podzadań. Model fork/join automatycznie i wydajnie wykonuje oraz koordynuje wszystkie zadania.

Klasa `MergeSort` jest zdefiniowana na listingu 23.5. Program wywołuje metodę `MergeSort.merge`, aby scalić dwie posortowane podlisty (wiersz 61.), i metodę `MergeSort.mergeSort` (wiersz 21.), by przeprowadzić sekwencyjne sortowanie przez scalanie. Widać tu, że sortowanie równoległe jest znacznie szybsze od sekwencyjnego.

Zauważ, że można równolegle wykonywać także pętlę inicjującą listę. Należy jednak unikać wywoływania tu metody `Math.random()`, ponieważ jest ona synchronizowana i nie można jej wykonywać równolegle (zobacz ćwiczenie 32.12). Metoda `parallelMergeSort` sortuje jedynie tablice wartości typu `int`, można jednak przekształcić ją w metodę generyczną (zobacz ćwiczenie 32.13).

Na ogólnym poziomie problem można rozwiązać równolegle wedle następującego szablonu:

```
jeśli (problem jest krótki)
  rozwiąż go sekwencyjnie;
w przeciwnym razie {
  podziel problem na niepokrywające się podproblemy;
  rozwiąż podproblemy równolegle;
  połącz wyniki z podproblemów, aby rozwiązać cały problem;
}
```

Na listingu 32.11 pokazana jest równoległa metoda do znajdowania maksimum na liście.

LISTING 32.11. ParallelMax.java

```
 1 import java.util.concurrent.*;
 2
 3 public class ParallelMax {
 4   public static void main(String[] args) {
 5     // Tworzenie listy
 6     final int N = 9000000;
 7     int[] list = new int[N];
 8     for (int i = 0; i < list.length; i++)
 9       list[i] = i;
10
11     long startTime = System.currentTimeMillis();
12     System.out.println("\nMaksimum: " + max(list));
13     long endTime = System.currentTimeMillis();
14     System.out.println("Liczba procesorów: " +
15       Runtime.getRuntime().availableProcessors());
16     System.out.println("Czas: " + (endTime - startTime)
17       + " ms");
18   }
19
20   public static int max(int[] list) {
21     RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
22     ForkJoinPool pool = new ForkJoinPool();
23     return pool.invoke(task);
24   }
25
```

wywołanie max

tworzenie zadania
tworzenie puli
wykonywanie zadania

definiowanie konkretnej klasy zadania

```
26   private static class MaxTask extends RecursiveTask<Integer> {
27     private final static int THRESHOLD = 1000;
28     private int[] list;
29     private int low;
30     private int high;
31
32     public MaxTask(int[] list, int low, int high) {
33       this.list = list;
34       this.low = low;
35       this.high = high;
36     }
37
38     @Override
```

wykonywanie zadania

```
39     public Integer compute() {
40       if (high - low < THRESHOLD) {
```

rozwiązywanie małego problemu

```
41         int max = list[0];
42         for (int i = low; i < high; i++)
43           if (list[i] > max)
44             max = list[i];
45         return new Integer(max);
46       }
47       else {
48         int mid = (low + high) / 2;
```

podział na 2 części

```
49         RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50         RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
```

podział zadania right

```
52         right.fork();
```

podział zadania left

```
53         left.fork();
```

złączanie zadań

```
54         return new Integer(Math.max(left.join().intValue(),
55           right.join().intValue()));
56       }
57     }
58   }
59 }
```

```
Maksimum: 8999999
Liczba procesorów: 2
Czas: 44 ms
```

Ponieważ ten algorytm zwraca liczbę całkowitą, klasa zadania rozszerza tu klasę `RecursiveTask<Integer>` (wiersze 26. – 58.). Metoda `compute` jest przesłonięta, aby zwracała maksymalny element z przedziału `list[low..high]` (wiersze 39. – 57.). Jeśli lista jest krótka, wydajniejsze jest znalezienie maksimum sekwencyjnie (wiersze 40. – 46.). Duże listy są dzielone na dwie połowy (wiersze 48. – 50.). Zadania `left` i `right` znajdują maksimum w lewej i prawej połowie. Wywołanie `fork()` powoduje wykonanie zadania (wiersze 52. i 53.). Metoda `join()` oczekuje na ukończenie zadania, a następnie zwraca wynik (wiersze 54. i 55.).

32.16.1. Jak zdefiniować podklasę klasy `ForkJoinTask`? Czym różni się klasa `RecursiveAction` od `RecursiveTask`?

32.16.2. Jak nakazać systemowi wykonanie zadania?

32.16.3. Jaka metoda służy do sprawdzania, czy zadanie zostało ukończone?

32.16.4. Jak utworzyć pulę typu `ForkJoinPool`? Jak umieścić zadanie w takiej puli?

NAJWAŻNIEJSZE POJĘCIA

warunek
zakleszczenie
szybkie zgłaszanie niepowodzenia
polityka uczciwości
model fork/join
blokada
monitor
wielowątkowość
sytuacja wyścigu
semafor
nakładka synchronizująca
blok synchronizowany
wątek
bezpieczny ze względu na wątki

PODSUMOWANIE ROZDZIAŁU

1. Każde zadanie jest instancją interfejsu `Runnable`. *Wątek* to obiekt ułatwiający wykonywanie zadań. Aby zdefiniować klasę zadania, zaimplementuj interfejs `Runnable`; następnie utwórz wątek, opakowując zadanie za pomocą konstruktora z klasy `Thread`.
2. Po utworzeniu obiektu wątku użyj metody `start()`, aby uruchomić wątek, i metody `sleep(long)`, by uśpić wątek, dzięki czemu inne wątki będą miały możliwość wykonania działań.
3. Obiekt wątku nigdy nie wywołuje bezpośrednio metody `run`. To maszyna JVM wywołuje ją, gdy nadejdzie czas na wykonanie wątku. W klasie trzeba przesłonić metodę `run`, aby poinformować system o tym, co wątek ma robić po uruchomieniu.
4. Aby zapobiec uszkodzeniu wspólnych zasobów przez wątki, użyj *synchronizowanych* metod lub bloków. *Synchronizowana metoda* przed rozpoczęciem pracy zajmuje *blokadę*. Metody instancji zajmują blokadę obiektu, dla którego zostały wywołane. Metody statyczne zajmują blokadę klasy.
5. Instrukcja synchronizowana umożliwia zajęcie blokady dowolnego obiektu (nie tylko *bieżącego* obiektu) w trakcie wykonywania bloku kodu w metodzie. Taki blok jest nazywany *synchronizowanym*.
6. Możesz użyć bezpośrednio dodawanych blokad i *warunków*, aby ułatwić komunikację między wątkami. Możesz też posługiwać się wbudowanymi monitorami obiektów.
7. Kolejki z blokowaniem (`ArrayBlockingQueue`, `LinkedBlockingQueue` i `PriorityBlockingQueue`) dostępne w Java Collections Framework zapewniają automatyczną synchronizację dostępu do kolejki.
8. Aby ograniczyć liczbę jednoczesnych dostępów do wspólnego zasobu, możesz użyć semaforów.
9. *Zakleszczenie* ma miejsce, gdy przynajmniej dwa wątki zajmują blokady dla różnych obiektów w taki sposób, że każdy z tych wątków zajmuje blokadę jednego obiektu i oczekuje na blokadę innego obiektu zajmowaną przez inny wątek. Aby uniknąć zakleszczenia, możesz wymagać *cyklicznego oczekiwania na zasoby*.
10. Model fork/join z JDK 7 został zaprojektowany do tworzenia równoległych programów. Możesz zdefiniować klasę rozszerzającą klasę `RecursiveAction` lub `RecursiveTask`, równolegle wykonywać zadania w puli typu `ForkJoinPool` i uzyskać ogólne rozwiązanie po ukończeniu wszystkich zadań.

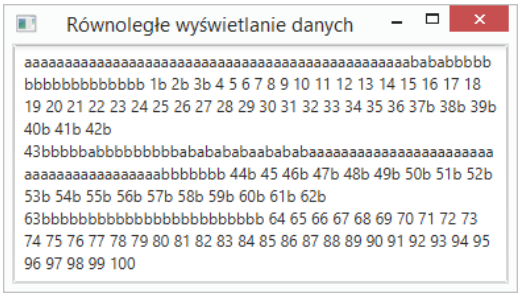
QUIZ

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

ĆWICZENIA PROGRAMISTYCZNE

Podrozdziały 32.1 – 32.5

***32.1.** *Modyfikacja listingu 32.1.* Zmodyfikuj listing 32.1, aby wyświetlał dane wyjściowe w obszarze tekstowym (rysunek 32.30).



RYSUNEK 32.30. Dane wyjściowe trzech wątków są wyświetlane w obszarze tekstowym

32.2. *Samochody wyścigowe.* Zmodyfikuj rozwiązanie ćwiczenia 15.29, używając wątku do sterowania samochodami. Porównaj nową wersję z rozwiązaniem ćwiczenia 15.29 przy ustawieniu w obu programach opóźnienia równego 10 ms. W którym programie animacja toczy się szybciej?

32.3. *Podnoszenie flagi.* Zmodyfikuj listing 15.13, używając wątku do pokazywania animacji podnoszenia flagi. Porównaj nowy program z listingiem 15.13, ustawiając w obu wersjach opóźnienie równe 10 ms. W którym programie animacja toczy się szybciej?

Podrozdziały 32.8 – 32.12

32.4. *Synchronizowanie wątków.* Napisz program, który uruchamia 1000 wątków. Każdy z nich ma dodawać 1 do zmiennej `sum` początkowo równej 0. Zmienną `sum` należy przekazywać do każdego wątku przez referencję. Aby to zrobić, zdefiniuj obiekt nakładkowy typu `Integer` przechowujący zmienną `sum`. Porównaj działanie programu z synchronizacją i bez niej.

32.5. *Wyświetlanie wentylatora.* Zmodyfikuj rozwiązanie ćwiczenia 15.28, używając wątku do sterowania animacją wentylatora.

32.6. *Odbijające się kulki.* Zmodyfikuj listing 15.17, *BallPane.java*, używając wątku do sterowania animacją ruchu kulek.

32.7. *Sterowanie zegarem.* Zmodyfikuj rozwiązanie ćwiczenia 15.32, używając wątku do sterowania animacją działania zegara.

32.8. *Synchronizacja konta.* Zmodyfikuj listing 32.6, *ThreadCooperation.java*, używając metod `wait()` i `notifyAll()` obiektu.

32.9. *Ilustrowanie działania wyjątku `ConcurrentModificationException`.* Iterator *szybko zgłasza niepowodzenie.* Napisz program, który to ilustruje. Utwórz dwa wątki, które równolegle używają zbioru i modyfikują go. Pierwszy wątek ma tworzyć zbiór typu `HashSet` zapełniony liczbami i co sekundę dodawać do zbioru nową wartość. Drugi wątek ma pobrać iterator tego zbioru i co sekundę poruszać się po zbiorze do przodu i wstecz. Program zgłosi wyjątek `ConcurrentModificationException`, ponieważ zbiór jest modyfikowany w pierwszym wątku w czasie, gdy drugi wątek porusza się po tym zbiorze.

***32.10.** *Używanie zbiorów synchronizowanych*. Posłuż się synchronizacją w celu rozwiązania problemu z poprzedniego ćwiczenia, tak by drugi wątek nie zgłaszał wyjątku `ConcurrentModificationException`.

Podrozdział 32.15

***32.11.** *Ilustrowanie zakleszczenia*. Napisz program ilustrujący zakleszczenie.

Podrozdział 32.18

***32.12.** *Równoległy inicjalizator tablic*. Użyj modelu fork/join do zaimplementowania metody przypisującej losowe wartości do listy:

```
public static void parallelAssignValues(double[] list)
```

Napisz program testowy, który tworzy listę 9 000 000 elementów i wywołuje metodę `parallelAssign`
`➥Values`, aby przypisać do tej listy losowe wartości. Zaimplementuj też algorytm sekwencyjny i porównaj czas wykonywania obu wersji kodu. Jeśli użyjesz metody `Math.random()`, czas wykonywania kodu równoległego będzie dłuższy od czasu pracy wersji sekwencyjnej, ponieważ metoda `Math.random()` jest synchronizowana i nie można jej wykonywać równolegle. Aby rozwiązać problem, utwórz obiekt typu `Random` do przypisywania losowych wartości do krótkich list.

32.13. *Generyczne równoległe sortowanie przez scalanie*. Zmodyfikuj listing 32.10, *ParallelMergeSort.java*, i zdefiniuj generyczną metodę `parallelMergeSort`:

```
public static <E extends Comparable<E>> void
  parallelMergeSort(E[] list)
```

***32.14.** *Równoległe sortowanie szybkie*. Zaimplementuj równoległą metodę wykonującą sortowanie szybkie listy (zobacz listing 23.7):

```
public static void parallelQuickSort(int[] list)
```

Napisz program testowy, który mierzy czas sortowania listy o długości 9 000 000 elementów za pomocą algorytmu równoległego i algorytmu sekwencyjnego.

***32.15.** *Równoległe sumowanie*. Użyj modelu fork/join do zaimplementowania metody obliczającej sumę elementów listy:

```
public static double parallelSum(double[] list)
```

Napisz program, który oblicza sumę 9 000 000 wartości typu `double`.

***32.16.** *Równoległe dodawanie macierzy*. W ćwiczeniu 8.5 opisane jest, jak dodawać macierze. Przyjmij, że używasz systemu wieloprocesorowego, co pozwala przyspieszyć dodawanie macierzy. Zaimplementuj równoległą metodę:

```
public static double[][] parallelAddMatrix(
  double[][] a, double[][] b)
```

Napisz program testowy, który mierzy czas dodawania dwóch macierzy o wymiarach 2000×2000 za pomocą algorytmu równoległego i algorytmu sekwencyjnego.

***32.17.** *Równoległe mnożenie macierzy*. W ćwiczeniu 7.6 opisane jest, jak mnożyć macierze. Przyjmij, że używasz systemu wieloprocesorowego, co pozwala przyspieszyć mnożenie macierzy. Zaimplementuj równoległą metodę:

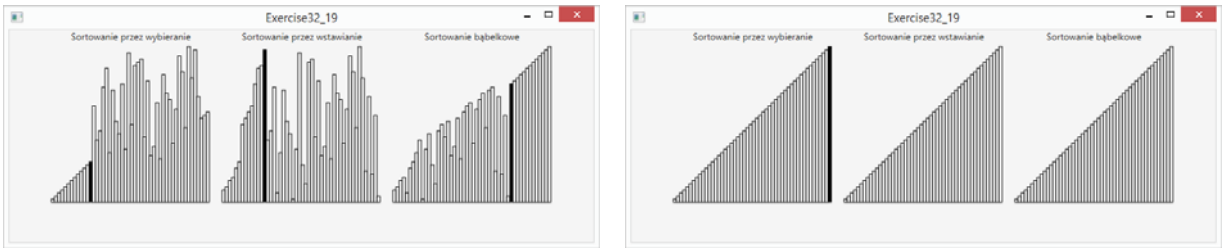
```
public static double[][] parallelMultiplyMatrix(
  double[][] a, double[][] b)
```

Napisz program testowy, który mierzy czas mnożenia dwóch macierzy o wymiarach 2000×2000 za pomocą algorytmu równoległego i algorytmu sekwencyjnego.

***32.18.** *Równoległe rozwiązanie problemu ośmiu hetmanów.* Zmodyfikuj listing 22.11, *EightQueens.java*; opracuj równoległy algorytm znajdujący wszystkie rozwiązania problemu ośmiu hetmanów. *Wskazówka*: uruchom osiem podzadań, z których każde umieszcza hetmana w innej kolumnie pierwszego wiersza.

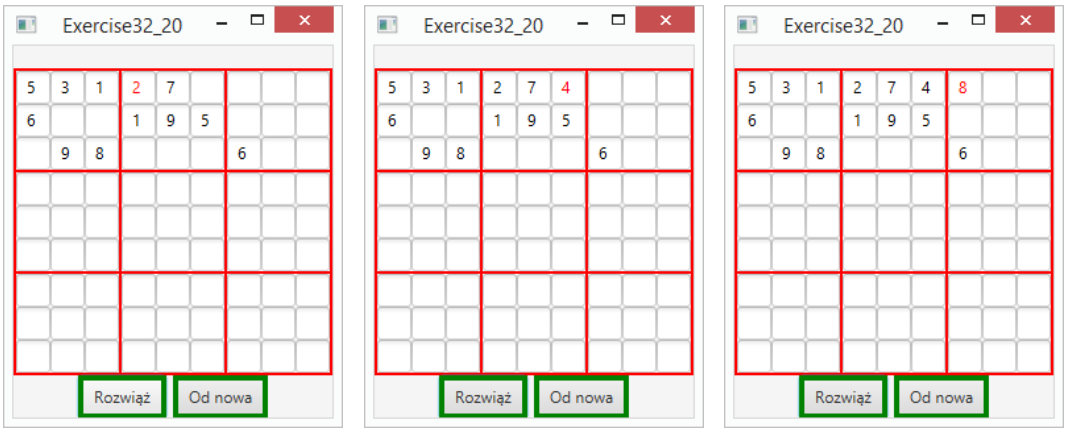
Ogólne

****32.19.** *Animacja sortowania.* Napisz kod animacji sortowania przez wybieranie, sortowania przez wstawianie i sortowania bąbelkowego (rysunek 32.31). Utwórz tablicę liczb całkowitych 1, 2, …, 50. Losowo potasuj te wartości. Utwórz panel do wyświetlania tablicy w formie histogramu. Każda metoda sortowania powinna działać w odrębnym wątku, a w każdym algorytmie należy użyć dwóch pętli zagnieżdżonych. Gdy algorytm zakończy iterację w pętli zewnętrznej, należy uśpić wątek na 0,5 sekundy i ponownie wyświetlić tablicę w formie histogramu. Wyróżnij kolorem ostatni słupek posortowanej podtablicy.



RYSUNEK 32.31. Animacja ilustruje trzy algorytmy sortowania

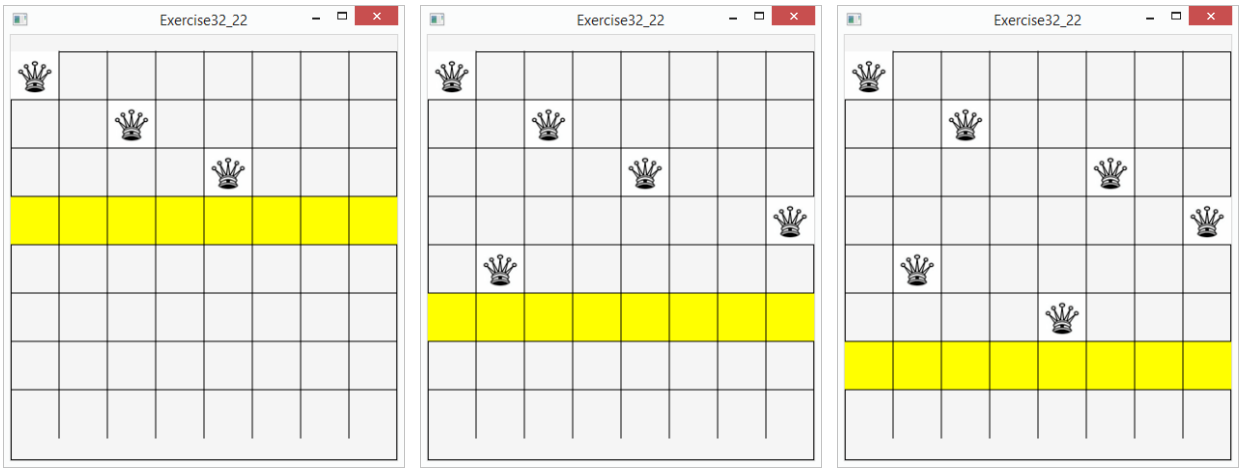
****32.20.** *Animacja wyszukiwania rozwiązania sudoku.* Zmodyfikuj rozwiązanie ćwiczenia 22.21, aby wyświetlać pośrednie wyniki wyszukiwania. Na rysunku 32.32 pokazany jest przebieg animacji z dodaną liczbą 2 (rysunek 32.32a), dodaną liczbą 4 (rysunek 32.32b) i dodaną liczbą 8 (rysunek 32.32c). Animacja wyświetla wszystkie kroki wyszukiwania.



RYSUNEK 32.32. Animacja wyświetla kroki wyszukiwania rozwiązania sudoku

32.21. *Łączenie zderzających się kulek*. Zmodyfikuj rozwiązanie ćwiczenia 20.5, używając wątku do animacji ruchu odbijających się kulek.

*****32.22.** *Animacja problemu ośmiu hetmanów*. Zmodyfikuj listing 22.11, *EightQueens.java*, aby wyświetlać pośrednie wyniki wyszukiwania. Na rysunku 32.33 pokazane jest, że wyróżniany jest sprawdzany wiersz. Co sekundę wyświetlany jest nowy stan planszy.



RYSUNEK 32.33. Animacja wyświetlająca kroki wyszukiwania rozwiązania problemu ośmiu hetmanów

ROZDZIAŁ

33

Sieci

Cele

- Objaśnienie pojęć TCP, IP, nazwa domeny, DNS, komunikacja strumieniowa i komunikacja pakietowa (podrozdział 33.2).
- Utworzenie serwerów z użyciem gniazd serwera (punkt 33.2.1) i klientów za pomocą gniazd klienta (punkt 33.2.2).
- Zaimplementowanie programów sieciowych w Javie z użyciem gniazd strumieni (punkt 33.2.3).
- Opracowanie przykładowej aplikacji typu klient-serwer (punkt 33.2.4)
- Pozyskanie adresów internetowych za pomocą klasy `InetAddress` (podrozdział 33.3).
- Opracowanie serwerów dla wielu klientów (podrozdział 33.4).
- Przesyłanie i pobieranie obiektów w sieci (podrozdział 33.5).
- Opracowanie interaktywnej gry w kółko i krzyżyk rozgrywanej w internecie (podrozdział 33.6).

33.1. Wprowadzenie

Sieci komputerowe służą do wysyłania i odbierania wiadomości w komputerach przez internet.

Aby możliwe było przeglądanie zasobów sieci WWW i wysyłanie e-maili, komputer musi być podłączony do internetu. *Internet* jest globalną siecią obejmującą miliony komputerów. Komputer może się połączyć z internetem za pomocą połączenia wdzwanianego od dostawcy ISP, modemu DSL, modemu kablowego lub sieci LAN.

adres IP

Gdy komputer komunikuje się z inną maszyną, musi znać jej adres. *Adres IP* (ang. *internet protocol*) w unikatowy sposób identyfikuje komputer w internecie. Taki adres składa się z czterech rozdzielonych kropkami liczb dziesiętnych z przedziału od 0 do 255 (na przykład 130.254.204.33). Ponieważ niełatwo jest zapamiętać tyle liczb, adresy IP często są odwzorowywane na bardziej zrozumiałe *nazwy domeny*, na przykład *liang.armstrong.edu*. Specjalne *serwery DNS* (ang. *domain name system*) w internecie tłumaczą nazwy hostów na adresy IP. Gdy komputer chce połączyć się z domeną *liang.armstrong.edu*, najpierw wysyła do serwera DNS prośbę o przetłumaczenie nazwy domeny na adres IP, a następnie zgłasza żądanie za pomocą tego adresu.

nazwa domeny
serwer DNS

TCP
UDP

IP jest niskopoziomowym protokołem do dostarczania danych w pakietach z jednego komputera do innego w internecie. Razem z IP używane są dwa protokoły wyższego poziomu: *TCP* (ang. *transmission control protocol*) i *UDP* (ang. *user datagram protocol*). TCP umożliwia dwóm hostom nawiązanie połączenia i wymianę strumieni danych. Ten protokół gwarantuje dostarczenie danych, a także zachowanie kolejności wysyłanych pakietów. UDP to standardowy, niskokosztowy, bezstanowy protokół typu host-host korzystający z protokołu IP. UDP umożliwia aplikacji z jednego komputera przekazywanie datagramów do aplikacji z innej maszyny.

komunikacja strumieniowa
komunikacja pakietowa

Java obsługuje komunikację strumieniową i pakietową. W *komunikacji strumieniowej* do transmisji danych używany jest protokół TCP. W *komunikacji pakietowej* wykorzystywany jest protokół UDP. Ponieważ TCP potrafi wykrywać utratę danych i ponownie je przesyłać, zapewnia bezstratną i niezawodną transmisję danych. Protokół UDP tego nie gwarantuje. W Javie w większości zastosowań używana jest komunikacja strumieniowa i to ona jest głównym tematem tego rozdziału. Komunikacja pakietowa jest omówiona w suplemencie III.P, *Networking Using Datagram Protocol*, dostępnym w witrynie poświęconej oryginalnemu wydaniu książki.

33.2. Model klient-serwer

Java udostępnia klasę `ServerSocket` do tworzenia gniazd serwera i klasę `Socket` do tworzenia gniazd klienta. Programy w internecie mogą się komunikować, używając gniazda serwera, gniazda klienta i strumienia I/O.

gniazda

Obsługa sieci jest ściśle zintegrowana z Javą. API Javy udostępnia klasy do tworzenia gniazd ułatwiających programom komunikowanie się przez internet. *Gniazda* to punkty końcowe logicznych połączeń między hostami i mogą służyć do wysyłania oraz przyjmowania danych. Java traktuje komunikację z użyciem gniazd podobnie jak operacje I/O. Dlatego programy mogą wczytywać i zapisywać dane w gniazdach równie łatwo jak w plikach.

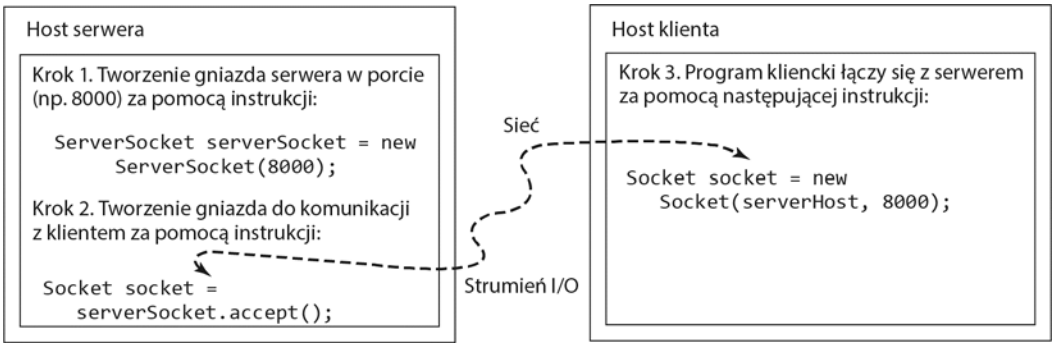
Programowanie aplikacji sieciowych zwykle wymaga utworzenia serwera i klientów. Klient przesyła żądanie na serwer, a serwer odpowiada. Klient najpierw próbuje nawiązać połączenie z serwerem. Serwer może zaakceptować je lub odrzucić. Po nawiązaniu połączenia klient i serwer komunikują się z użyciem gniazd.

Gdy klient próbuje połączyć się z serwerem, serwer musi być włączony. Serwer oczekuje na żądania od klientów. Instrukcje potrzebne do tworzenia gniazd na serwerze i w kliencie są pokazane na rysunku 33.1.

33.2.1. Gniazda serwera

gniazdo serwera
port

Aby przygotować serwer, trzeba utworzyć *gniazdo serwera* i powiązać je z *portem*, gdzie serwer będzie oczekiwał na połączenia. Port identyfikuje usługę TCP dostępną w gnieździe. Porty mogą mieć numery od 0 do 65 536, przy czym numery od 0 do 1024 są zarezerwowane dla specjalnych usług.



RYSUNEK 33.1. Serwer tworzy gniazdo serwera i po nawiązaniu połączenia z klientem komunikuje się z gniazdem klienta

Serwer e-mail działa w porcie 25, a serwer WWW zwykle działa w porcie 80. Możesz wybrać dowolny numer portu, który nie jest aktualnie używany przez inne programy. Ta instrukcja tworzy gniazdo serwera `serverSocket`:

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

Uwaga

BindException

Próba utworzenia gniazda serwera w już używanym porcie spowoduje wyjątek `java.net.BindException`.

33.2.2. Gniazda klienta

Po utworzeniu gniazda serwera serwer może użyć następującej instrukcji do oczekiwania na połączenia:

```
Socket socket = serverSocket.accept();
```

łączenie się z klientem

Powoduje ona oczekiwanie na połączenie się klienta z gniazdem serwera. Klient zgłasza następującą instrukcję, aby zażądać połączenia z serwerem:

```
Socket socket = new Socket(serverName, port);
```

gniazdo klienta
używanie adresu IP

Ta instrukcja otwiera gniazdo, przez które program kliencki może komunikować się z serwerem. Tu `serverName` to nazwa hosta lub adres IP serwera w internecie. W poniższej instrukcji na maszynie klienckiej tworzone jest gniazdo łączące się z hostem 130.254.204.33 w porcie 8000:

```
Socket socket = new Socket("130.254.204.33", 8000);
```

używanie nazwy domeny

Do utworzenia gniazda możesz też użyć nazwy domeny:

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

Gdy tworzysz gniazdo, podając nazwę hosta, maszyna JVM kieruje na serwer DNS żądanie przetłumaczenia tej nazwy na adres IP.

Uwaga

localhost

W programie można używać nazwy hosta `localhost` lub adresu IP 127.0.0.1, aby wskazać maszynę, na której ten klient działa.

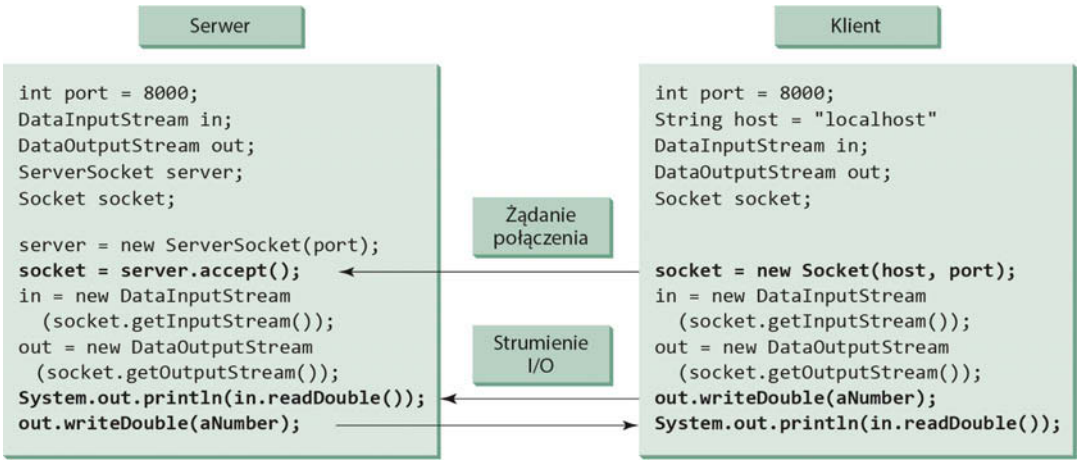
UnknownHostException

Uwaga

Konstruktor `Socket` zgłasza wyjątek `java.net.UnknownHostException`, jeśli nie można znaleźć hosta.

33.2.3. Transfer danych za pomocą gniazd

Po zaakceptowaniu połączenia przez serwer komunikacja między serwerem i klientem odbywa się tak samo jak przy użyciu strumieni I/O. Instrukcje potrzebne do utworzenia strumieni i wymiany danych są pokazane na rysunku 33.2.



Rysunek 33.2. Serwer i klient przekazują dane za pomocą strumieni I/O powiązanych z gniazdami

Aby uzyskać strumień wejściowy i wyjściowy, użyj metod `getInputStream()` i `getOutputStream()` obiektu gniazda. Poniższe instrukcje tworzą na podstawie gniazda strumień `input` typu `InputStream` i strumień `output` typu `OutputStream`:

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

Strumienie typów `InputStream` i `OutputStream` służą do odczytu i zapisu bajtów. Możesz też użyć klas `Data`
`InputStream`, `DataOutputStream`, `BufferedReader` i `PrintWriter` do opakowania strumieni `InputStream` i `Output`
`Stream` na potrzeby odczytu i zapisu danych typów `int`, `double` lub `String`. Poniższe instrukcje tworzą strumień `input` typu `DataInputStream` i strumień `output` typu `DataOutputStream` do odczytu i zapisu wartości typów podstawowych:

```
DataInputStream input = new DataInputStream
  (socket.getInputStream());
DataOutputStream output = new DataOutputStream
  (socket.getOutputStream());
```

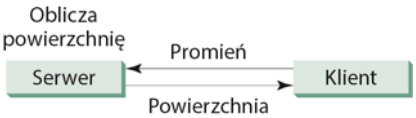
Serwer może używać metody `input.readDouble()` do pobierania wartości typu `double` od klienta i metody `output.writeDouble(d)` do przekazywania do klienta wartości `d` typu `double`.

Wskazówka

Pamiętaj, że binarne operacje I/O są wydajniejsze od tekstowych operacji I/O, ponieważ operacje tekstowe wymagają kodowania i dekodowania danych. Dlatego aby poprawić wydajność, lepiej jest używać binarnych operacji I/O do przekazywania danych między serwerem i klientem.

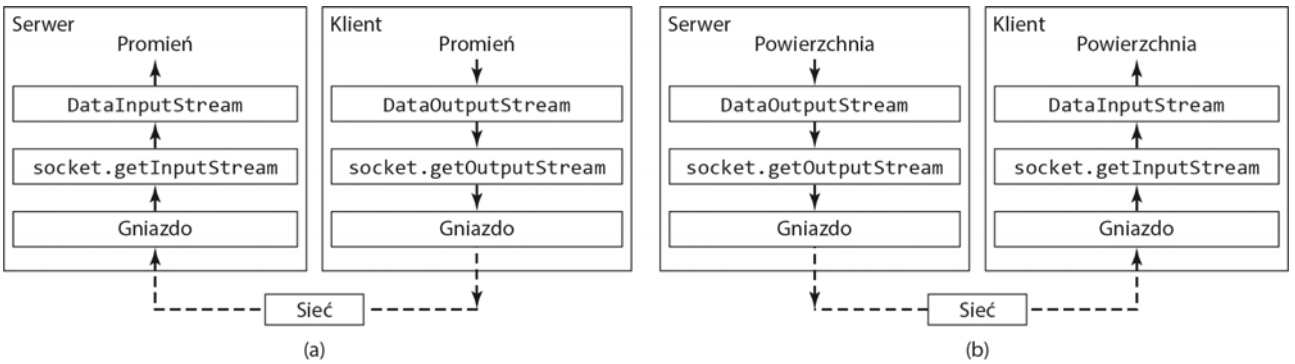
33.2.4. Przykładowa aplikacja typu klient-serwer

Teraz przyjrzysz się przykładowemu programowi klienckiemu i serwerowi. Klient przesyła dane na serwer. Serwer przyjmuje dane, używa ich do wygenerowania wyniku, a następnie odsyła wynik klientowi. Klient wyświetla wynik w konsoli. W tym przykładzie dane przesyłane od klienta to promień koła, a wynik generowany przez serwer to powierzchnia tego koła (rysunek 33.3).



RYSUNEK 33.3. Klient przesyła promień na serwer, który oblicza powierzchnię tej figury i zwraca ją klientowi

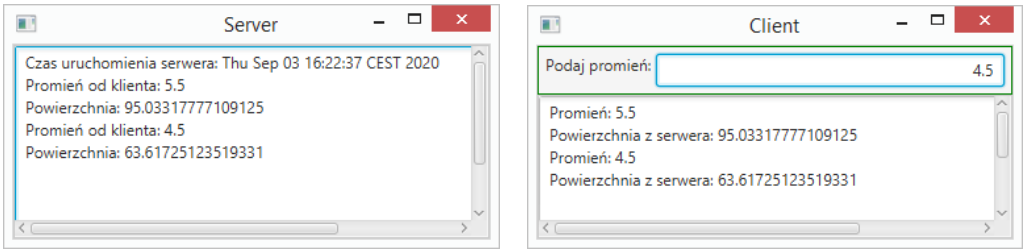
Klient przesyła promień za pomocą strumienia `DataOutputStream` przez gniazdo wyjściowe, a serwer przyjmuje te dane za pomocą strumienia `DataInputStream` przez gniazdo wejściowe (rysunek 33.4a). Serwer oblicza powierzchnię, po czym przesyła ją do klienta za pomocą strumienia `DataOutputStream` przez gniazdo wyjściowe, a klient odbiera dane za pomocą strumienia `DataInputStream` przez gniazdo wejściowe (rysunek 33.4b). Kod serwera i klienta są przedstawione na listingach 33.1 i 33.2. Rysunek 33.5 pokazuje przykładowy przebieg działania serwera i klienta.



RYSUNEK 33.4. (a) Klient przesyła promień na serwer; (b) Serwer przesyła powierzchnię do klienta

LISTING 33.1. Server.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.Date;
4 import javafx.application.Application;
5 import javafx.application.Platform;
6 import javafx.scene.Scene;
7 import javafx.scene.control.ScrollPane;
8 import javafx.scene.control.TextArea;
```



RYSUNEK 33.5. Klient przesyła promień na serwer. Serwer przyjmuje promień, oblicza powierzchnię i przesyła powierzchnię do klienta

```
 9 import javafx.stage.Stage;
10
11 public class Server extends Application {
12   @Override // Przesłanianie metody start z klasy Application
13   public void start(Stage primaryStage) {
14     // Obszar tekstowy do wyświetlania informacji
15     TextArea ta = new TextArea();
16
17     // Tworzenie sceny i umieszczanie jej w oknie
18     Scene scene = new Scene(new ScrollPane(ta), 450, 200);
19     primaryStage.setTitle("Server"); // Ustawianie nagłówka okna
20     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
21     primaryStage.show(); // Wyświetlanie okna
22
23     new Thread(() -> {
24       try {
25         // Tworzenie gniazda serwera
26         ServerSocket serverSocket = new ServerSocket(8000);
27         Platform.runLater(() ->
28           ta.appendText("Czas uruchomienia serwera: " + new Date() + '\n'));
29
30         // Oczekiwanie na żądanie nawiązania połączenia
31         Socket socket = serverSocket.accept();
32
33         // Tworzenie strumieni wejściowego i wyjściowego
34         DataInputStream inputFromClient = new DataInputStream(
35           socket.getInputStream());
36         DataOutputStream outputToClient = new DataOutputStream(
37           socket.getOutputStream());
38
39         while (true) {
40           // Przyjmowanie promienia od klienta
41           double radius = inputFromClient.readDouble();
42
43           // Obliczanie powierzchni
44           double area = radius * radius * Math.PI;
45
46           // Przesyłanie powierzchni do klienta
47           outputToClient.writeDouble(area);
48
49           Platform.runLater(() -> {
```

tworzenie interfejsu użytkownika serwera

gniazdo serwera

aktualizowanie interfejsu użytkownika

łączenie się z klientem

dane wejściowe od klienta

dane wyjściowe dla klienta

wczytywanie promienia

przesyłanie powierzchni

aktualizowanie interfejsu użytkownika

```
50             ta.appendText("Promień od klienta: "
51               + radius + '\n');
52             ta.appendText("Powierzchnia: " + area + '\n');
53           });
54         }
55       }
56       catch(IOException ex) {
57         ex.printStackTrace();
58       }
59     }).start();
60   }
61 }
```

LISTING 33.2. Client.java

```
 1 import java.io.*;
 2 import java.net.*;
 3 import javafx.application.Application;
 4 import javafx.geometry.Insets;
 5 import javafx.geometry.Pos;
 6 import javafx.scene.Scene;
 7 import javafx.scene.control.Label;
 8 import javafx.scene.control.ScrollPane;
 9 import javafx.scene.control.TextArea;
10 import javafx.scene.control.TextField;
11 import javafx.scene.layout.BorderPane;
12 import javafx.stage.Stage;
13
14 public class Client extends Application {
15   // Strumienie wejściowy i wyjściowy
16   DataOutputStream toServer = null;
17   DataInputStream fromServer = null;
18
19   @Override // Przesłanianie metody start z klasy Application
20   public void start(Stage primaryStage) {
21     // Panel p przechowuje etykietę i pole tekstowe
22     BorderPane paneForTextField = new BorderPane();
23     paneForTextField.setPadding(new Insets(5, 5, 5, 5));
24     paneForTextField.setStyle("-fx-border-color: green");
25     paneForTextField.setLeft(new Label("Podaj promień: "));
26
27     TextField tf = new TextField();
28     tf.setAlignment(Pos.BOTTOM_RIGHT);
29     paneForTextField.setCenter(tf);
30
31     BorderPane mainPane = new BorderPane();
32     // Obszar tekstowy do wyświetlania danych
33     TextArea ta = new TextArea();
34     mainPane.setCenter(new ScrollPane(ta));
35     mainPane.setTop(paneForTextField);
36
37     // Tworzenie sceny i umieszczanie jej w oknie
38     Scene scene = new Scene(mainPane, 450, 200);
39     primaryStage.setTitle("Client"); // Ustawianie nagłówka okna
40     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
41     primaryStage.show(); // Wyświetlanie okna
```

tworzenie interfejsu użytkownika

obsługa zdarzenia

wczytywanie promienia

zapis promienia

wczytywanie powierzchni

żądanie nawiązania połączenia

dane wejściowe z serwera

dane wyjściowe dla serwera

```
42
43      tf.setOnAction(e -> {
44        try {
45          // Pobieranie promienia z pola tekstowego
46          double radius = Double.parseDouble(tf.getText().trim());
47
48          // Przesyłanie promienia na serwer
49          toServer.writeDouble(radius);
50          toServer.flush();
51
52          // Pobieranie powierzchni od serwera
53          double area = fromServer.readDouble();
54
55          // Wyświetlanie danych w obszarze tekstowym
56          ta.appendText("Promień: " + radius + "\n");
57          ta.appendText("Powierzchnia z serwera: "
58            + area + '\n');
59        }
60        catch (IOException ex) {
61          System.err.println(ex);
62        }
63      });
64
65      try {
66        // Tworzenie gniazda na potrzeby połączenia z serwerem
67        Socket socket = new Socket("localhost", 8000);
68        // Socket socket = new Socket("130.254.204.36", 8000);
69        // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70
71        // Tworzenie strumienia wejściowego na potrzeby pobierania danych z serwera
72        fromServer = new DataInputStream(socket.getInputStream());
73
74        // Tworzenie strumienia wyjściowego na potrzeby przesyłania danych na serwer
75        toServer = new DataOutputStream(socket.getOutputStream());
76      }
77      catch (IOException ex) {
78        ta.appendText(ex.toString() + '\n');
79      }
80    }
81  }
```

Najpierw uruchom serwer, a następnie program kliencki. W programie klienckim wpisz promień w polu tekstowym, a następnie wciśnij klawisz *Enter*, by przesłać promień na serwer. Serwer obliczy powierzchnię i odeśle ją klientowi. Ten proces jest powtarzany do czasu zakończenia pracy przez jeden z programów.

Klasy do obsługi komunikacji w sieci znajdują się w pakiecie `java.net`. Gdy piszesz w Javie programy korzystające z sieci, powinieneś zaimportować ten pakiet.

Klasa `Server` tworzy gniazdo `serverSocket` typu `ServerSocket` i łączy je z portem 8000 (wiersz 26. z pliku *Server.java*):

```
ServerSocket serverSocket = new ServerSocket(8000);
```

Następnie serwer rozpoczyna oczekiwanie na żądania nawiązania połączenia (wiersz 31. z pliku *Server.java*):

```
Socket socket = serverSocket.accept();
```

Serwer oczekuje do czasu zażądania nawiązania połączenia przez klienta. Po nawiązaniu połączenia serwer wczytuje promień od klienta za pomocą strumienia wejściowego, oblicza powierzchnię i przesyła wynik do klienta za pomocą strumienia wyjściowego. Działanie metody `accept()` z klasy `ServerSocket` wymaga czasu. Niewłaściwe jest uruchamianie tej metody w wątku aplikacji opartej na JavaFX. Dlatego ta metoda działa w odrębnym wątku (wiersze 23. – 59.). Instrukcje aktualizujące GUI należy uruchamiać w wątku aplikacji opartej na JavaFX, używając metody `Platform.runLater` (27. – 28. i 49. – 53.).

Klasa `Client` używa pokazanej niżej instrukcji (wiersz 67. z pliku *Client.java*), aby utworzyć gniazdo, które żąda nawiązania połączenia z serwerem z tej samej maszyny (`localhost`) w porcie 8000:

```
Socket socket = new Socket("localhost", 8000);
```

Jeśli uruchamiasz serwer i klienta na różnych maszynach, zastąp nazwę `localhost` nazwą lub adresem IP hosta. W tym przykładzie serwer i klient działają na tej samej maszynie.

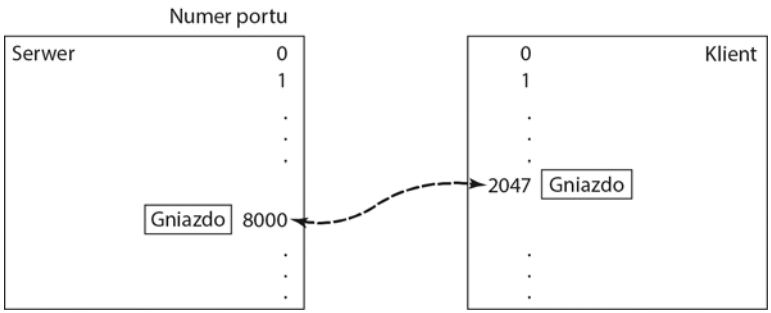
Jeśli serwer nie działa, program kliencki zakończy pracę, zgłaszając wyjątek `java.net.ConnectException`. Po nawiązaniu połączenia klient ma dostęp do strumieni wejściowego i wyjściowego (opakowanych w strumienie `DataInputStream` i `DataOutputStream`), co pozwala przesyłać dane na serwer i odbierać od niego odpowiedzi.

Jeśli po uruchomieniu serwera wystąpi wyjątek `java.net.BindException`, oznacza to, że port serwera jest aktualnie używany. Trzeba wtedy zatrzymać proces, używając potrzebnego portu, i ponownie uruchomić serwer.

Uwaga

Gdy tworzysz gniazdo serwera, musisz podać port (na przykład 8000). Gdy klient nawiązuje połączenie z serwerem (wiersz 67. z pliku *Client.java*), po stronie klienta tworzone jest gniazdo używające własnego lokalnego portu. Ten port (na przykład 2047) jest automatycznie wybierany przez maszynę JVM, co ilustruje rysunek 33.6.

port gniazda klienta



Rysunek 33.6. Maszyna JVM automatycznie wybiera dostępny port, aby utworzyć gniazdo klienta

Aby wyświetlić numer lokalnego portu klienta, wstaw poniższą instrukcję w wierszu 70. w pliku *Client.java*:

```
System.out.println("Lokalny port: " + socket.getLocalPort());
```

33.2.1. Jak utworzyć gniazdo serwera? Jakich numerów portów można używać? Co się stanie, jeśli żądany numer portu jest już zajęty? Czy port pozwala łączyć się z wieloma klientami?

33.2.2. Czym różni się gniazdo serwera od gniazda klienta?

33.2.3. W jaki sposób program kliencki inicjuje połączenie?

33.2.4. W jaki sposób serwer akceptuje połączenie?

33.2.5. W jaki sposób przekazywane są dane między klientem a serwerem?

33.3. Klasa InetAddress

Na serwerze można użyć klasy `InetAddress`, aby uzyskać informacje o adresie IP i nazwie hosta klienta.

Czasem przydatna jest informacja, kto łączy się z serwerem. Aby sprawdzić nazwę hosta i adres IP klienta, możesz użyć klasy `InetAddress`. Jest to klasa modelująca adres IP. Za pomocą pokazanej niżej instrukcji możesz w serwerze utworzyć obiekt typu `InetAddress` na podstawie gniazda łączącego się z klientem:

```
InetAddress inetAddress = socket.getInetAddress();
```

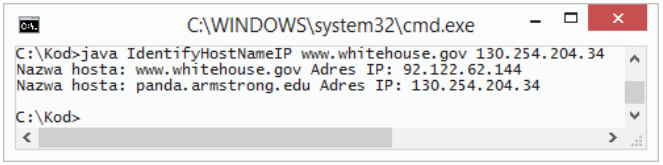
Następnie można wyświetlić nazwę hosta i adres IP klienta:

```
System.out.println("Nazwa hosta klienta: " +
  inetAddress.getHostName());

System.out.println("Adres IP klienta: "+
  inetAddress.getHostAddress());
```

Obiekt typu `InetAddress` można też utworzyć na podstawie nazwy hosta lub adresu IP, używając statycznej metody `getByName`. Poniższa instrukcja tworzy taki obiekt dla hosta `liang.armstrong.edu`:

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

Na listingu 33.3 pokazany jest program, który identyfikuje nazwę hosta i adres IP na podstawie argumentów podanych w wierszu poleceń. W wierszu 7. za pomocą metody `getByName` tworzony jest obiekt typu `InetAddress`. W wierszach 8. i 9. metody `getHostName` i `getHostAddress` pobierają nazwę i adres IP hosta. Rysunek 33.7 przedstawia przykładowy przebieg programu.



RYSUNEK 33.7. Program podaje nazwę i adres IP hosta

LISTING 33.3. IdentifyHostNameIP.java

```
 1 import java.net.*;
 2
 3 public class IdentifyHostNameIP {
 4   public static void main(String[] args) {
 5     for (int i = 0; i < args.length; i++) {
 6       try {
 7         InetAddress address = InetAddress.getByName(args[i]);
 8         System.out.print("Nazwa hosta: " + address.getHostName() + " ");
 9         System.out.println("Adres IP: " + address.getHostAddress());
10       }
11       catch (UnknownHostException ex) {
12         System.err.println("Nieznany host lub adres IP: " + args[i]);
13       }
14     }
15   }
16 }
```

tworzenie obiektu typu InetAddress
pobieranie nazwy hosta
pobieranie adresu IP hosta

33.3.1. Jak otrzymać instancję typu `InetAddress`?

33.3.2. Za pomocą jakich metod można pobrać adres IP i nazwę hosta z obiektu typu `InetAddress`?

33.4. Obsługa wielu klientów

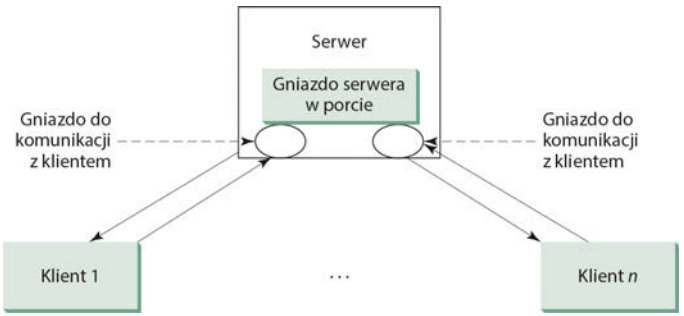
Serwer może obsługiwać wielu klientów. Połączenie z każdym klientem jest obsługiwane za pomocą osobnego wątku.

Często z jednym serwerem w tym samym momencie chce połączyć się wielu klientów. Zwykle serwer jest włączony cały czas na komputerze serwerowym, a klienty z różnych lokalizacji w internecie mogą się z nim łączyć. Możesz użyć wątków do jednoczesnej obsługi wielu klientów serwera — wystarczy utworzyć wątek dla każdego połączenia. Nawiązywanie połączenia na serwerze wygląda wtedy tak:

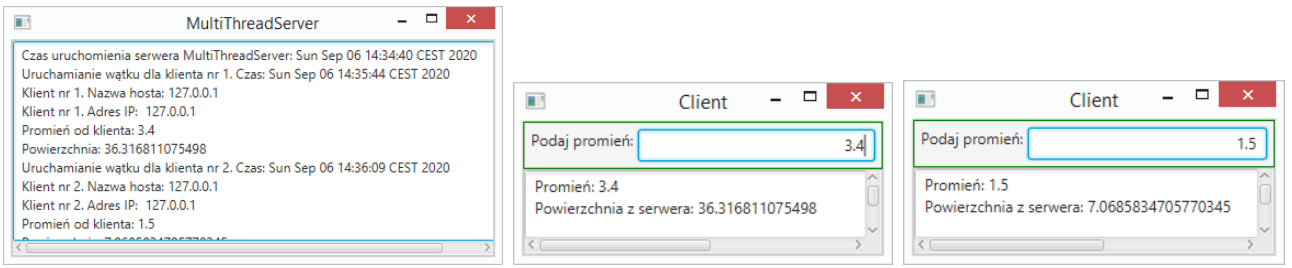
```
while (true) {
  Socket socket = serverSocket.accept(); // Łączenie się z klientem
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

Gniazdo serwera może obsługiwać wiele połączeń. Każda iteracja pętli `while` tworzy nowe połączenie. Gdy połączenie jest nawiązywane, tworzony jest nowy wątek do obsługi komunikacji między serwerem a nowym klientem. Dzięki temu można jednocześnie utrzymywać wiele połączeń.

Kod z listingu 33.4 tworzy klasę serwera, która jednocześnie obsługuje wielu klientów. Serwer dla każdego połączenia tworzy wątek, który cały czas pobiera od klientów dane wejściowe (promień koła) i odsyła wyniki (powierzchnię koła), co ilustruje rysunek 33.8. Nadal używany jest program kliencki z listingu 33.2. Przykładowy przebieg pracy serwera z dwoma klientami jest pokazany na rysunku 33.9.



RYSUNEK 33.8. Wielowątkowość umożliwia serwerowi obsługę wielu niezależnych klientów



RYSUNEK 33.9. Serwer tworzy wątek na potrzeby obsługi klienta

LISTING 33.4. MultiThreadServer.java

```
 1 import java.io.*;
 2 import java.net.*;
 3 import java.util.Date;
 4 import javafx.application.Application;
 5 import javafx.application.Platform;
 6 import javafx.scene.Scene;
 7 import javafx.scene.control.ScrollPane;
 8 import javafx.scene.control.TextArea;
 9 import javafx.stage.Stage;
10
11 public class MultiThreadServer extends Application {
12   // Obszar tekstowy do wyświetlania zawartości
13   private TextArea ta = new TextArea();
14
15   // Numerowanie klientów
16   private int clientNo = 0;
17
18   @Override // Przesłanianie metody start w klasie Application
19   public void start(Stage primaryStage) {
20     // Tworzenie sceny i umieszczanie jej w oknie
21     Scene scene = new Scene(new ScrollPane(ta), 450, 200);
22     primaryStage.setTitle("MultiThreadServer"); // Ustawianie nagłówka okna
23     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
24     primaryStage.show(); // Wyświetlanie okna
25
26     new Thread( () -> {
27       try {
28         // Tworzenie gniazda serwera
29         ServerSocket serverSocket = new ServerSocket(8000);
30         ta.appendText("Czas uruchomienia serwera MultiThreadServer: "
31           + new Date() + '\n');
32
33         while (true) {
34           // Oczekiwanie na żądanie nawiązania nowego połączenia
35           Socket socket = serverSocket.accept();
36
37           // Zwiększanie wartości zmiennej clientNo
38           clientNo++;
39
40           Platform.runLater( () -> {
41             // Wyświetlanie numeru klienta
42             ta.appendText("Uruchamianie wątku dla klienta nr " + clientNo +
43               ". Czas: " + new Date() + '\n');
44
45             // Pobieranie nazwy i adresu IP hosta klienta
46             InetAddress inetAddress = socket.getInetAddress();
47             ta.appendText("Klient nr " + clientNo + ". Nazwa hosta: "
48               + inetAddress.getHostName() + "\n");
49             ta.appendText("Klient nr " + clientNo + ". Adres IP: "
50               + inetAddress.getHostAddress() + "\n");
51           });
52
53           // Tworzenie i uruchamianie nowego wątku na potrzeby połączenia
```

gniazdo serwera

łączenie się z klientem

aktualizowanie GUI

informacje o kliencie

tworzenie wątku

uruchamianie wątku

gniazdo połączenia

aktualizowanie GUI

```
54            new Thread(new HandleAClient(socket)).start();
55          }
56        }
57        catch(IOException ex) {
58          System.err.println(ex);
59        }
60      }).start();
61    }
62
63    // Definiowanie klasy wątku do obsługi nowego połączenia
64    class HandleAClient implements Runnable {
65      private Socket socket; // Gniazdo połączenia
66
67      /** Tworzenie wątku */
68      public HandleAClient(Socket socket) {
69        this.socket = socket;
70      }
71
72      /** Uruchamianie wątku */
73      public void run() {
74        try {
75          // Tworzenie strumieni wejściowego i wyjściowego
76          DataInputStream inputFromClient = new DataInputStream(
77            socket.getInputStream());
78          DataOutputStream outputToClient = new DataOutputStream(
79            socket.getOutputStream());
80
81          // Stała obsługa klienta
82          while (true) {
83            // Pobieranie promienia od klienta
84            double radius = inputFromClient.readDouble();
85
86            // Obliczanie powierzchni
87            double area = radius * radius * Math.PI;
88
89            // Odsyłanie powierzchni do klienta
90            outputToClient.writeDouble(area);
91
92            Platform.runLater(() -> {
93              ta.appendText("Promień od klienta: " +
94                radius + '\n');
95              ta.appendText("Powierzchnia: " + area + '\n');
96            });
97          }
98        }
99        catch(IOException ex) {
100         ex.printStackTrace();
101       }
102     }
103   }
104 }
```

Ten serwer tworzy gniazdo w porcie 8000 (wiersz 29.) i oczekuje na połączenie (wiersz 35.). Po nawiązaniu połączenia z klientem serwer tworzy nowy wątek do obsługi komunikacji (wiersz 54.). Następnie oczekuje na kolejne połączenie w nieskończonej pętli `while` (wiersze 33. – 55.).

Wątki działają niezależnie od siebie i komunikują się z wyznaczonymi klientami. Każdy wątek tworzy strumienie wejściowy i wyjściowy, które przyjmują dane i odsyłają wyniki klientom.

33.4.1. Jak sprawić, by serwer obsługiwał wielu klientów?

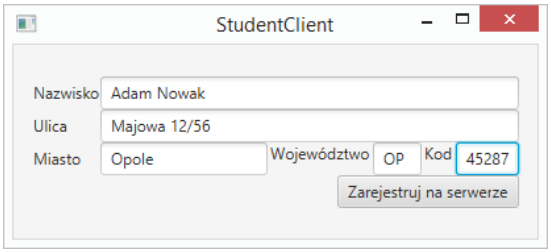
33.5. Wysyłanie i przyjmowanie obiektów

Program może wysyłać obiekty i przyjmować je od innych aplikacji.

W poprzednich przykładach nauczyłeś się wysyłać i odbierać dane typów podstawowych. Można też wysyłać i przyjmować obiekty za pomocą strumieni typów `ObjectOutputStream` i `ObjectInputStream`. Przekazywane obiekty muszą umożliwiać serializację. Następny przykład ilustruje, jak wysyłać i przyjmować obiekty.

Używane są tu trzy klasy: *StudentAddress.java* (listing 33.5), *StudentClient.java* (listing 33.6), *StudentServer.java* (listing 33.7). Program kliencki pobiera informacje o studencie i przekazuje je na serwer (rysunek 33.10).

Klasa `StudentAddress` zawiera informacje o studencie: imię i nazwisko, ulicę, miasto i województwo oraz kod pocztowy. Ta klasa implementuje interfejs `Serializable`, dlatego możliwe jest wysyłanie i przyjmowanie obiektów tego typu za pomocą wyjściowego i wejściowego strumienia obiektów.



RYSUNEK 33.10. Klient wysyła w obiekcie informacje o studencie na serwer

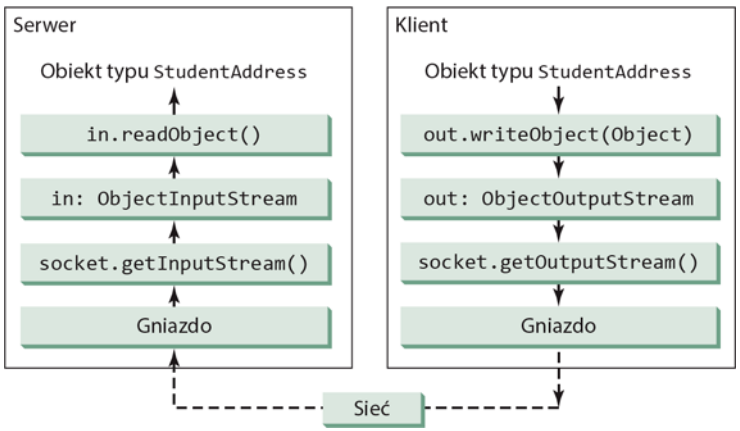
LISTING 33.5. StudentAddress.java

umożliwianie serializacji

```
1 public class StudentAddress implements java.io.Serializable {
2   private String name;
3   private String street;
4   private String city;
5   private String state;
6   private String zip;
7
8   public StudentAddress(String name, String street, String city,
9     String state, String zip) {
10    this.name = name;
11    this.street = street;
12    this.city = city;
13    this.state = state;
14    this.zip = zip;
15  }
16
17  public String getName() {
18    return name;
19  }
20
```

```
21   public String getStreet() {
22     return street;
23   }
24
25   public String getCity() {
26     return city;
27   }
28
29   public String getState() {
30     return state;
31   }
32
33   public String getZip() {
34     return zip;
35   }
36 }
```

Klient wysyła obiekt typu `StudentAddress` za pomocą strumienia `ObjectOutputStream` przez gniazdo wyjściowe. Serwer przyjmuje obiekt z danymi studenta za pomocą strumienia `ObjectInputStream` przez gniazdo wejściowe (rysunek 33.11). Klient używa metody `writeObject` z klasy `ObjectOutputStream`, aby przesłać dane studenta na serwer. Serwer przyjmuje informacje o studencie za pomocą metody `readObject` z klasy `ObjectInputStream`. Serwer i program kliencki są przedstawione na listingach 33.6 i 33.7.



RYSUNEK 33.11. Klient przesyła na serwer obiekt typu StudentAddress

LISTING 33.6. StudentClient.java

```
 1 import java.io.*;
 2 import java.net.*;
 3 import javafx.application.Application;
 4 import javafx.event.ActionEvent;
 5 import javafx.event.EventHandler;
 6 import javafx.geometry.HPos;
 7 import javafx.geometry.Pos;
 8 import javafx.scene.Scene;
 9 import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TextField;
```

```
12 import javafx.scene.layout.GridPane;
13 import javafx.scene.layout.HBox;
14 import javafx.stage.Stage;
15
16 public class StudentClient extends Application {
17   private TextField tfName = new TextField();
18   private TextField tfStreet = new TextField();
19   private TextField tfCity = new TextField();
20   private TextField tfState = new TextField();
21   private TextField tfZip = new TextField();
22
23   // Przycisk do przesyłania danych studenta na serwer
24   private Button btRegister = new Button("Zarejestruj na serwerze");
25
26   // Nazwa lub adres IP hosta
27   String host = "localhost";
28
29   @Override // Przesłanianie metody start z klasy Application
30   public void start(Stage primaryStage) {
31     GridPane pane = new GridPane();
32     pane.add(new Label("Nazwisko"), 0, 0);
33     pane.add(tfName, 1, 0);
34     pane.add(new Label("Ulica"), 0, 1);
35     pane.add(tfStreet, 1, 1);
36     pane.add(new Label("Miasto"), 0, 2);
37
38     HBox hBox = new HBox(2);
39     pane.add(hBox, 1, 2);
40     hBox.getChildren().addAll(tfCity, new Label("Województwo"), tfState,
41       new Label("Kod"), tfZip);
42     pane.add(btRegister, 1, 3);
43     GridPane.setHalignment(btRegister, HPos.RIGHT);
44
45     pane.setAlignment(Pos.CENTER);
46     tfName.setPrefColumnCount(15);
47     tfStreet.setPrefColumnCount(15);
48     tfCity.setPrefColumnCount(10);
49     tfState.setPrefColumnCount(2);
50     tfZip.setPrefColumnCount(3);
51
52     btRegister.setOnAction(new ButtonListener());
53
54     // Tworzenie sceny i umieszczanie jej w oknie
55     Scene scene = new Scene(pane, 450, 200);
56     primaryStage.setTitle("StudentClient"); // Ustawianie nagłówka okna
57     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
58     primaryStage.show(); // Wyświetlanie okna
59   }
60
61   /** Obsługa zdarzeń */
62   private class ButtonListener implements EventHandler<ActionEvent> {
63     @Override
64     public void handle(ActionEvent e) {
65       try {
66         // Nawiązywanie połączenia z serwerem
```

tworzenie interfejsu użytkownika

rejestrowanie odbiornika

gniazdo serwera

strumień wyjściowy

wysyłanie na serwer

```
67          Socket socket = new Socket(host, 8000);
68
69          // Tworzenie strumienia wyjściowego z danymi dla serwera
70          ObjectOutputStream toServer =
71            new ObjectOutputStream(socket.getOutputStream());
72
73          // Pobieranie zawartości pól tekstowych
74          String name = tfName.getText().trim();
75          String street = tfStreet.getText().trim();
76          String city = tfCity.getText().trim();
77          String state = tfState.getText().trim();
78          String zip = tfZip.getText().trim();
79
80          // Tworzenie obiektu typu StudentAddress i przesyłanie go na serwer
81          StudentAddress s =
82            new StudentAddress(name, street, city, state, zip);
83          toServer.writeObject(s);
84        }
85        catch (IOException ex) {
86          ex.printStackTrace();
87        }
88      }
89    }
90 }
```

LISTING 33.7. StudentServer.java

```
 1 import java.io.*;
 2 import java.net.*;
 3
 4 public class StudentServer {
 5   private ObjectOutputStream outputToFile;
 6   private ObjectInputStream inputFromClient;
 7
 8   public static void main(String[] args) {
 9     new StudentServer();
10   }
11
12   public StudentServer() {
13     try {
14       // Tworzenie gniazda serwera
15       ServerSocket serverSocket = new ServerSocket(8000);
16       System.out.println("Serwer został uruchomiony ");
17
18       // Tworzenie strumienia wyjściowego obiektów
19       outputToFile = new ObjectOutputStream(
20         new FileOutputStream("student.dat", true));
21
22       while (true) {
23         // Oczekiwanie na żądanie nawiązania połączenia
24         Socket socket = serverSocket.accept();
25
26         // Tworzenie strumienia wejściowego z użyciem gniazda
27         inputFromClient =
28           new ObjectInputStream(socket.getInputStream());
29
```

gniazdo serwera

zapis danych w pliku

łączenie się z klientem

strumień wejściowy

pobieranie obiektu od klienta

zapis w pliku

```
30          // Odczyt ze strumienia wejściowego
31          Object object = inputFromClient.readObject();
32
33          // Zapis w pliku
34          outputToFile.writeObject(object);
35          System.out.println("Zapisano nowy obiekt z danymi studenta");
36        }
37      }
38      catch(ClassNotFoundException ex) {
39        ex.printStackTrace();
40      }
41      catch(IOException ex) {
42        ex.printStackTrace();
43      }
44      finally {
45        try {
46          inputFromClient.close();
47          outputToFile.close();
48        }
49        catch(Exception ex) {
50          ex.printStackTrace();
51        }
52      }
53    }
54 }
```

Gdy po stronie klienta użytkownik kliknie przycisk *Zarejestruj na serwerze*, klient utworzy gniazdo na potrzeby połączenia z hostem (wiersz 67.), utworzy obiekt typu `ObjectOutputStream` na podstawie gniazda (wiersze 70. i 71.) i wywoła metodę `writeObject`, by przesłać obiekt typu `StudentAddress` na serwer za pomocą strumienia wyjściowego obiektów (wiersz 83.).

Gdy klient nawiąże połączenie z serwerem, serwer utworzy obiekt typu `ObjectInputStream` na podstawie gniazda (wiersze 27. i 28.), wywoła metodę `readObject`, aby pobrać obiekt typu `StudentAddress` przez strumień wejściowy obiektów (wiersz 31.), i zapisze ten obiekt w pliku (wiersz 34.).

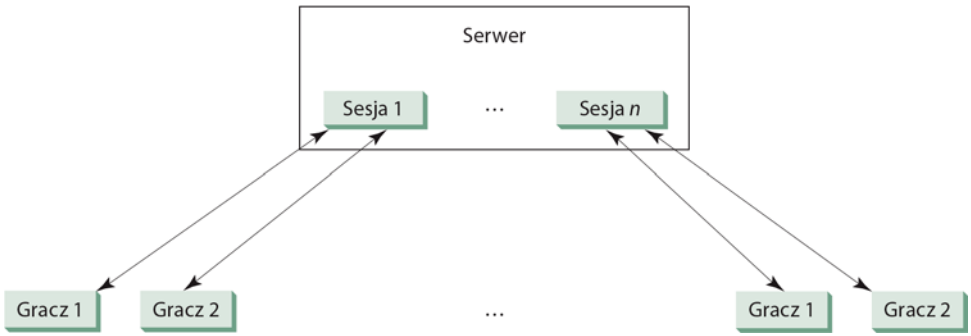
33.5.1. W jaki sposób serwer przyjmuje połączenie od klienta? Jak klient łączy się z serwerem?
33.5.2. Jak na serwerze sprawdzić nazwę hosta programu klienckiego?
33.5.3. Jak przesyłać i przyjmować obiekty?

33.6. Studium przypadku: kółko i krzyżyk w środowisku rozproszonym

W tym podrozdziale napiszesz program umożliwiający dwóm osobom grę w kółko i krzyżyk przez internet.

W podrozdziale 16.12, „Studium przypadku: tworzenie gry w kółko i krzyżyk”, napisałeś program do gry w kółko i krzyżyk dla dwóch osób korzystających z tej samej maszyny. W tym podrozdziale dowiesz się, jak napisać podobną grę działającą w środowisku rozproszonym. Użyjesz wielu wątków i sieci ze strumieniami opartymi na gniazdach. Program będzie umożliwiał rozgrywkę użytkownikom korzystającym z dowolnych maszyn podłączonych do internetu.

Potrzebny będzie serwer dla wielu klientów. Serwer ten będzie tworzyć gniazda i przyjmować połączenia od dwóch graczy, aby rozpocząć sesję. Każda sesja ma działać w wątku komunikującym się z dwoma graczami i określającym stan gry. Serwer może utworzyć dowolną liczbę sesji (rysunek 33.12).



RYSUNEK 33.12. Serwer może utworzyć wiele sesji, z których każda umożliwia grę w kółko i krzyżyk dwóm osobom

W każdej sesji pierwszy klient łączący się z serwerem jest uznawany za gracza 1 posługującego się symbolem X, a drugi klient — za gracza 2 używającego symbolu O. Serwer powiadamia graczy o używanych symbolach. Po połączeniu się z klientami serwer uruchamia wątek obsługujący grę i wielokrotnie wykonujący odpowiednie kroki (rysunek 33.13).

Serwer nie udostępnia interfejsu graficznego, ale utworzenie GUI z informacjami o grze byłoby pomocne dla użytkowników. Możesz utworzyć w GUI panel przewijania z obszarem tekstowym i wyświetlać w tym obszarze informacje o grze. Po połączeniu się dwóch graczy z serwerem ten tworzy wątek do obsługi sesji.

Klient odpowiada za komunikację z graczami. Wyświetla interfejs z dziewięcioma komórkami oraz nagłówek i stan w etykietach. Klasa klienta bardzo przypomina klasę `TicTacToe` ze studium przypadku z listingu 16.13. Jednak tu klient nie określa stanu gry (wygranej lub remisu), a jedynie przekazuje ruchy na serwer i pobiera z niego stan gry.

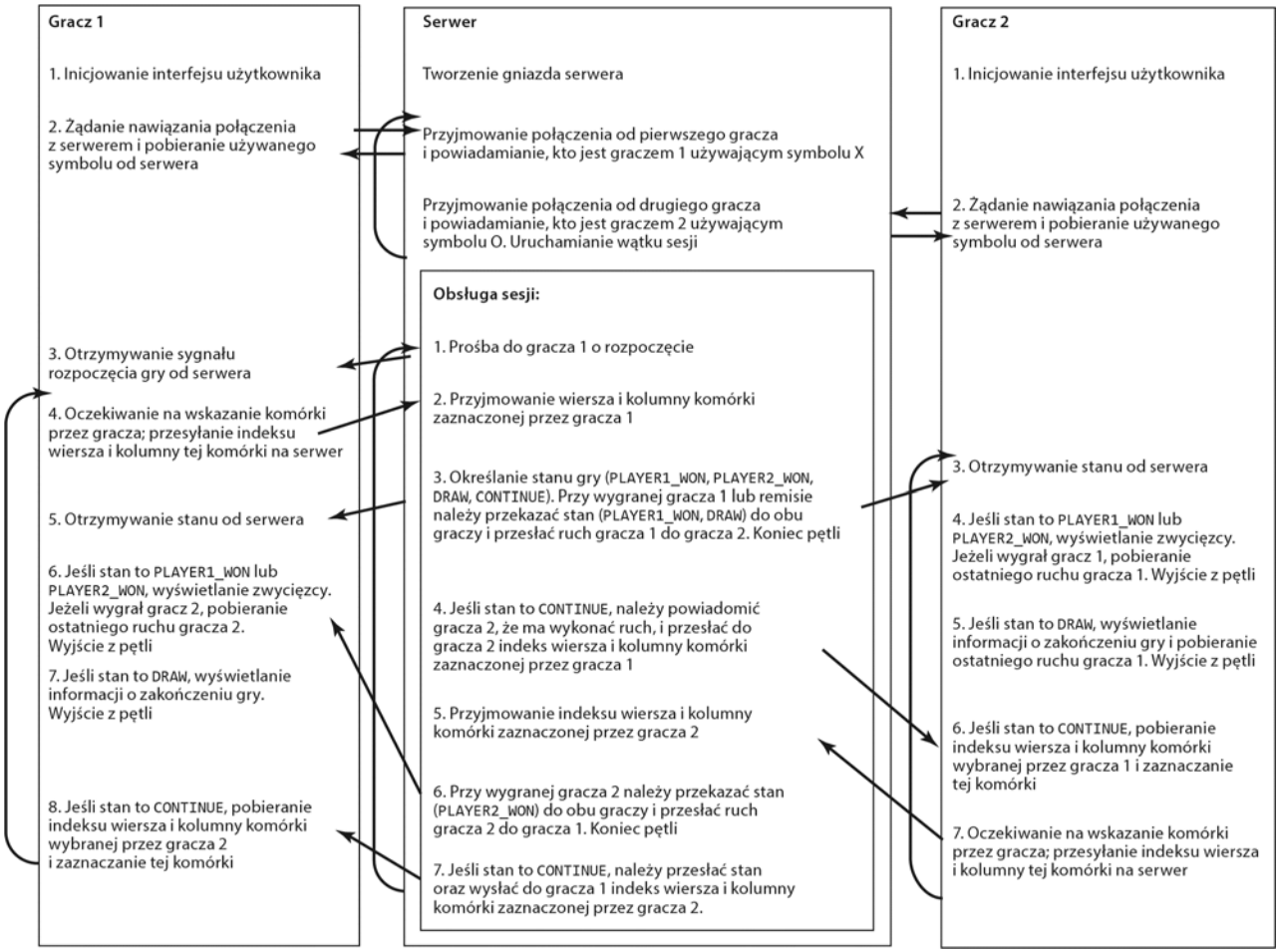
Na podstawie tego omówienia można utworzyć następujące klasy:

- `TicTacToeServer` obsługującą wszystkie klienty (listing 33.9),
- `HandleASession` obsługującą rozgrywkę między dwoma graczami (listing 33.9, *TicTacToeServer.java*),
- `TicTacToeClient` reprezentującą gracza (listing 33.10),
- `Cell` reprezentującą komórkę gry (jest to klasa wewnętrzna w klasie `TicTacToeClient`),
- interfejs `TicTacToeConstants` definiujący stałe wspólne dla wszystkich klas z listingu 33.8.

Zależności między tymi klasami są pokazane na rysunku 33.14.

LISTING 33.8. TicTacToeConstants.java

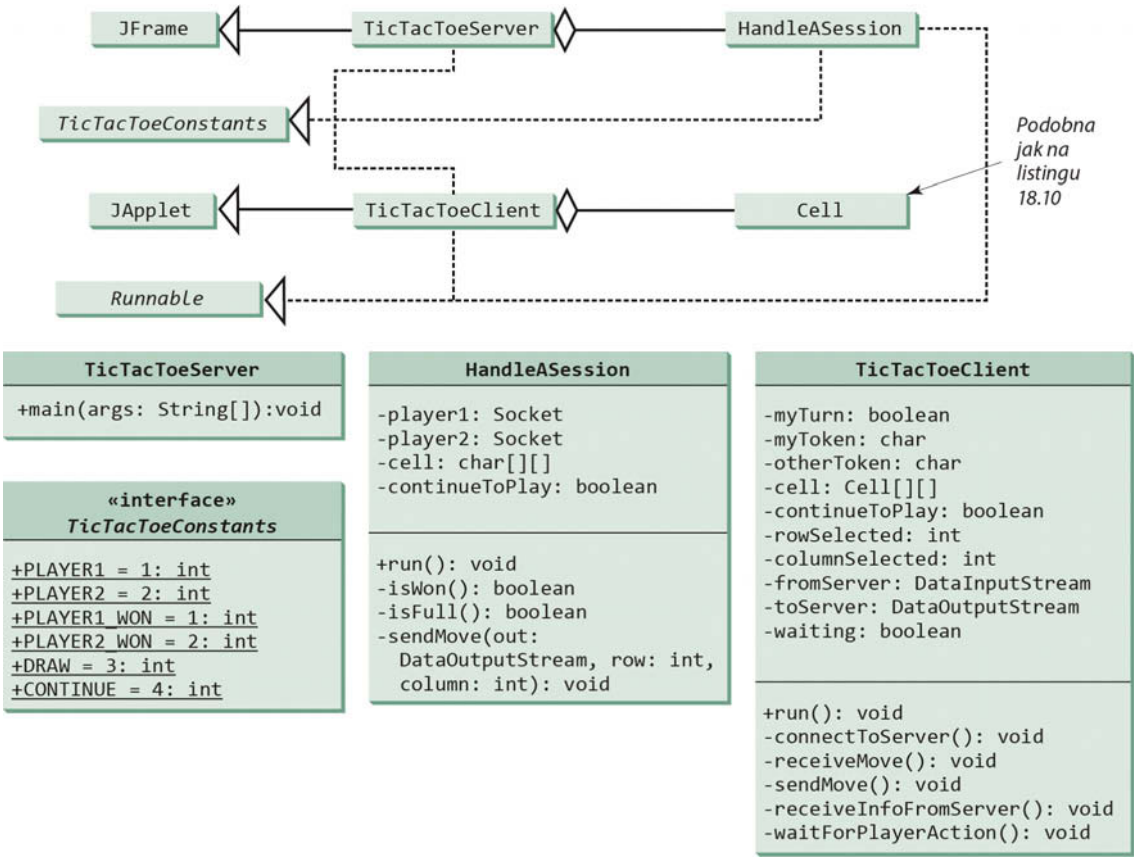
```
1 public interface TicTacToeConstants {
2   public static int PLAYER1 = 1; // Oznacza gracza 1
3   public static int PLAYER2 = 2; // Oznacza gracza 2
4   public static int PLAYER1_WON = 1; // Oznacza wygraną gracza 1
5   public static int PLAYER2_WON = 2; // Oznacza wygraną gracza 2
6   public static int DRAW = 3; // Oznacza remis
7   public static int CONTINUE = 4; // Oznacza kontynuowanie gry
8 }
```



RYSUNEK 33.13. Serwer uruchamia wątek obsługujący komunikację między dwoma graczami

LISTING 33.9. TicTacToeServer.java

```
 1 import java.io.*;
 2 import java.net.*;
 3 import java.util.Date;
 4 import javafx.application.Application;
 5 import javafx.application.Platform;
 6 import javafx.scene.Scene;
 7 import javafx.scene.control.ScrollPane;
 8 import javafx.scene.control.TextArea;
 9 import javafx.stage.Stage;
10
11 public class TicTacToeServer extends Application
12     implements TicTacToeConstants {
13   private int sessionNo = 1; // Numer sesji
14
```



Rysunek 33.14. Klasa TicTacToeServer tworzy obiekt typu HandleASession dla każdej sesji rozgrywki między dwoma graczami. Klasa TicTacToeClient tworzy dziewięć komórek w interfejsie użytkownika

tworzenie interfejsu użytkownika

gniazdo serwera

```
15  @Override // Przesłanianie metody start w klasie Application
16  public void start(Stage primaryStage) {
17    TextArea taLog = new TextArea();
18
19    // Tworzenie sceny i umieszczanie jej w oknie
20    Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
21    primaryStage.setTitle("TicTacToeServer"); // Ustawianie nagłówka okna
22    primaryStage.setScene(scene); // Umieszczanie sceny w oknie
23    primaryStage.show(); // Wyświetlanie okna
24
25    new Thread( () -> {
26      try {
27        // Tworzenie gniazda serwera
28        ServerSocket serverSocket = new ServerSocket(8000);
29        Platform.runLater(() -> taLog.appendText(new Date() +
30          ": Uruchomienie serwera w porcie 8000\n"));
31
32        // Serwer jest gotowy do tworzenia sesji dla każdej pary graczy
```

```
33          while (true) {
34            Platform.runLater(() -> taLog.appendText(new Date() +
35              ": Oczekiwanie na dołączenie graczy do sesji nr " + sessionNo + '\n'));
36
37            // Łączenie się z graczem 1
38            Socket player1 = serverSocket.accept();
39
40            Platform.runLater(() -> {
41              taLog.appendText(new Date() + ": Gracz 1 dołączył do sesji nr "
42                + sessionNo + '\n');
43              taLog.appendText("Adres IP gracza 1 " +
44                player1.getInetAddress().getHostAddress() + '\n');
45            });
46
47            // Powiadamianie, że dana osoba jest graczem 1
48            new DataOutputStream(
49              player1.getOutputStream()).writeInt(PLAYER1);
50
51            // Łączenie się z graczem 2
52            Socket player2 = serverSocket.accept();
53
54            Platform.runLater(() -> {
55              taLog.appendText(new Date() +
56                ": Gracz 2 dołączył do sesji nr " + sessionNo + '\n');
57              taLog.appendText("Adres IP gracza 2 " +
58                player2.getInetAddress().getHostAddress() + '\n');
59            });
60
61            // Powiadamianie użytkownika, że jest graczem 2
62            new DataOutputStream(
63              player2.getOutputStream()).writeInt(PLAYER2);
64
65            // Wyświetlanie danej sesji i zwiększanie jej numeru
66            Platform.runLater(() ->
67              taLog.appendText(new Date() +
68                ": Uruchamianie wątku dla sesji nr " + sessionNo++ + '\n'));
69
70            // Uruchamianie nowego wątku dla sesji dla dwóch graczy
71            new Thread(new HandleASession(player1, player2)).start();
72          }
73        }
74        catch(IOException ex) {
75          ex.printStackTrace();
76        }
77      }).start();
78    }
79
80    // Definicja klasy wątku do obsługi nowej sesji dla dwóch graczy
81    class HandleASession implements Runnable, TicTacToeConstants {
82      private Socket player1;
83      private Socket player2;
84
85      // Tworzenie i inicjowanie komórek
86      private char[][] cell = new char[3][3];
87
```

łączenie się z klientem

dla gracza 1

łączenie się z klientem

dla gracza 2

sesja dla 2 graczy

```
88    private DataInputStream fromPlayer1;
89    private DataOutputStream toPlayer1;
90    private DataInputStream fromPlayer2;
91    private DataOutputStream toPlayer2;
92
93    // Kontynuowanie rozgrywki
94    private boolean continueToPlay = true;
95
96    /** Tworzenie wątku */
97    public HandleASession(Socket player1, Socket player2) {
98      this.player1 = player1;
99      this.player2 = player2;
100
101     // Inicjowanie komórek
102     for (int i = 0; i < 3; i++)
103       for (int j = 0; j < 3; j++)
104         cell[i][j] = ' ';
105   }
106
107   /** Implementacja metody run() dla wątku */
108   public void run() {
109     try {
110       // Tworzenie wejściowego i wyjściowego strumienia danych
111       DataInputStream fromPlayer1 = new DataInputStream(
112         player1.getInputStream());
113       DataOutputStream toPlayer1 = new DataOutputStream(
114         player1.getOutputStream());
115       DataInputStream fromPlayer2 = new DataInputStream(
116         player2.getInputStream());
117       DataOutputStream toPlayer2 = new DataOutputStream(
118         player2.getOutputStream());
119
120       // Można przesłać dowolne dane, aby powiadomić gracza 1, by rozpoczął grę.
121       // Ta instrukcja ma jedynie informować gracza 1 o początku gry
122       toPlayer1.writeInt(1);
123
124       // Stała obsługa graczy oraz ustalanie i przesyłanie
125       // stanu gry
126       while (true) {
127         // Przyjmowanie ruchu od gracza 1
128         int row = fromPlayer1.readInt();
129         int column = fromPlayer1.readInt();
130         cell[row][column] = 'X';
131
132         // Sprawdzanie, czy gracz 1 wygrał
133         if (isWon('X')) {
134           toPlayer1.writeInt(PLAYER1_WON);
135           toPlayer2.writeInt(PLAYER1_WON);
136           sendMove(toPlayer2, row, column);
137           break; // Wyjście z pętli
138         }
139         else if (isFull()) { // Sprawdzanie, czy wszystkie komórki są zapełnione
140           toPlayer1.writeInt(DRAW);
141           toPlayer2.writeInt(DRAW);
142           sendMove(toPlayer2, row, column);
```

strumienie I/O

X wygrał?

pełna plansza?

```
143           break;
144         }
145         else {
146           // Powiadamianie gracza 2, że ma wykonać ruch
147           toPlayer2.writeInt(CONTINUE);
148
149           // Przesyłanie do gracza 2 wiersza i kolumny komórki zaznaczonej przez gracza 1
150           sendMove(toPlayer2, row, column);
151         }
152
153         // Przyjmowanie ruchu od gracza 2
154         row = fromPlayer2.readInt();
155         column = fromPlayer2.readInt();
156         cell[row][column] = 'O';
157
158         // Sprawdzanie, czy gracz 2 wygrał
O wygrał?  159         if (isWon('O')) {
160           toPlayer1.writeInt(PLAYER2_WON);
161           toPlayer2.writeInt(PLAYER2_WON);
162           sendMove(toPlayer1, row, column);
163           break;
164         }
165         else {
166           // Powiadamianie gracza 1, że powinien wykonać ruch
167           toPlayer1.writeInt(CONTINUE);
168
169           // Przesyłanie do gracza 1 wiersza i kolumny komórki wybranej przez gracza 2
170           sendMove(toPlayer1, row, column);
171         }
172       }
173     }
174     catch(IOException ex) {
175       ex.printStackTrace();
176     }
177   }
178
179   /** Przesyłanie ruchu do drugiego gracza */
przesyłanie ruchu  180   private void sendMove(DataOutputStream out, int row, int column)
181       throws IOException {
182     out.writeInt(row); // Przesyłanie indeksu wiersza
183     out.writeInt(column); // Przesyłanie indeksu kolumny
184   }
185
186   /** Sprawdzanie, czy wszystkie komórki są zajęte */
187   private boolean isFull() {
188     for (int i = 0; i < 3; i++)
189       for (int j = 0; j < 3; j++)
190         if (cell[i][j] == ' ')
191           return false; // Przynajmniej jedna komórka jest pusta
192
193     // Wszystkie komórki są zajęte
194     return true;
195   }
196
197   /** Sprawdzanie, czy gracz używający podanego symbolu wygrał */
```

```
198    private boolean isWon(char token) {
199      // Sprawdzanie wszystkich wierszy
200      for (int i = 0; i < 3; i++)
201        if ((cell[i][0] == token)
202            && (cell[i][1] == token)
203            && (cell[i][2] == token)) {
204          return true;
205        }
206
207      /** Sprawdzanie wszystkich kolumn */
208      for (int j = 0; j < 3; j++)
209        if ((cell[0][j] == token)
210            && (cell[1][j] == token)
211            && (cell[2][j] == token)) {
212          return true;
213      }
214
215      /** Sprawdzanie przekątnej lewa góra — prawa dół */
216      if ((cell[0][0] == token)
217          && (cell[1][1] == token)
218          && (cell[2][2] == token)) {
219        return true;
220      }
221
222      /** Sprawdzanie przekątnej lewa dół — prawa góra */
223      if ((cell[0][2] == token)
224          && (cell[1][1] == token)
225          && (cell[2][0] == token)) {
226        return true;
227      }
228
229      /** Sprawdzono wszystkie możliwości, ale nie znaleziono zwycięzcy */
230      return false;
231    }
232  }
233 }
```

LISTING 33.10. TicTacToeClient.java

```
 1 import java.io.*;
 2 import java.net.*;
 3 import java.util.Date;
 4 import javafx.application.Application;
 5 import javafx.application.Platform;
 6 import javafx.scene.Scene;
 7 import javafx.scene.control.Label;
 8 import javafx.scene.control.ScrollPane;
 9 import javafx.scene.control.TextArea;
10 import javafx.scene.layout.BorderPane;
11 import javafx.scene.layout.GridPane;
12 import javafx.scene.layout.Pane;
13 import javafx.scene.paint.Color;
14 import javafx.scene.shape.Ellipse;
15 import javafx.scene.shape.Line;
16 import javafx.stage.Stage;
17
```

```
18 public class TicTacToeClient extends Application
19     implements TicTacToeConstants {
20   // Informuje, na którego gracza przypada ruch
21   private boolean myTurn = false;
22
23   // Symbol używany przez gracza
24   private char myToken = ' ';
25
26   // Symbol używany przez przeciwnika
27   private char otherToken = ' ';
28
29   // Tworzenie i inicjowanie komórek
30   private Cell[][] cell = new Cell[3][3];
31
32   // Tworzenie i inicjowanie etykiety z nagłówkiem
33   private Label lblTitle = new Label();
34
35   // Tworzenie i inicjowanie etykiety ze stanem gry
36   private Label lblStatus = new Label();
37
38   // Wiersz i kolumna komórki zaznaczonej w bieżącym ruchu
39   private int rowSelected;
40   private int columnSelected;
41
42   // Strumienie wejściowy i wyjściowy do komunikacji z serwerem
43   private DataInputStream fromServer;
44   private DataOutputStream toServer;
45
46   // Gra jest kontynuowana?
47   private boolean continueToPlay = true;
48
49   // Oczekiwanie na zaznaczenia komórki przez gracza
50   private boolean waiting = true;
51
52   // Nazwa lub adres IP hosta
53   private String host = "localhost";
54
55   @Override // Przesłanianie metody start z klasy Application
56   public void start(Stage primaryStage) {
57     // Panel z komórkami
```

tworzenie interfejsu użytkownika

```
58     GridPane pane = new GridPane();
59     for (int i = 0; i < 3; i++)
60       for (int j = 0; j < 3; j++)
61         pane.add(cell[i][j] = new Cell(i, j), j, i);
62
63     BorderPane borderPane = new BorderPane();
64     borderPane.setTop(lblTitle);
65     borderPane.setCenter(pane);
66     borderPane.setBottom(lblStatus);
67
68     // Tworzenie sceny i umieszczanie jej w oknie
69     Scene scene = new Scene(borderPane, 320, 350);
70     primaryStage.setTitle("TicTacToeClient"); // Ustawianie nagłówka okna
71     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
72     primaryStage.show(); // Wyświetlanie okna
```

łączenie się z serwerem

pobieranie danych z serwera

przesyłanie danych na serwer

```
73
74      // Łączenie się z serwerem
75      connectToServer();
76    }
77
78    private void connectToServer() {
79      try {
80        // Tworzenie gniazda do łączenia się z serwerem
81        Socket socket = new Socket(host, 8000);
82
83        // Tworzenie strumienia wejściowego do pobierania danych z serwera
84        fromServer = new DataInputStream(socket.getInputStream());
85
86        // Tworzenie strumienia wyjściowego do przesyłania danych na serwer
87        toServer = new DataOutputStream(socket.getOutputStream());
88      }
89      catch (Exception ex) {
90        ex.printStackTrace();
91      }
92
93      // Sterowanie grą w odrębnym wątku
94      new Thread(() -> {
95        try {
96          // Pobieranie powiadomienia od serwera
97          int player = fromServer.readInt();
98
99          // Użytkownik jest graczem 1 czy 2?
100         if (player == PLAYER1) {
101           myToken = 'X';
102           otherToken = 'O';
103           Platform.runLater(() -> {
104             lblTitle.setText("Gracz 1 używający symbolu 'X'");
105             lblStatus.setText("Oczekiwanie na dołączenie gracza 2");
106           });
107
108           // Otrzymanie od serwera powiadomienia o rozpoczęciu gry
109           fromServer.readInt(); // Wczytane dane są ignorowane
110
111           // Drugi gracz dołączył do gry
112           Platform.runLater(() ->
113             lblStatus.setText("Gracz 2 dołączył. Rozpoczynasz grę"));
114
115           // Kolejka danego użytkownika
116           myTurn = true;
117         }
118         else if (player == PLAYER2) {
119           myToken = 'O';
120           otherToken = 'X';
121           Platform.runLater(() -> {
122             lblTitle.setText("Gracz 2 używający symbolu 'O'");
123             lblStatus.setText("Oczekiwanie na ruch gracza 1");
124           });
125         }
126
127         // Kontynuowanie gry
```

```
128         while (continueToPlay) {
129           if (player == PLAYER1) {
130             waitForPlayerAction(); // Oczekiwanie na ruch gracza 1
131             sendMove(); // Przesyłanie ruchu na serwer
132             receiveInfoFromServer(); // Przyjmowanie informacji z serwera
133           }
134           else if (player == PLAYER2) {
135             receiveInfoFromServer(); // Przyjmowanie informacji z serwera
136             waitForPlayerAction(); // Oczekiwanie na ruch gracza 2
137             sendMove(); // Przesyłanie ruchu gracza 2 na serwer
138           }
139         }
140       }
141       catch (Exception ex) {
142         ex.printStackTrace();
143       }
144     }).start();
145   }
146
147   /** Oczekiwanie na zaznaczenie komórki przez gracza */
148   private void waitForPlayerAction() throws InterruptedException {
149     while (waiting) {
150       Thread.sleep(100);
151     }
152
153     waiting = true;
154   }
155
156   /** Przesyłanie ruchu danego gracza na serwer */
157   private void sendMove() throws IOException {
158     toServer.writeInt(rowSelected); // Przesyłanie wiersza
159     toServer.writeInt(columnSelected); // Przesyłanie kolumny
160   }
161
162   /** Przyjmowanie informacji z serwera */
163   private void receiveInfoFromServer() throws IOException {
164     // Pobieranie stanu gry
165     int status = fromServer.readInt();
166
167     if (status == PLAYER1_WON) {
168       // Gracz 1 wygrał, koniec gry
169       continueToPlay = false;
170       if (myToken == 'X') {
171         Platform.runLater(() -> lblStatus.setText("Wygrałeś! (X)"));
172       }
173       else if (myToken == 'O') {
174         Platform.runLater(() ->
175           lblStatus.setText("Wygrał gracz 1 (X)!"));
176         receiveMove();
177       }
178     }
179     else if (status == PLAYER2_WON) {
180       // Wygrał gracz 2, koniec gry
181       continueToPlay = false;
182       if (myToken == 'O') {
```

```
183         Platform.runLater(() -> lblStatus.setText("Wygrałeś! (0)"));
184       }
185       else if (myToken == 'X') {
186         Platform.runLater(() ->
187           lblStatus.setText("Wygrał gracz 2 (0)!"));
188         receiveMove();
189       }
190     }
191     else if (status == DRAW) {
192       // Brak zwycięzcy, koniec gry
193       continueToPlay = false;
194       Platform.runLater(() ->
195         lblStatus.setText("Koniec gry, brak zwycięzcy!"));
196
197       if (myToken == 'O') {
198         receiveMove();
199       }
200     }
201     else {
202       receiveMove();
203       Platform.runLater(() -> lblStatus.setText("Twój ruch"));
204       myTurn = true; // Ruch danego gracza
205     }
206   }
207
208   private void receiveMove() throws IOException {
209     // Pobieranie ruchu drugiego gracza
210     int row = fromServer.readInt();
211     int column = fromServer.readInt();
212     Platform.runLater(() -> cell[row][column].setToken(otherToken));
213   }
214
215   // Klasa wewnętrzna reprezentująca komórkę
reprezentuje komórkę
216   public class Cell extends Pane {
217     // Indeksy wiersza i kolumny danej komórki na planszy
218     private int row;
219     private int column;
220
221     // Symbol umieszczony w komórce
222     private char token = ' ';
223
224     public Cell(int row, int column) {
225       this.row = row;
226       this.column = column;
227       this.setPrefSize(2000, 2000); // Co się stanie, jeśli pominiesz ten wiersz?
228       setStyle("-fx-border-color: black"); // Obramowanie komórki
rejestrowanie odbiornika
229       this.setOnMouseClicked(e -> handleMouseClick());
230     }
231
232     /** Zwracanie symbolu */
233     public char getToken() {
234       return token;
235     }
236
237     /** Ustawianie nowego symbolu */
```

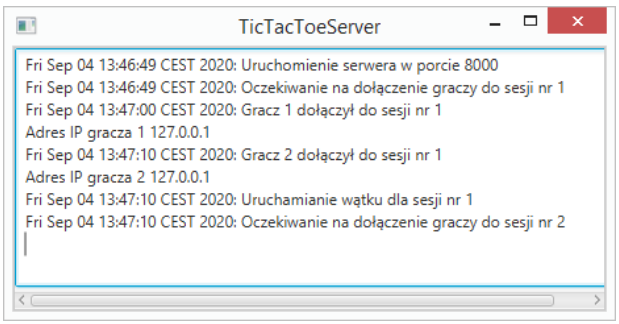
```
238     public void setToken(char c) {
239       token = c;
240       repaint();
241     }
242
243     protected void repaint() {
244       if (token == 'X') {
245         Line line1 = new Line(10, 10,
246           this.getWidth() - 10, this.getHeight() - 10);
247         line1.endXProperty().bind(this.widthProperty().subtract(10));
248         line1.endYProperty().bind(this.heightProperty().subtract(10));
249         Line line2 = new Line(10, this.getHeight() - 10,
250           this.getWidth() - 10, 10);
251         line2.startYProperty().bind(
252           this.heightProperty().subtract(10));
253         line2.endXProperty().bind(this.widthProperty().subtract(10));
254
255         // Dodawanie linii do panelu
256         this.getChildren().addAll(line1, line2);
257       }
258       else if (token == 'O') {
259         Ellipse ellipse = new Ellipse(this.getWidth() / 2,
260           this.getHeight() / 2, this.getWidth() / 2 - 10,
261           this.getHeight() / 2 - 10);
262         ellipse.centerXProperty().bind(
263           this.widthProperty().divide(2));
264         ellipse.centerYProperty().bind(
265           this.heightProperty().divide(2));
266         ellipse.radiusXProperty().bind(
267           this.widthProperty().divide(2).subtract(10));
268         ellipse.radiusYProperty().bind(
269           this.heightProperty().divide(2).subtract(10));
270         ellipse.setStroke(Color.BLACK);
271         ellipse.setFill(Color.WHITE);
272
273         getChildren().add(ellipse); // Dodawanie elipsy do panelu
274       }
275     }
276
277     /* Obsługa zdarzenia kliknięcia myszą */
278     private void handleMouseClick() {
279       // Jeśli komórka nie jest zajęta i przypada ruch danego gracza
280       if (token == ' ' && myTurn) {
281         setToken(myToken); // Należy umieścić symbol tego gracza w komórce
282         myTurn = false;
283         rowSelected = row;
284         columnSelected = column;
285         lblStatus.setText("Oczekiwanie na ruch przeciwnika");
286         waiting = false; // Gracz wykonał poprawny ruch
287       }
288     }
289   }
290 }
```

rysowanie X

rysowanie O

obsługa kliknięcia myszą

Serwer może jednocześnie obsługiwać dowolną liczbę sesji. Każda sesja dotyczy dwóch graczy. Klienta można uruchamiać jako aplet Javy. Aby można było uruchomić klienta jako aplet Javy w przeglądarce, serwer też trzeba włączyć w przeglądarce. Działanie serwera i klientów ilustrują rysunki 33.15 i 33.16.

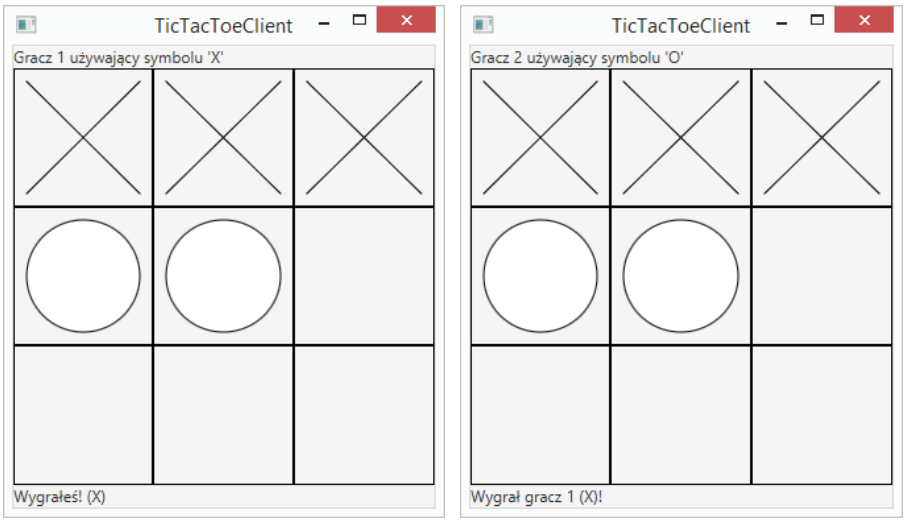


Rysunek 33.15. Klasa TicTacToeServer przyjmuje żądania nawiązania połączenia i tworzy sesje do obsługi par graczy

W interfejsie `TicTacToeConstants` zdefiniowane są stałe wspólne dla wszystkich klas z projektu. Każda klasa używająca tych stałych musi implementować ten interfejs. Definiowanie stałych w jednym interfejsie to technika często stosowana w Javie.

Po utworzeniu sesji serwer na zmianę przyjmuje ruchy od obu graczy. Po otrzymaniu ruchu serwer sprawdza stan gry. Jeśli nie została ona ukończona, serwer przesyła stan (`CONTINUE`) i ruch gracza do przeciwnika. Gdy stan to wygrana lub remis, serwer przesyła tę informację (`PLAYER1_WON`, `PLAYER2_WON` lub `DRAW`) do obu graczy.

Implementacja programów sieciowych w Javie na poziomie gniazd wymaga ścisłej synchronizacji. Operacja przesyłania danych z jednej maszyny wymaga operacji przyjmującej dane na drugiej maszynie. Ten przykład pokazuje, że proces przesyłania i otrzymywania danych po stronie serwera i klienta jest ściśle zsynchronizowany.



Rysunek 33.16. Program TicTacToeClient może działać jako aplet lub jako samodzielna aplikacja

33.6.1. Co się stanie, jeśli nie ustawisz preferowanej wielkości komórek w wierszu 227. listingu 33.10?

33.6.2. Co zrobi program kliencki z listingu 33.10, jeśli pustą komórkę kliknie gracz, na którego nie przypada ruch?

Najważniejsze pojęcia

gniazdo klienta
nazwa domeny
serwer DNS
localhost
adres IP
port
komunikacja pakietowa
gniazdo serwera
gniazdo
komunikacja strumieniowa
TCP
UDP

Podsumowanie rozdziału

1. Java obsługuje gniazda strumieni i gniazda datagramów. *Gniazda strumieni* używają do transmisji danych protokołu TCP, a w *gniazdach datagramów* używany jest protokół UDP. Ponieważ protokół TCP umożliwia wykrywanie utraconych danych i ponowne ich przesyłane, transfer jest bezstratny i niezawodny. Protokół UDP nie gwarantuje transmisji bezstratnej.
2. Aby utworzyć serwer, najpierw trzeba uzyskać gniazdo serwera, używając wywołania `new ServerSocket(port)`. Po utworzeniu gniazda serwer może zacząć oczekiwać na połączenia, używając metody `accept()` gniazda. Klient żąda połączenia z serwerem za pomocą wywołania `new Socket(serverName, port)`, które tworzy gniazdo klienta.
3. Po nawiązaniu połączenia między serwerem i klientem komunikacja z użyciem gniazd strumieni bardzo przypomina komunikację z użyciem strumieni I/O. Strumień wejściowy tworzy metoda `getInputStream()`, a do tworzenia strumienia wyjściowego służy metoda `getOutputStream()` gniazda.
4. Serwer często jednocześnie obsługuje wielu klientów. Na potrzeby jednoczesnej obsługi wielu klientów należy dla każdego połączenia utworzyć osobny wątek.

Quiz

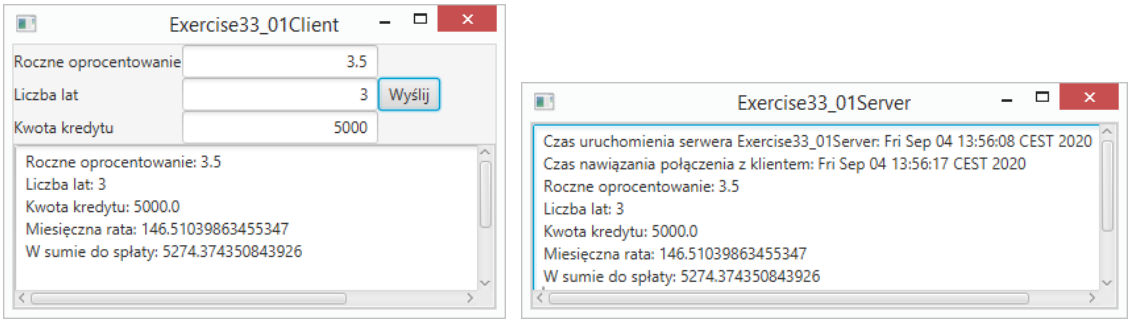
Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

Ćwiczenia programistyczne

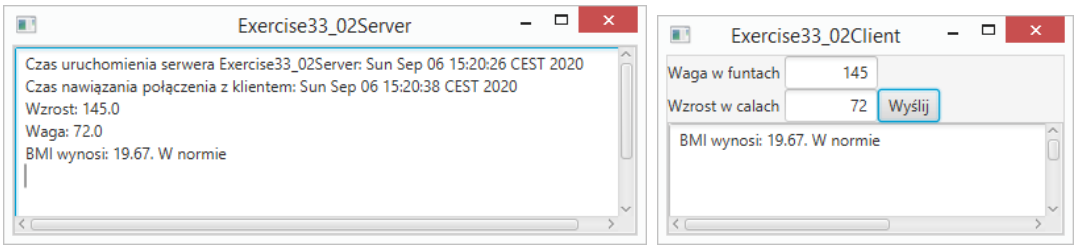
Podrozdział 33.2

***33.1.** *Serwer z informacjami o kredycie.* Napisz serwer do obsługi klientów. Klient ma przekazywać na serwer informacje o kredycie (roczne oprocentowanie, liczbę lat spłaty i kwotę kredytu; rysunek 33.17a). Serwer ma obliczać miesięczną i łączną kwotę do spłaty oraz przekazywać te informacje do klienta (rysunek 33.17b). Nazwij klienta *Exercise33_01Client*, a serwer — *Exercise33_01Server*.

***33.2.** *Serwer zwracający wskaźnik BMI.* Napisz serwer do obsługi klientów. Klient ma przesyłać wzrost i wagę użytkownika na serwer (rysunek 33.18a). Serwer ma obliczać wskaźnik BMI i przesyłać do klienta łańcuch znaków z tym wskaźnikiem (rysunek 33.18b). Obliczanie wskaźnika BMI jest opisane w podrozdziale 3.8. Nazwij klienta *Exercise33_02Client*, a serwer — *Exercise33_02Server*.



RYSUNEK 33.17. Klient (a) przesyła roczne oprocentowanie, liczbę lat spłaty i kwotę kredytu na serwer. Serwer (b) odsyła klientowi wysokość miesięcznej raty i łączną kwotę do spłaty



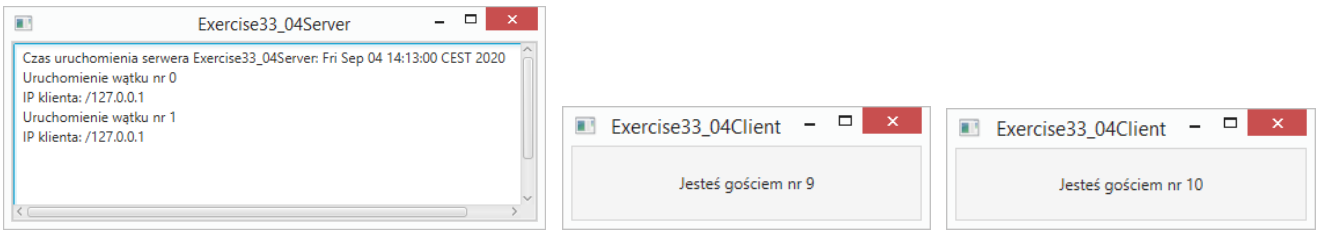
RYSUNEK 33.18. Klient (a) przesyła wagę i wzrost użytkownika na serwer. Serwer (b) odsyła wskaźnik BMI

Podrozdziały 33.3 i 33.4

***33.3.** *Serwer z informacjami o kredycie obsługujący wielu klientów*. Zmodyfikuj rozwiązanie ćwiczenia 33.1, aby serwer obsługiwał wielu klientów.

Podrozdział 33.5

33.4. *Zliczanie klientów*. Napisz serwer, który śledzi liczbę połączonych z nim klientów. Po nawiązaniu nowego połączenia liczba klientów jest zwiększana o 1. Ta liczba jest zapisywana w pliku o dostępie swobodnym. Napisz program kliencki, który pobiera tę liczbę z serwera i wyświetla komunikaty w formacie „Jesteś gościem nr 11” (rysunek 33.19). Nazwij klienta *Exercise33_04Client*, a serwer — *Exercise33_04Server*.

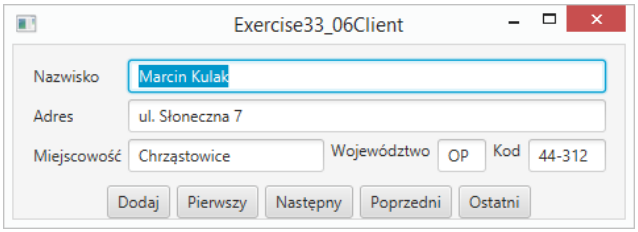


RYSUNEK 33.19. Klient wyświetla, ilu klientów połączyło się z serwerem. Serwer przechowuje tę liczbę

33.5. *Przesyłanie informacji o kredytach w obiekcie.* Zmodyfikuj klienta z ćwiczenia 33.1, aby przesyłał obiekt z informacjami o rocznym oprocentowaniu, liczbie lat spłaty i kwocie kredytu. Zmodyfikuj serwer, aby odsyłał informacje o wysokości raty i łącznej kwocie do spłaty w obiekcie.

Podrozdział 33.6

33.6. *Wyświetlanie i dodawanie adresów.* Napisz aplikację typu klient-serwer służącą do wyświetlania i dodawania adresów (rysunek 33.20).



RYSUNEK 33.20. Możesz wyświetlać i dodawać adresy

- Użyj zdefiniowanej na listingu 33.5 klasy `StudentAddress` do przechowywania nazwiska, ulicy, miasta, województwa i kodu w obiekcie.
- Udostępnij przyciski *Pierwszy*, *Następny*, *Poprzedni* i *Ostatni* do wyświetlania adresów oraz przycisk *Dodaj* do dodawania nowych adresów.
- Ogranicz liczbę jednoczesnych połączeń do dwóch.

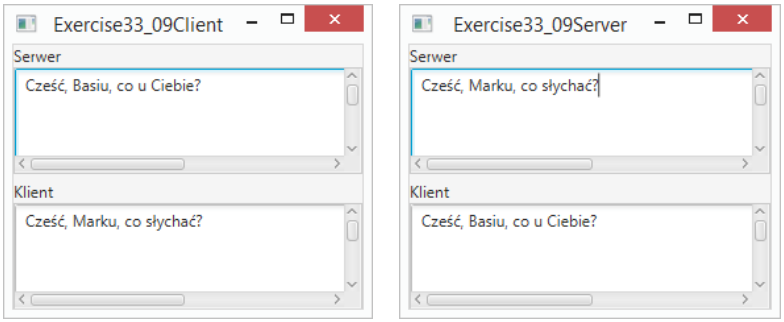
Nazwij klienta *Exercise33_06Client*, a serwer — *Exercise33_06Server*.

***33.7.** *Zapisywanie ostatnich 100 liczb w tablicy.* Rozwiązanie ćwiczenia 22.12 pobiera ostatnich 100 liczb pierwszych z pliku *PrimeNumbers.dat*. Napisz program kliencki, który żąda od serwera 100 ostatnich liczb pierwszych i zapisuje je w tablicy. Nazwij klienta *Exercise33_07Client*, a serwer — *Exercise33_07 Server*. Przyjmij, że liczby są typu `long`, a program przechowuje je w pliku *PrimeNumbers.dat* w formacie dwójkowym.

***33.8.** *Zapisywanie ostatnich 100 liczb w obiekcie typu `ArrayList`.* Rozwiązanie ćwiczenia 22.12 pobiera ostatnich 100 liczb pierwszych z pliku *PrimeNumbers.dat*. Napisz program kliencki, który żąda od serwera 100 ostatnich liczb pierwszych i zapisuje je w obiekcie typu `ArrayList`. Nazwij klienta *Exercise33_08Client*, a serwer — *Exercise33_08Server*. Przyjmij, że liczby są typu `long`, a program przechowuje je w pliku *PrimeNumbers.dat* w formacie dwójkowym.

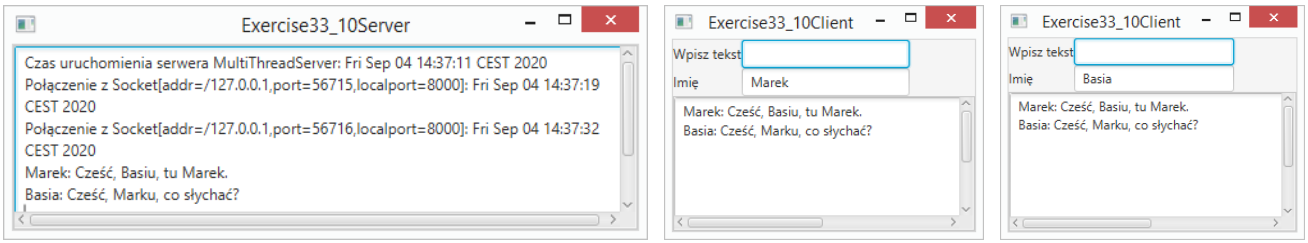
Podrozdział 33.7

****33.9.** *Czat.* Napisz program do obsługi czatów między dwoma użytkownikami. Program jednego użytkownika ma działać jako serwer (rysunek 33.21a), a program drugiego — jako klient (rysunek 33.21b). Serwer ma dwa obszary tekstowe: do wprowadzania tekstu i inny (nieedytowalny) do wyświetlania wiadomości otrzymanych od klienta. Po wciśnięciu klawisza *Enter* nowy wiersz ma być przesyłany do klienta. Klient ma dwa obszary tekstowe: nieedytowalny do wyświetlania tekstu z serwera i inny do wprowadzania tekstu. Po wciśnięciu klawisza *Enter* nowy wiersz ma być przesyłany na serwer. Nazwij klienta *Exercise33_09Client*, a serwer — *Exercise33_09Server*.



Rysunek 33.21. Serwer i klient przesyłają między sobą tekst

*****33.10.** *Czat z obsługą wielu klientów*. Napisz program umożliwiający udział w czacie dowolnej liczbie klientów. Zaimplementuj serwer obsługujący wszystkie klienty (rysunek 33.22). Nazwij klienta *Exercise33_10Client*, a serwer — *Exercise33_10Server*.



Rysunek 33.22. Serwer (a) obsługuje dwóch klientów — (b) i (c)

ROZDZIAŁ

34

Umiędzynarodowienie

Cele

- Omówienie mechanizmów umiędzynarodowiania dostępnych w Javie (podrozdział 34.1).
- Ustawienia regionalne oprogramowania na podstawie języka, państwa i dialektu (podrozdział 34.2).
- Wyświetlanie daty i czasu zgodnie z ustawieniami regionalnymi (podrozdział 34.3).
- Wyświetlanie liczb, kwot pieniędzy i procentów zgodnie z ustawieniami regionalnymi (podrozdział 34.4).
- Tworzenie aplikacji dla odbiorców międzynarodowych z wykorzystaniem pakietów zasobów (podrozdział 34.5).
- Podawanie schematu kodowania na potrzeby tekstowych operacji I/O (podrozdział 34.6).

34.1. Wprowadzenie

Ten rozdział stanowi wprowadzenie do pisania w Javie kodu dla odbiorców międzynarodowych[1].

Wiele witryn udostępnia kilka wersji stron, dzięki czemu czytelnicy mogą wybrać zrozumiały dla nich język. Ponieważ na świecie istnieje wiele języków, bardzo trudno byłoby tworzyć i utrzymywać tyle różnych wersji, by spełnić wymagania wszystkich odbiorców. Java pomaga wykonać to zadanie. Jest to pierwszy język, który od początku był tak projektowany, by umożliwiać umiędzynarodowianie. W Javie można dostosować programy do dowolnej liczby państw i języków bez konieczności wprowadzania skomplikowanych zmian w kodzie.

Oto główne mechanizmy Javy wspomagające umiędzynarodowianie:

- W Javie używane są znaki *Unicode*. Jest to 16-bitowy schemat kodowania opracowany przez konsorcjum Unicode, ułatwiający wymianę, parsowanie i wyświetlanie pisanych tekstów w różnych językach świata. Kodowanie Unicode sprawia, że łatwo jest pisać w Javie programy do operowania łańcuchami znaków w dowolnym języku. Wszystkie znaki Unicode znajdziesz na stronie *http://mindprod.com/jgloss/reuters.html*.
- Java udostępnia klasę `Locale` do przechowywania informacji o ustawieniach regionalnych. Obiekt typu `Locale` określa sposób wyświetlania zależnych od ustawień regionalnych informacji, na przykład dat, czasu i liczb, a także wykonywania zależnych od ustawień regionalnych operacji takich jak sortowanie łańcuchów znaków. Klasy do formatowania dat, czasu i liczb oraz sortowania łańcuchów znaków znajdują się w pakiecie `java.text`.
- Java obejmuje klasę `ResourceBundle` do oddzielania informacji specyficznych dla ustawień regionalnych (na przykład komunikatów o stanie i napisów na elementach GUI) od kodu programu. Wspomniane informacje są przechowywane niezależnie od kodu źródłowego i mogą być używane oraz dynamicznie wczytywane w czasie wykonywania programu z obiektu typu `ResourceBundle`. Nie trzeba ich na stałe zapisywać w programie.

W tym rozdziale zobaczysz, jak formatować daty, liczby, kwoty pieniędzy i procenty na podstawie różnych regionów, państw i języków. Dowiesz się też, jak używać pakietów zasobów do definiowania, które obrazy i łańcuchy znaków mają być używane w komponencie w zależności od ustawień regionalnych i preferencji użytkownika.

34.2. Klasa Locale

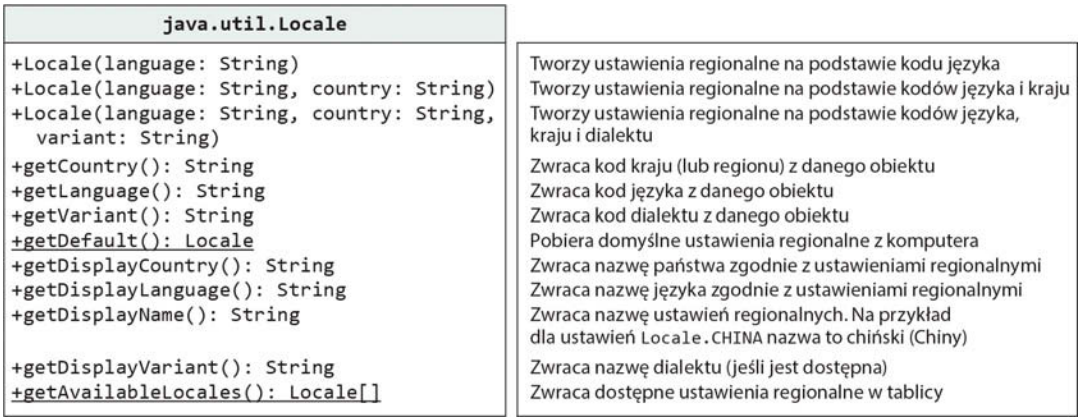
Klasa `Locale` definiuje ustawienia regionalne: język i kraj.

Obiekt typu `Locale` reprezentuje geograficzny, polityczny lub kulturowy region, w którym używany jest określony język lub dane formatowanie. Na przykład Amerykanie mówią po angielsku, a Chińczycy po chińsku. Konwencje dotyczące formatowania dat, liczb, walut i procentów mogą być w poszczególnych krajach różne. Na przykład Chińczycy zapisują daty w formacie rok/miesiąc/dzień, a Amerykanie za pomocą formatu miesiąc/dzień/rok. Należy zauważyć, że ustawienia regionalne zależą nie tylko od kraju. Na przykład Kanadyjczycy w zależności od zamieszkiwanego regionu używają albo kanadyjskiego dialektu języka angielskiego, albo kanadyjskiego dialektu języka francuskiego.

Aby utworzyć obiekt typu `Locale`, użyj jednego z trzech konstruktorów, podając określony język oraz opcjonalnie kraj i dialekt (rysunek 34.1).

Język należy podać za pomocą poprawnego kodu języka (dwie małe litery zgodnie ze standardem ISO-639). Na przykład `zh` oznacza język chiński, `da` duński, `en` angielski, `de` niemiecki, a `ko` koreański. Listę kodów języków zawiera tabela 34.1.

[1] W witrynie poświęconej oryginalnemu wydaniu książki jest to rozdział 36. — *przyp. tłum.*



RYSUNEK 34.1. Klasa Locale zawiera ustawienia regionalne

TABELA 34.1. Często stosowane kody języków

Kod	*Język*	*Kod*	*Język*
da	duński	ja	japoński
de	niemiecki	ko	koreański
el	grecki	nl	holenderski
en	angielski	no	norweski
es	hiszpański	pt	portugalski
fi	fiński	sv	szwedzki
fr	francuski	tr	turecki
it	włoski	zh	chiński
pl	polski		

Jako państwo należy podać poprawny kod ISO kraju (dwie wielkie litery zgodnie ze standardem ISO-3166). Na przykład `CA` oznacza Kanadę, `CN` to Chiny, `DK` to Dania, `DE` to Niemcy, a `US` to Stany Zjednoczone. Listę kodów państw znajdziesz w tabeli 34.2.

Argument określający dialekt jest stosowany rzadko. Jest on potrzebny tylko w wyjątkowych i zależnych od systemu sytuacjach, aby określać informacje specyficzne dla przeglądarki lub producenta oprogramowania. Na przykład w języku norweskim stosowane są dwa zbiory zasad pisowni: tradycyjna (nazywana *bokmål*) i nowa (nazywana *nynorsk*). Ustawienia regionalne z pisownią tradycyjną można utworzyć tak:

```
new Locale("no", "NO", "B");
```

Dla wygody w klasie `Locale` dostępnych jest wiele predefiniowanych stałych dla ustawień regionalnych (na przykład `Locale.CANADA` reprezentuje Kanadę i język angielski, `Locale.CANADA_FRENCH` oznacza Kanadę i język francuski). Oto kilka innych często używanych stałych:

`Locale.US`, `Locale.UK`, `Locale.FRANCE`, `Locale.GERMANY`, `Locale.ITALY`, `Locale.CHINA`, `Locale.KOREA`, `Locale.JAPAN` i `Locale.TAIWAN`.

TABELA 34.2. Często używane kody państw

Kod	*Kraj*	*Kod*	*Kraj*
AT	Austria	IE	Irlandia
BE	Belgia	HK	Hongkong
CA	Kanada	IT	Włochy
CH	Szwajcaria	JP	Japonia
CN	Chiny	KR	Korea
DE	Niemcy	NL	Holandia
DK	Dania	NO	Norwegia
ES	Hiszpania	PT	Portugalia
FI	Finlandia	SE	Szwecja
FR	Francja	TR	Turcja
GB	Wielka Brytania	TW	Tajwan
GR	Grecja	US	Stany Zjednoczone
PL	Polska		

W klasie `Locale` dostępne są też stałe reprezentujące język:

`Locale.CHINESE`, `Locale.ENGLISH`, `Locale.FRENCH`, `Locale.GERMAN`, `Locale.ITALIAN`, `Locale.JAPANESE`, `Locale.KOREAN`, `Locale.SIMPLIFIED_CHINESE` i `Locale.TRADITIONAL_CHINESE`.

Wskazówka

Aby pobrać wszystkie ustawienia regionalne dostępne w systemie, możesz wywołać metodę statyczną `getAvailableLocales()` z klasy `Calendar`:

```
Locale[] availableLocales = Calendar.getAvailableLocales();
```

To wywołanie zwraca tablicę z wszystkimi ustawieniami regionalnymi.

Wskazówka

W komputerze używane są domyślne ustawienia regionalne. Możesz je zastąpić, podając w parametrach język i region w momencie uruchamiania programu:

```
java –Duser.language=zh –Duser.region=CN MainClass
```

Niektóre operacje są *zależne od ustawień regionalnych*. Takimi operacjami są m.in. wyświetlanie wartości takich jak data lub czas. Liczby należy formatować zgodnie ze zwyczajami typowymi dla ustawień regionalnych użytkownika. W dalszych podrozdziałach opisane są operacje zależne od ustawień regionalnych.

34.2.1. W jaki sposób Java obsługuje znaki z języków takich jak chiński lub arabski?

34.2.2. Jak utworzyć obiekt typu `Locale`? Jak pobrać wszystkie dostępne ustawienia regionalne za pomocą obiektu typu `Calendar`?

34.2.3. Jak utworzyć ustawienia regionalne dla francuskojęzycznego regionu Kanady? Jak utworzyć ustawienia regionalne dla Holandii?

34.3. Wyświetlanie daty i czasu

Reprezentacja daty i czasu jest zależna od ustawień regionalnych.

Aplikacje często potrzebują daty i czasu. Java udostępnia niezależną od systemu reprezentację daty i czasu za pomocą klasy `java.util.Date`. Dostępne są też klasa `java.util.TimeZone` reprezentująca strefy czasowe i klasa `java.util.Calendar` do pobierania szczegółowych informacji z obiektu typu `Date`. Dla różnych ustawień regionalnych obowiązują różne sposoby wyświetlania dat i czasu. Czy najpierw należy wyświetlić rok, miesiąc, czy dzień? Czy do rozdzielania pól daty należy używać ukośników, kropek czy dwukropków? Jakie są nazwy miesięcy w danym języku? Do formatowania dat i czasu w sposób zależny od ustawień regionalnych można użyć klasy `java.text.Date` ➥`Format`. Klasa `Date` została opisana w punkcie 9.6.1, „Klasa Date”, a klasa `Calendar` i jej podklasa `GregorianCa` ➥`lendar` w podrozdziale 13.4, „Studium przypadku — Calendar i GregorianCalendar”.

34.3.1. Klasa TimeZone

Klasa `TimeZone` reprezentuje strefę czasową, a także określa, czy używany jest czas letni, czy zimowy. Aby uzyskać obiekt typu `TimeZone` dla strefy o określonym identyfikatorze, użyj wywołania `TimeZone.getTimeZone(id)`. W celu ustawienia strefy czasowej w obiekcie typu `Calendar` użyj metody `setTimeZone` i identyfikatora strefy. Na przykład wywołanie `cal.setTimeZone(TimeZone.getTimeZone("CST"))` ustawia strefę czasową na czas centralny (ang. *Central Standard Time*). Aby wyświetlić wszystkie strefy czasowe obsługiwane w Javie, zastosuj statyczną metodę `getAvailableIDs()` z klasy `TimeZone`. Międzynarodowe identyfikatory stref czasowych są zwykle łańcuchami znaków w formacie kontynent/miasto (na przykład Europe/Berlin, Asia/Taipei lub America/Washington). Możesz też użyć statycznej metody `getDefault()` z klasy `TimeZone`, aby pobrać domyślną strefę czasową z danego hosta.

34.3.2. Klasa DateFormat

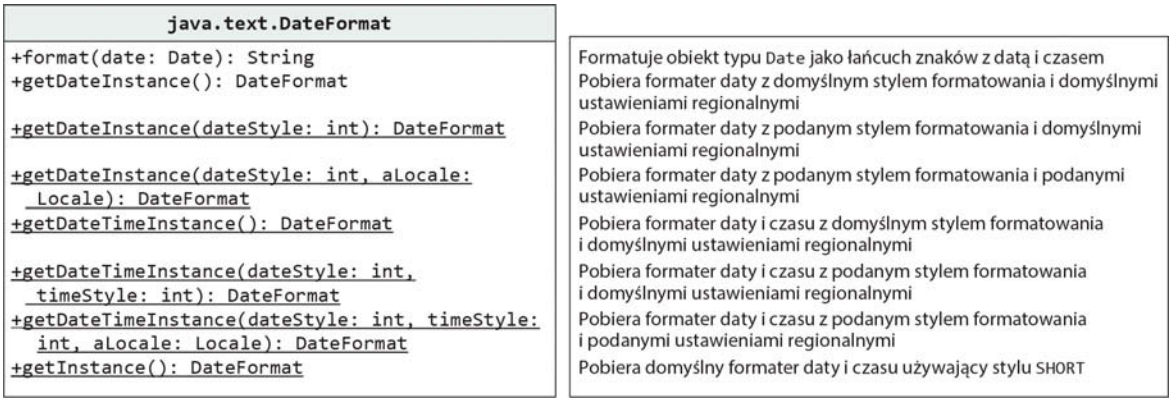
Klasa `DateFormat` umożliwia formatowanie daty i czasu na różne sposoby. Dostępnych jest kilka standardowych stylów formatowania. Aby sformatować datę i czas, utwórz obiekt typu `DateFormat` za pomocą którejś z trzech metod statycznych (`getDateInstance`, `getTimeInstance` lub `getDateTimeInstance`) i wywołaj dla tego obiektu metodę `format(Date)` (rysunek 34.2).

Jako parametry `dateStyle` i `timeStyle` można podawać następujące stałe: `DateFormat.SHORT`, `DateFormat.MEDIUM`, `DateFormat.LONG`, `DateFormat.FULL`. Dokładny efekt zależy od ustawień regionalnych, ale zwykle:

- stała `SHORT` oznacza zapis liczbowy, na przykład `20-09-07` (data) i `00:50` (czas);
- stała `MEDIUM` oznacza dłuższy zapis, na przykład `2020-09-07` (data) i `00:52:16` (czas);
- stała `LONG` to jeszcze dłuższy zapis, na przykład `7 września 2020` (data) i `00:53:15 CEST` (czas);
- stała `FULL` oznacza pełny zapis, na przykład `poniedziałek, 7 września 2020` (data) i `00:54:01 CEST` (czas).

Pokazane poniżej instrukcje wyświetlają bieżący czas zgodnie z parametrami: strefą (`CST`), stylem formatowania (`FULL` dla daty i `FULL` dla czasu) oraz ustawieniami regionalnymi (`US`).

```
GregorianCalendar calendar = new GregorianCalendar();
DateFormat formatter = DateFormat.getDateTimeInstance(
  DateFormat.FULL, DateFormat.FULL, Locale.US);
TimeZone timeZone = TimeZone.getTimeZone("CST");
formatter.setTimeZone(timeZone);
System.out.println("Czas lokalny: " +
  formatter.format(calendar.getTime()));
```



RYSUNEK 34.2. Klasa DateFormat służy do formatowania daty i czasu

34.3.3. Klasa SimpleDateFormat

Podklasa `SimpleDateFormat` służąca do formatowania daty i czasu umożliwia ustawienie dowolnego wzorca formatowania. Pokazany poniżej konstruktor pozwala utworzyć obiekt tego typu, który umożliwia przekształcenie obiektu typu `Date` na łańcuch znaków o oczekiwanym formacie.

```
public SimpleDateFormat(String pattern)
```

Parametr `pattern` to łańcuch składający się ze znaków o specjalnym znaczeniu. Na przykład `y` oznacza rok, `M` to miesiąc, `d` to dzień miesiąca, `G` określa erę, `h` to godzina, `m` to minuta, `s` to sekunda, a `z` to strefa czasowa. Poniższy kod wyświetla więc tekst w formacie: „Aktualny czas: 2020.09.06 n.e., godzina 05:32:31 CEST” na podstawie wzorca `"yyyy.MM.dd G ', godzina' hh:mm:ss z"`.

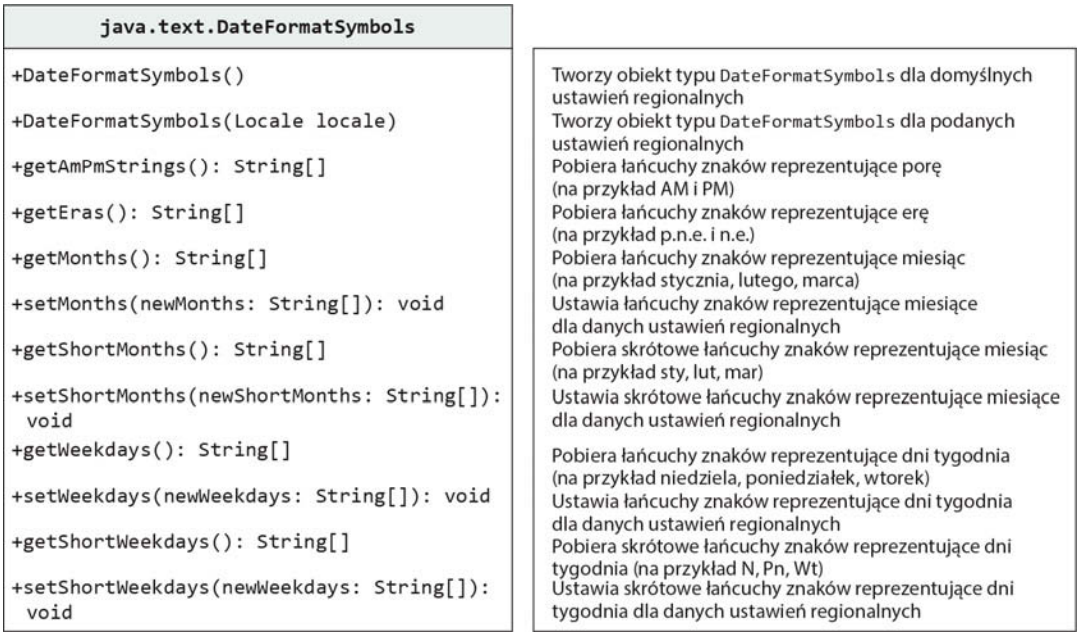
```
SimpleDateFormat formatter
  = new SimpleDateFormat("yyyy.MM.dd G ', godzina' hh:mm:ss z");
Date currentTime = new Date();
String dateString = formatter.format(currentTime);
System.out.println("Aktualny czas: " + dateString);
```

34.3.4. Klasa DateFormatSymbols

Klasa `DateFormatSymbols` zawiera dane na potrzeby formatowania dat i czasu zgodnie z ustawieniami regionalnymi, na przykład nazwy miesięcy i dni tygodnia (rysunek 34.3).

Na przykład poniższa instrukcja wyświetla nazwy miesięcy i dni tygodnia zgodnie z domyślnymi ustawieniami regionalnymi:

```
DateFormatSymbols symbols = new DateFormatSymbols();
String[] monthNames = symbols.getMonths();
for (int i = 0; i < monthNames.length; i++) {
  System.out.println(monthNames[i]); // Wyświetla stycznia, ...
}

String[] weekdayNames = symbols.getWeekdays();
for (int i = 0; i < weekdayNames.length; i++) {
  System.out.println(weekdayNames[i]); // Wyświetla niedziela, poniedziałek, ...
}
```



RYSUNEK 34.3. Klasa DateFormatSymbols zawiera dane do formatowania daty i czasu zgodnie z ustawieniami regionalnymi

Następne dwa przykłady pokazują, jak wyświetlać datę, czas i kalendarz zgodnie z ustawieniami regionalnymi. Pierwszy przykład ilustruje tworzenie zegara i wyświetlanie daty oraz czasu w formacie zgodnym z ustawieniami regionalnymi. Drugi przykład wyświetla kilka kalendarzy z nazwami dni tygodnia w odpowiednich językach lokalnych.

34.3.5. Przykład — wyświetlanie zegara międzynarodowego

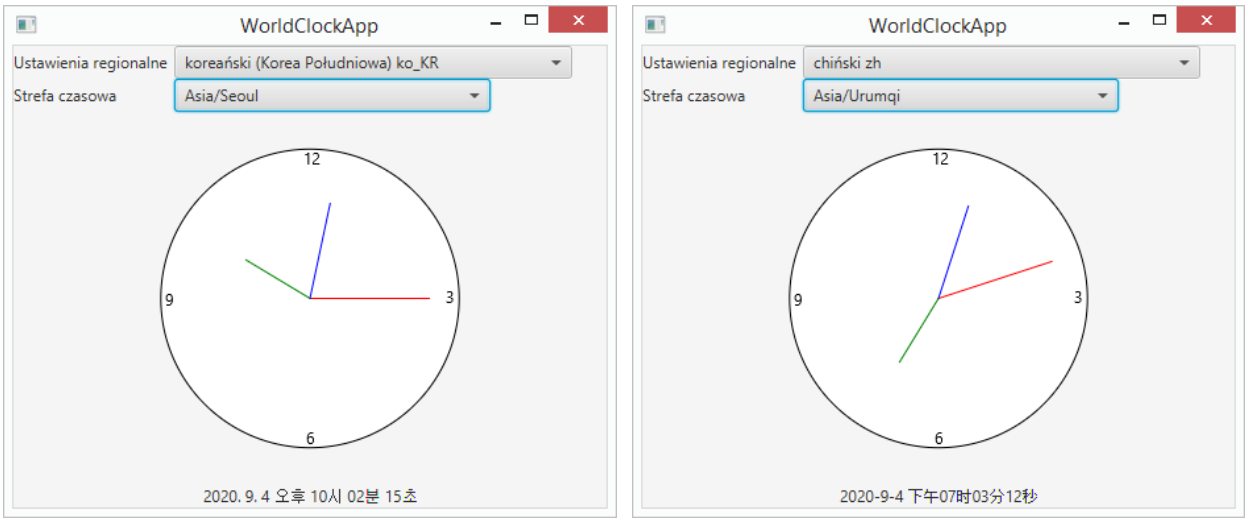
Napisz program, który pokazuje zegar wyświetlający bieżący czas zgodnie z podanymi ustawieniami regionalnymi i strefą czasową. Ustawienia regionalne i strefę czasową można wybrać w polach kombi zawierających dostępne ustawienia i strefy (rysunek 34.4).

Oto główne etapy tworzenia tego programu:

1. Utwórz klasę `WorldClock` rozszerzającą klasę `BorderPane` (listing 34.1). Nowa klasa ma wyświetlać pośrodku panelu obiekt typu `ClockPane` (z listingu 14.21, *ClockPane.java*). Dodaj kontrolkę `Label` do wyświetlania czasu za pomocą cyfr i umieść ją u dołu. Użyj klasy `GregorianCalendar` do pobierania bieżącego czasu zgodnie z podanymi ustawieniami regionalnymi i strefą czasową.

LISTING 34.1. WorldClock.java

```
1 import java.util.Calendar;
2 import java.util.TimeZone;
3 import java.util.GregorianCalendar;
4 import java.text.*;
5 import java.util.Locale;
6 import javafx.animation.KeyFrame;
7 import javafx.animation.Timeline;
```



RYSUNEK 34.4. Program wyświetla zegar z bieżącym czasem zgodnie z wybranymi ustawieniami regionalnymi i strefą czasową

```
 8 import javafx.event.ActionEvent;
 9 import javafx.event.EventHandler;
10 import javafx.geometry.Pos;
11 import javafx.scene.control.Label;
12 import javafx.scene.layout.BorderPane;
13 import javafx.util.Duration;
14
15 public class WorldClock extends BorderPane {
16   private TimeZone timeZone = TimeZone.getTimeZone("EST");
17   private Locale locale = Locale.getDefault();
18   private ClockPane clock = new ClockPane(); // Nadal jest to zegar
19   private Label lblDigitTime = new Label();
20
21   public WorldClock() {
22     setCenter(clock);
23     setBottom(lblDigitTime);
24     BorderPane.setAlignment(lblDigitTime, Pos.CENTER);
25
26     EventHandler<ActionEvent> eventHandler = e -> {
27       setCurrentTime(); // Ustawianie nowego czasu
28     };
29
30     // Tworzenie animacji pracy zegara
31     Timeline animation = new Timeline(
32       new KeyFrame(Duration.millis(1000), eventHandler));
33     animation.setCycleCount(Timeline.INDEFINITE);
34     animation.play(); // Uruchamianie animacji
35
36     // Zmiana wielkości zegara
37     widthProperty().addListener(ov -> clock.setWidth(getWidth()));
38     heightProperty().addListener(ov -> clock.setHeight(getHeight()));
39   }
```

```
40
41   public void setTimeZone(TimeZone timeZone) {
42     this.timeZone = timeZone;
43   }
44
45   public void setLocale(Locale locale) {
46     this.locale = locale;
47   }
48
49   private void setCurrentTime() {
50     Calendar calendar = new GregorianCalendar(timeZone, locale);
51     clock.setHour(calendar.get(Calendar.HOUR));
52     clock.setMinute(calendar.get(Calendar.MINUTE));
53     clock.setSecond(calendar.get(Calendar.SECOND));
54
55     // Wyświetlanie czasu w etykiecie
56     DateFormat formatter = DateFormat.getDateTimeInstance
57       (DateFormat.MEDIUM, DateFormat.LONG, locale);
58     formatter.setTimeZone(timeZone);
59     lblDigitTime.setText(formatter.format(calendar.getTime()));
60   }
61 }
```

2. Utwórz klasę `WorldClockControl` rozszerzającą klasę `BorderPane` (listing 34.2). Nowa klasa ma zawierać obiekt typu `WorldClock` i dwie kontrolki `ComboBox` służące do wybierania ustawień regionalnych oraz stref czasowych.

LISTING 34.2. WorldClockControl.java

```
 1 import java.util.*;
 2 import javafx.geometry.Pos;
 3 import javafx.scene.control.ComboBox;
 4 import javafx.scene.control.Label;
 5 import javafx.scene.layout.BorderPane;
 6 import javafx.scene.layout.GridPane;
 7
 8 public class WorldClockControl extends BorderPane {
 9   // Pobieranie identyfikatorów wszystkich dostępnych ustawień regionalnych i stref czasowych
10   private Locale[] availableLocales = Locale.getAvailableLocales();
11   private String[] availableTimeZones = TimeZone.getAvailableIDs();
12
13   // Pola kombi do wyświetlania dostępnych ustawień regionalnych i stref czasowych
14   private ComboBox<String> cboLocales = new ComboBox<>();
15   private ComboBox<String> cboTimeZones = new ComboBox<>();
16
17   // Tworzenie zegara
18   private WorldClock clock = new WorldClock();
19
20   public WorldClockControl() {
21     // Inicjowanie pola cboLocales wszystkimi dostępnymi ustawieniami regionalnymi
22     setAvailableLocales();
23
24     // Inicjowanie pola cboTimeZones wszystkimi dostępnymi strefami czasowymi
25     setAvailableTimeZones();
26
27     // Inicjowanie ustawień regionalnych i strefy czasowej
```

```
    clock.setLocale(
      availableLocales[cboLocales.getSelectionModel()
        .getSelectedIndex()]);
    clock.setTimeZone(TimeZone.getTimeZone(
      availableTimeZones[cboTimeZones.getSelectionModel()
        .getSelectedIndex()]));

    GridPane pane = new GridPane();
    pane.setHgap(5);
    pane.add(new Label("Ust. region."), 0, 0);
    pane.add(new Label("Strefa czas."), 0, 1);
    pane.add(cboLocales, 1, 0);
    pane.add(cboTimeZones, 1, 1);

    setTop(pane);
    setCenter(clock);
    BorderPane.setAlignment(pane, Pos.CENTER);
    BorderPane.setAlignment(clock, Pos.CENTER);

    cboLocales.setOnAction(e ->
      clock.setLocale(availableLocales[cboLocales.
        getSelectionModel().getSelectedIndex()]));
    cboTimeZones.setOnAction(e ->
      clock.setTimeZone(TimeZone.getTimeZone(
        availableTimeZones[cboTimeZones.
          getSelectionModel().getSelectedIndex()])));
  }

  private void setAvailableLocales() {
    for (int i = 0; i < availableLocales.length; i++)
      cboLocales.getItems().add(availableLocales[i]
        .getDisplayName() + " " + availableLocales[i].toString());

    cboLocales.getSelectionModel().selectFirst();
  }

  private void setAvailableTimeZones() {
    // Sortowanie stref czasowych
    Arrays.sort(availableTimeZones);
    for (int i = 0; i < availableTimeZones.length; i++) {
      cboTimeZones.getItems().add(availableTimeZones[i]);
    }
    cboTimeZones.getSelectionModel().selectFirst();
  }
}
```

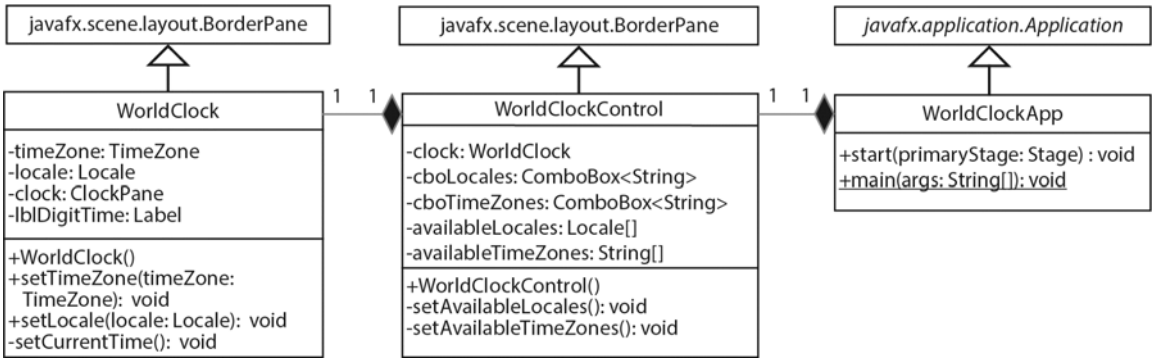
3. Utwórz aplikację `WorldClockApp` (listing 34.3) do wyświetlania obiektu typu `WorldClockControl`.

LISTING 34.3. WorldClockApp.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class WorldClockApp extends Application {
  @Override // Przesłanianie metody start w klasie Application
```

```
 7   public void start(Stage primaryStage) {
 8     // Tworzenie sceny i umieszczanie jej w oknie
 9     Scene scene = new Scene(new WorldClockControl() ,450, 350);
10     primaryStage.setTitle("WorldClockApp"); // Ustawianie nagłówka okna
11     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
12     primaryStage.show(); // Wyświetlanie okna
13   }
14 }
```

Zależności między tymi klasami są pokazane na rysunku 34.5.



RYSUNEK 34.5. Klasa WorldClockApp zawiera obiekt typu WorldClockControl, a ten obejmuje obiekt typu WorldClock

Klasa `WorldClock` tworzy panel typu `ClockPane` (wiersz 18.) i wyśrodkowuje go (wiersz 22.). Metoda `setCurrent`
`↪Time()` za pomocą klasy `GregorianCalendar` pobiera obiekt typu `Calendar` dla podanych ustawień regionalnych i strefy czasowej (wiersz 50.). Czas na zegarze jest aktualizowany co sekundę za pomocą obiektu typu `Calendar` (wiersze 51. – 53.).

W wierszach 56. i 57. program tworzy obiekt typu `DateFormat`, po czym używa go do sformatowania daty zgodnie z ustawieniami regionalnymi (wiersz 59.).

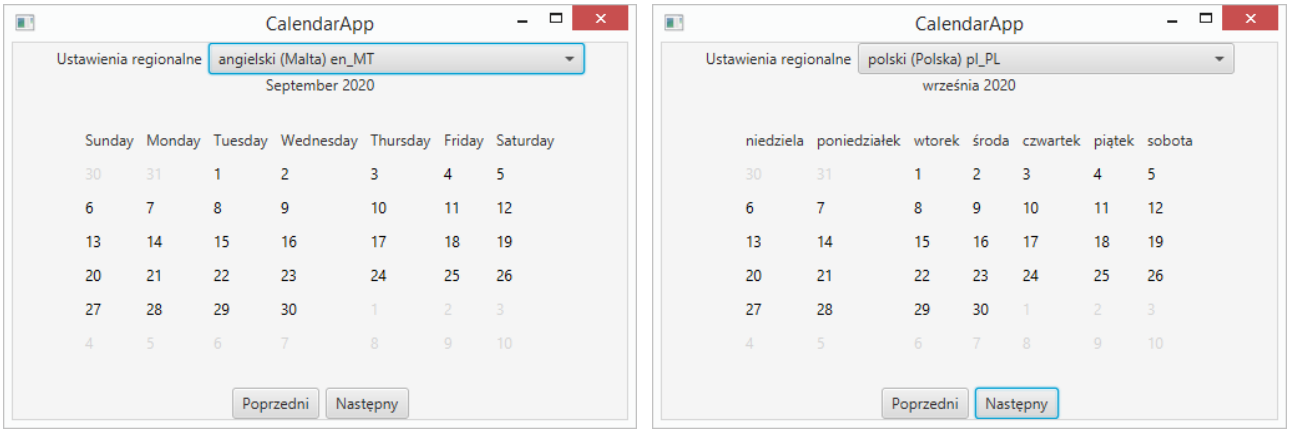
Klasa `WorldClockControl` zawiera obiekt typu `WorldClock` i dwa pola kombi. Te pola zawierają wszystkie dostępne ustawienia regionalne i strefy czasowe (wiersze 56. – 71.). Wybrane ustawienia regionalne i strefa czasowa są stosowane do zegara (wiersze 47. – 53.) oraz służą do wyświetlania nowego czasu zgodnie z bieżącymi ustawieniami regionalnymi i strefą czasową.

34.3.6. Przykład — wyświetlanie kalendarza

Napisz program wyświetlający kalendarz zgodnie z ustawieniami regionalnymi (rysunek 34.6). Użytkownik może wybrać ustawienia regionalne w polu kombi zawierającym wszystkie takie ustawienia dostępne w systemie. Po uruchomieniu program ma wyświetlać kalendarz na bieżący miesiąc aktualnego roku. Użytkownik może przeglądać kalendarz za pomocą przycisków *Następny* i *Poprzedni*.

Oto główne etapy tworzenia tego programu:

1. Zdefiniuj klasę `CalendarPane` rozszerzającą klasę `BorderPane` (listing 34.4). Nowa klasa ma wyświetlać kalendarz na dany miesiąc określonego roku zgodnie z ustawieniami regionalnymi.



RYSUNEK 34.6. Program wyświetla kalendarz zgodnie z podanymi ustawieniami regionalnymi[2]

LISTING 34.4. CalendarPane.java

```
 1 import java.text.DateFormatSymbols;
 2 import java.text.SimpleDateFormat;
 3 import java.util.Calendar;
 4 import java.util.GregorianCalendar;
 5 import java.util.Locale;
 6 import javafx.geometry.Pos;
 7 import javafx.scene.control.Label;
 8 import javafx.scene.layout.BorderPane;
 9 import javafx.scene.layout.GridPane;
10 import javafx.scene.paint.Color;
11 import javafx.scene.text.TextAlignment;
12
13 public class CalendarPane extends BorderPane {
14   // Nagłówek
15   private Label lblHeader = new Label();
16
17   // Maksymalna liczba etykiet do wyświetlania nazw i numerów dni
18   private Label[] lblDay = new Label[49];
19
20   private Calendar calendar;
21   private int month; // Miesiąc
22   private int year; // Rok
23   private Locale locale = Locale.CHINA;
24
25   public CalendarPane() {
26     // Tworzenie etykiet do wyświetlania dni
27     for (int i = 0; i < 49; i++) {
28       lblDay[i] = new Label();
29       lblDay[i].setTextAlignment(TextAlignment.RIGHT);
30     }
```

[2] Technika opisana przez autora powoduje pobieranie polskich nazw miesięcy w dopełniaczu. Aby pobrać nazwę miesiąca w mianowniku, zastosuj następującą składnię: `Month.of(numer_miesiąca).getDisplayName(Text`
`↪Style.FULL_STANDALONE, Locale.forLanguageTag("pl-PL"));` — *przyp. tłum.*

```
31
32    showDayNames(); // Wyświetlanie nazw dni zgodnie z ustawieniami regionalnymi
33
34    GridPane dayPane = new GridPane();
35    dayPane.setAlignment(Pos.CENTER);
36
37    dayPane.setHgap(10);
38    dayPane.setVgap(10);
39    for (int i = 0; i < 49; i++) {
40      dayPane.add(lblDay[i], i % 7, i / 7);
41    }
42
43    // Umieszczanie nagłówka i głównej zawartości kalendarza w panelu
44    this.setTop(lblHeader);
45    BorderPane.setAlignment(lblHeader, Pos.CENTER);
46    this.setCenter(dayPane);
47
48    // Ustawianie bieżącego miesiąca i roku
49    calendar = new GregorianCalendar();
50    month = calendar.get(Calendar.MONTH);
51    year = calendar.get(Calendar.YEAR);
52    updateCalendar();
53
54    // Wyświetlanie kalendarza
55    showHeader();
56    showDays();
57  }
58
59  /** Aktualizowanie nazw dni zgodnie z ustawieniami regionalnymi */
60  private void showDayNames() {
61    DateFormatSymbols dfs = new DateFormatSymbols(locale);
62    String dayNames[] = dfs.getWeekdays();
63
64    // jlblDay[0], jlblDay[1], ..., jlblDay[6] zawierają nazwy dni
65    for (int i = 0; i < 7; i++) {
66      lblDay[i].setText(dayNames[i + 1]);
67    }
68  }
69
70  /** Aktualizowanie nagłówka na podstawie ustawień regionalnych */
71  private void showHeader() {
72    SimpleDateFormat sdf =
73      new SimpleDateFormat("MMMM yyyy", locale);
74    String header = sdf.format(calendar.getTime());
75    lblHeader.setText(header);
76  }
77
78  public void showDays() {
79    // Pobieranie dnia tygodnia przypadającego pierwszego dnia miesiąca
80    int startingDayOfMonth = calendar.get(Calendar.DAY_OF_WEEK);
81
82    // Zapełnianie kalendarza dniami poprzedzającymi bieżący miesiąc
83    Calendar cloneCalendar = (Calendar) calendar.clone();
84    cloneCalendar.add(Calendar.DATE, -1); // Uwzględnianie poprzedniego miesiąca
85    int daysInPrecedingMonth = cloneCalendar.getActualMaximum(
```

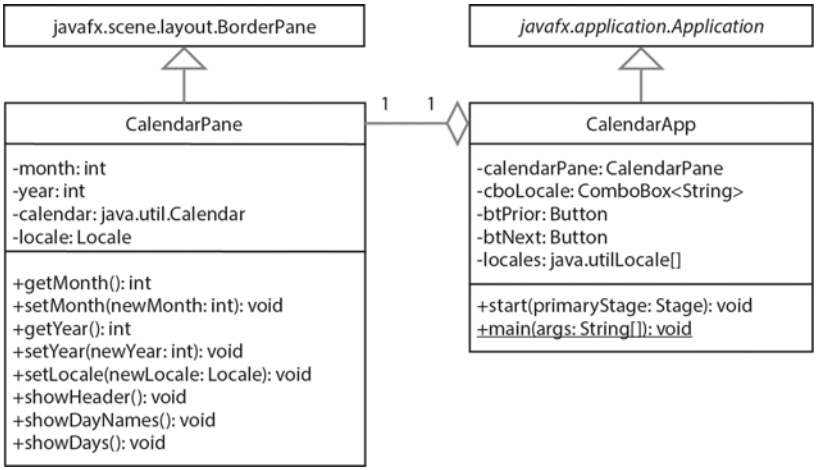
```
86        Calendar.DAY_OF_MONTH);
87
88      for (int i = 0; i < startingDayOfMonth - 1; i++) {
89        lblDay[i + 7].setTextFill(Color.LIGHTGRAY);
90        lblDay[i + 7].setText(daysInPrecedingMonth
91          - startingDayOfMonth + 2+ i + "");
92      }
93
94      // Wyświetlanie dni z bieżącego miesiąca
95      int daysInCurrentMonth = calendar.getActualMaximum(
96        Calendar.DAY_OF_MONTH);
97      for (int i = 1; i <= daysInCurrentMonth; i++) {
98        lblDay[i - 2 + startingDayOfMonth + 7].setTextFill(Color.BLACK);
99        lblDay[i - 2 + startingDayOfMonth + 7].setText(i + "");
100     }
101
102     // Uzupełnianie kalendarza dniami z następnego miesiąca
103     int j = 1;
104     for (int i = daysInCurrentMonth - 1 + startingDayOfMonth + 7;
105          i < 49; i++) {
106       lblDay[i].setTextFill(Color.LIGHTGRAY);
107       lblDay[i].setText(j++ + "");
108     }
109   }
110
111   /** Ustawianie kalendarza na pierwszy dzień
112     * podanego miesiąca określonego roku
113     */
114   public void updateCalendar() {
115     calendar.set(Calendar.YEAR, year);
116     calendar.set(Calendar.MONTH, month);
117     calendar.set(Calendar.DATE, 1);
118   }
119
120   public int getMonth() {
121     return month;
122   }
123
124   public void setMonth(int newMonth) {
125     month = newMonth;
126     updateCalendar();
127     showHeader();
128     showDays();
129   }
130
131   public int getYear() {
132     return year;
133   }
134
135   public void setYear(int newYear) {
136     year = newYear;
137     updateCalendar();
138     showHeader();
139     showDays();
140   }
141
```

```
142  public void setLocale(Locale locale) {
143    this.locale = locale;
144    updateCalendar();
145    showDayNames();
146    showHeader();
147    showDays();
148  }
149 }
```

2. Zdefiniuj aplikację `CalendarApp` (listing 34.5). Utwórz panel z obiektem typu `CalendarPane` pośrodku, dwoma przyciskami (*Poprzedni* i *Następny*) u dołu oraz polem kombi w górnej części. Zależności między używanymi klasami ilustruje rysunek 34.7.

LISTING 34.5. CalendarApp.java

```
 1 import java.util.Locale;
 2 import javafx.application.Application;
 3 import javafx.geometry.Pos;
 4 import javafx.scene.Scene;
 5 import javafx.scene.control.Button;
 6 import javafx.scene.control.ComboBox;
 7 import javafx.scene.control.Label;
 8 import javafx.scene.layout.BorderPane;
 9 import javafx.scene.layout.HBox;
10 import javafx.stage.Stage;
11
12 public class CalendarApp extends Application {
13   private CalendarPane calendarPane = new CalendarPane();
14   private Button btPrior = new Button("Poprzedni");
15   private Button btNext = new Button("Następny");
16   private ComboBox<String> cboLocales = new ComboBox<>();
17   private Locale[] availableLocales = Locale.getAvailableLocales();
18
19   @Override // Przesłanianie metody start z klasy Application
20   public void start(Stage primaryStage) {
21     HBox hBox = new HBox(5);
22     hBox.getChildren().addAll(btPrior, btNext);
23     hBox.setAlignment(Pos.CENTER);
24
25     // Inicjowanie pola cboLocales wszystkimi dostępnymi ustawieniami regionalnymi
26     setAvailableLocales();
27     HBox hBoxLocale = new HBox(5);
28     hBoxLocale.getChildren().addAll(
29       new Label("Ustawienia regionalne"), cboLocales);
30
31     BorderPane pane = new BorderPane();
32     pane.setCenter(calendarPane);
33     pane.setTop(hBoxLocale);
34     hBoxLocale.setAlignment(Pos.CENTER);
35     pane.setBottom(hBox);
36     hBox.setAlignment(Pos.CENTER);
37
38     // Tworzenie sceny i umieszczanie jej w oknie
39     Scene scene = new Scene(pane, 600, 300);
40     primaryStage.setTitle("CalendarApp"); // Ustawianie nagłówka okna
```



RYSUNEK 34.7. Klasa CalendarApp zawiera obiekt typu CalendarPane

```
41      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
42      primaryStage.show(); // Wyświetlanie okna
43
44      btPrior.setOnAction(e -> {
45        int currentMonth = calendarPane.getMonth();
46        if (currentMonth == 0) { // Poprzedni miesiąc to grudzień (11)
47          calendarPane.setYear(calendarPane.getYear() - 1);
48          calendarPane.setMonth(11);
49        }
50        else {
51          calendarPane.setMonth((currentMonth - 1) % 12);
52        }
53      });
54
55      btNext.setOnAction(e -> {
56        int currentMonth = calendarPane.getMonth();
57        if (currentMonth == 11) // Następny miesiąc to styczeń (0)
58          calendarPane.setYear(calendarPane.getYear() + 1);
59
60        calendarPane.setMonth((currentMonth + 1) % 12);
61      });
62
63      cboLocales.setOnAction(e ->
64        calendarPane.setLocale(availableLocales[cboLocales.
65          getSelectionModel().getSelectedIndex()]));
66    }
67
68    private void setAvailableLocales() {
69      for (int i = 0; i < availableLocales.length; i++)
70        cboLocales.getItems().add(availableLocales[i]
71          .getDisplayName() + " " + availableLocales[i].toString());
72
73      cboLocales.getSelectionModel().selectFirst();
74    }
75  }
```

Klasa `CalendarPane` służy do kontrolowania i wyświetlania kalendarza. Wyświetla ona miesiąc i rok w nagłówku oraz nazwy i numery dni w treści kalendarza. Nagłówek i nazwy dni są zależne od ustawień regionalnych.

Metoda `showHeader` (wiersze 71. – 76.) wyświetla nagłówek kalendarza w formacie `"MMMM rrrr"`. Klasa `SimpleDateFormat` używana w metodzie `showHeader` rozszerza klasę `DateFormat` i pozwala podać niestandardowy format daty, aby wyświetlić ją w nietypowy sposób.

Metoda `showDayNames` (wiersze 60. – 68.) wyświetla nazwy dni w kalendarzu. Klasa `DateFormatSymbols` używana w tej metodzie zawiera dane do formatowania dat i czasu zgodnie z ustawieniami regionalnymi, na przykład nazwy miesięcy, nazwy dni tygodnia i dane na temat stref czasowych. Metoda `getWeekdays` służy do pobierania tablicy nazw dni.

Metoda `showDays` (wiersze 60. – 68.) wyświetla dni z określonego miesiąca roku. Na rysunku 34.6 widać, że etykiety dni poprzedzających bieżący miesiąc są uzupełniane kilkoma ostatnimi dniami z poprzedniego miesiąca, a etykiety dni następujących po bieżącym miesiącu są uzupełniane kilkoma pierwszymi dniami z następnego miesiąca.

Aby uzupełnić kalendarz dniami z poprzedniego miesiąca, sklonuj obiekt `calendar`, zapisując go w obiekcie `cloneCalendar` (wiersz 83.). Obiekt `cloneCalendar` jest kopią obiektu `calendar` zajmującą odrębne miejsce w pamięci. Możesz więc zmieniać właściwości obiektu `cloneCalendar` bez modyfikowania obiektu `calendar`. Metoda `clone()` jest zdefiniowana w klasie `Object` i opisana w podrozdziale 13.7, „Interfejs Cloneable”. Można sklonować dowolny obiekt, jeśli jego klasa implementuje interfejs `Cloneable` (klasa `Calendar` implementuje ten interfejs).

Wywołanie `cloneCalendar.getActualMaximum(Calendar.DAY_OF_MONTH)` (wiersze 95. i 96.) zwraca liczbę dni danego miesiąca.

Klasa `CalendarApp` tworzy interfejs użytkownika i obsługuje zdarzenia związane z przyciskami oraz wybieranie ustawień regionalnych za pomocą pól kombi. Metoda `Locale.getAvailableLocales()` (wiersz 17.) służy do znajdowania wszystkich dostępnych ustawień regionalnych z obsługą kalendarza. Metoda `getDisplayName()` zwraca nazwy wszystkich ustawień regionalnych i dodaje je do pola kombi (wiersze 70. i 71.). Gdy użytkownik wybiera nazwę ustawień regionalnych w polu kombi, nowe ustawienia są przekazywane do panelu `calendarPane` i wyświetlany jest nowy kalendarz zgodnie z wybranymi ustawieniami (wiersze 63. – 65.).

34.3.1. Jak ustawić strefę czasową PST w obiekcie typu `Calendar`?

34.3.2. Jak wyświetlić bieżącą datę i czas z ustawieniami niemieckimi?

34.3.3. Jak użyć klasy `SimpleDateFormat` do wyświetlenia daty i czasu według wzorca `"yyyy.MM.dd hh:mm:ss"`?

34.3.4. W wierszu 66. listingu 34.2, *WorldClockControl.java*, do sortowania dostępnych stref czasowych używana jest metoda `Arrays.sort(availableTimeZones)`. Co się stanie, jeśli spróbujesz posortować ustawienia regionalne za pomocą wywołania `Arrays.sort(availableLocales)`?

34.4. Formatowanie liczb

Możesz formatować liczby na podstawie ustawień regionalnych.

Formatowanie liczb jest wysoce zależne od ustawień regionalnych. Na przykład wartość 5000,555 w Stanach Zjednoczonych jest wyświetlana jako `5,000.555`, we Francji i w Polsce jako `5 000,555`, a w Niemczech jako `5.000,555`.

Liczby są formatowane za pomocą klasy `java.text.NumberFormat`. Jest to abstrakcyjna klasa bazowa udostępniająca metody do formatowania i parsowania liczb (rysunek 34.8).

Za pomocą obiektu typu `NumberFormat` można formatować i parsować liczby zgodnie z dowolnymi ustawieniami regionalnymi. Kod jest wtedy niezależny od konwencji dotyczących separatora części ułamkowej, separatora tysięcy, formatu wartości pieniężnych i formatu wartości procentowych.

java.text.NumberFormat	
+getInstance(): NumberFormat	Zwraca domyślny format z domyślnymi ustawieniami regionalnymi
+getInstance(locale: Locale): NumberFormat	Zwraca domyślny format z podanymi ustawieniami regionalnymi
+getIntegerInstance(): NumberFormat	Zwraca format dla liczb całkowitych z domyślnymi ustawieniami regionalnymi
+getIntegerInstance(locale: Locale): NumberFormat	Zwraca format dla liczb całkowitych z podanymi ustawieniami regionalnymi
+getCurrencyInstance(): NumberFormat	Zwraca format dla walut z domyślnymi ustawieniami regionalnymi
+getNumberInstance(): NumberFormat	Działa jak `getInstance()`
+getNumberInstance(locale: Locale): NumberFormat	Działa jak `getInstance(ust_reg)`
+getPercentInstance(): NumberFormat	Zwraca format dla wartości procentowych z domyślnymi ustawieniami regionalnymi
+getPercentInstance(locale: Locale): NumberFormat	Zwraca format dla wartości procentowych z podanymi ustawieniami regionalnymi
+format (number: double): String	Formatuje liczbę zmiennoprzecinkową.
+format (number: long): String	Formatuje liczbę całkowitą
+getMaximumFractionDigits(): int	Zwraca maksymalną liczbę cyfr w części ułamkowej
+setMaximumFractionDigits(newValue: int): void	Ustawia maksymalną liczbę cyfr w części ułamkowej
+getMinimumFractionDigits(): int	Zwraca minimalną liczbę cyfr w części ułamkowej
+setMinimumFractionDigits(newValue: int): void	Ustawia minimalną liczbę cyfr w części ułamkowej
+getMaximumIntegerDigits(): int	Zwraca maksymalną liczbę cyfr w części całkowitej
+setMaximumIntegerDigits(newValue: int): void	Ustawia maksymalną liczbę cyfr w części całkowitej
+getMinimumIntegerDigits(): int	Zwraca minimalną liczbę cyfr w części całkowitej
+setMinimumIntegerDigits(newValue: int): void	Ustawia minimalną liczbę cyfr w części całkowitej
+isGroupingUsed(): boolean	Zwraca `true`, jeśli w danym formacie używane jest grupowanie. Na przykład dla angielskich ustawień regionalnych z włączonym grupowaniem 1234567 ma postać `"1,234,567"`
+setGroupingUsed(newValue: boolean): void	Określa, czy w danym formacie należy stosować formatowanie
+parse(source: String): Number	Parsuje łańcuch znaków na liczbę
+getAvailableLocales(): Locale[]	Pobiera zestaw ustawień regionalnych z formatami liczb

RYSUNEK 34.8. Klasa NumberFormat udostępnia metody do formatowania i parsowania liczb

34.4.1. Formatowanie zwykłych liczb

Obiekt typu `NumberFormat` dla bieżących ustawień regionalnych możesz pobrać za pomocą wywołania `NumberFor`
`↪mat.getInstance()` lub `NumberFormat.getNumberInstance`. Obiekt typu `NumberFormat` dla wybranych ustawień regionalnych zwraca wywołania `NumberFormat.getInstance(Locale)` i `NumberFormat.getNumberInstance(Locale)`. Następnie możesz wywołać metodę `format(liczba)` dla obiektu typu `NumberFormat`, aby zwrócić liczbę sformatowaną jako łańcuch znaków.

Na przykład aby wyświetlić liczbę 5000,555 z ustawieniami francuskimi, użyj następującego kodu:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(numberFormat.format(5000.555));
```

Wyświetlanie liczb możesz kontrolować za pomocą metod takich jak `setMaximumFractionDigits` i `setMinimum ↪FractionDigits`. Na przykład jeśli użyjesz formatu `numberFormat.setMaximumFractionDigits(1)`, wartość 5000,555 zostanie wyświetlona jako `5 000,6`.

34.4.2. Formatowanie wartości pieniężnych

Aby sformatować liczbę jako wartość pieniężną, użyj wywołania `NumberFormat.getCurrencyInstance()` (zwraca format dla bieżących ustawień regionalnych) lub `NumberFormat.getCurrencyInstance(Locale)` (zwraca format dla podanych ustawień regionalnych).

Jeśli chcesz wyświetlić liczbę 5000,555 jako wartość pieniężną dla ustawień amerykańskich, użyj następującego kodu:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(currencyFormat.format(5000.555));
```

Wartość `5000.555` zostanie sformatowana jako `$5,000.56`. Jeżeli użyjesz ustawień francuskich, liczba będzie miała format `5 000,56 €`.

34.4.3. Formatowanie wartości procentowych

Aby sformatować wartość procentową, użyj wywołania `NumberFormat.getPercentInstance()` (stosuje bieżące ustawienia regionalne) lub `NumberFormat.getPercentInstance(Locale)` (stosuje podane ustawienia regionalne).

W celu wyświetlenia liczby 0,555367 jako wartości procentowej z ustawieniami amerykańskimi użyj tego kodu:

```
NumberFormat percentFormat =
  NumberFormat.getPercentInstance(Locale.US);
System.out.println(percentFormat.format(0.555367));
```

Liczba `0.555367` zostanie sformatowana jako `56%`. Domyślnie część ułamkowa procentów jest przycinana. Jeśli chcesz zachować trzy cyfry po przecinku, użyj wywołania `percentFormat.setMinimumFractionDigits(3)`. Wtedy liczba `0.555367` zostanie sformatowana jako `55.537%`.

34.4.4. Parsowanie liczb

Możesz przekształcić liczbę na łańcuch znaków, używając metody `format(numericalValue)`. Możesz też wywołać metodę `parse(String)`, by przekształcić zwykłą liczbę, wartość pieniężną lub wartość procentową zgodnie z podanymi ustawieniami regionalnymi na obiekt typu `java.lang.Number`. Jeśli parsowanie zakończy się niepowodzeniem, metoda `parse` zwróci wyjątek `java.text.ParseException`. Na przykład wartość pieniężną w formacie amerykańskim `$5,000.56` można przekształcić na liczbę tak:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
try {
  Number number = currencyFormat.parse("$5,000.56");
  System.out.println(number.doubleValue());
}
catch (java.text.ParseException ex) {
  System.out.println("Parsowanie zakończone niepowodzeniem");
}
```

34.4.5. Klasa DecimalFormat

Jeśli chcesz uzyskać jeszcze większą kontrolę nad formatowaniem lub parsowaniem, zrzutuj uzyskany za pomocą metod fabrycznych obiekt typu `NumberFormat` na typ `java.text.DecimalFormat` (jest to podklasa klasy `NumberFormat`). Następnie użyj metody `applyPattern(String pattern)` z klasy `DecimalFormat`, aby podać wzorzec służący do wyświetlania liczby.

We wzorcu można określić minimalną liczbę cyfr w części całkowitej i maksymalną liczbę cyfr w części ułamkowej. Znaki `'0'` i `'#'` oznaczają wymaganą i opcjonalną cyfrę. Opcjonalne cyfry nie są wyświetlane, jeśli są równe zero. Na przykład wzorzec `"00.0##"` oznacza co najmniej dwie cyfry w części całkowitej i maksymalnie trzy cyfry w części ułamkowej. Jeśli w części całkowitej występować będą więcej niż dwie cyfry, wszystkie zostaną wyświetlone. Jeżeli w części ułamkowej znajdą się więcej niż trzy cyfry, wartość zostanie zaokrąglona. Gdy użyjesz wzorca `"00.0##"` i ustawień amerykańskich, liczba 111,2226 zostanie sformatowana jako `111.223`, liczba 1111,2226 jako `1111.223`, liczba 1,22 jako `01.22`, a liczba 1 jako `01.0`. Oto kod:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.US);
DecimalFormat decimalFormat = (DecimalFormat)numberFormat;
decimalFormat.applyPattern("00.0##");
System.out.println(decimalFormat.format(111.2226));
System.out.println(decimalFormat.format(1111.2226));
System.out.println(decimalFormat.format(1.22));
System.out.println(decimalFormat.format(1));
```

Na końcu wzorca można umieścić symbol `'%'`, aby określić, że liczbę należy sformatować jako wartość procentową. Wtedy liczba zostanie pomnożona przez 100 i dodany zostanie do niej znak procentów (`%`).

34.4.6. Przykład — formatowanie liczb

Utwórz kalkulator kredytów. Ma on umożliwiać wybór ustawień regionalnych i wyświetlanie liczb zgodnie z nimi. Na rysunku 34.9 pokazane jest, że użytkownik może wpisać stopę oprocentowania, liczbę lat spłaty i kwotę kredytu, a następnie kliknąć przycisk *Oblicz*, aby wyświetlić stopę oprocentowania jako wartość procentową, liczbę lat jako zwykłą wartość liczbową, a kwotę kredytu, łączną kwotę do spłaty i miesięczną ratę jako wartości pieniężne. Na listingu 34.6 pokazane jest rozwiązanie tego problemu.

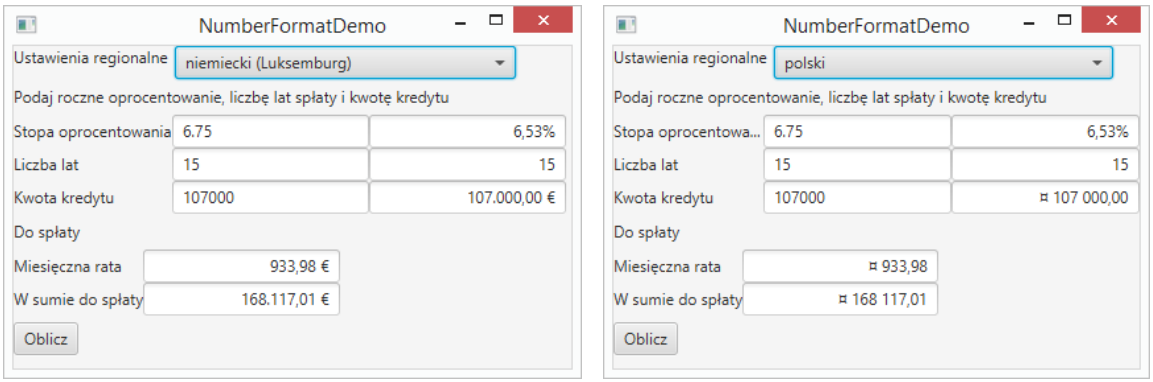
LISTING 34.6. NumberFormatDemo.java

```
 1 import java.util.*;
 2 import java.text.NumberFormat;
 3 import javafx.application.Application;
 4 import javafx.geometry.Pos;
 5 import javafx.scene.Scene;
 6 import javafx.scene.control.Button;
 7 import javafx.scene.control.ComboBox;
 8 import javafx.scene.control.Label;
 9 import javafx.scene.control.TextField;
10 import javafx.scene.layout.GridPane;
11 importjavafx.scene.layout.HBox;
12 importjavafx.scene.layout.VBox;
13 importjavafx.stage.Stage;
14
15 public class NumberFormatDemo extends Application {
16   // Pole kombi do wybierania dostępnych ustawień regionalnych
17   private ComboBox<String> cboLocale = new ComboBox<>();
18
```

```
19  // Pola tekstowe ze stopą oprocentowania, latami spłaty i kwotą kredytu
20  private TextField tfInterestRate = new TextField("6.75");
21  private TextField tfNumberOfYears = new TextField("15");
22  private TextField tfLoanAmount = new TextField("107000");
23  private TextField tfFormattedInterestRate = new TextField();
24  private TextField tfFormattedNumberOfYears = new TextField();
25  private TextField tfFormattedLoanAmount = new TextField();
26
27  // Pola tekstowe z miesięczną ratą i łączną kwotą do spłaty
28  private TextField tfTotalPayment = new TextField();
29  private TextField tfMonthlyPayment = new TextField();
30
31  // Przycisk Oblicz
32  private Button btCompute = new Button("Oblicz");
33
34  // Aktualne ustawienia regionalne
35  private Locale locale = Locale.getDefault();
36
37  // Deklaracja tablicy locales do przechowywania dostępnych ustawień regionalnych
38  private Locale locales[] = Calendar.getAvailableLocales();
39
40  /** Inicjowanie pola kombi */
41  public void initializeComboBox() {
42    // Dodawanie nazw ustawień regionalnych do pola kombi
43    for (int i = 0; i < locales.length; i++)
44      cboLocale.getItems().add(locales[i].getDisplayName());
45  }
46
47  @Override // Przesłanianie metody start z klasy Application
48  public void start(Stage primaryStage) {
49    initializeComboBox();
50
51    // Panel zawierający pole kombi do wybierania ustawień regionalnych
52    HBox hBox = new HBox(5);
53    hBox.getChildren().addAll(
54      new Label("Wybierz ustawienia regionalne"), cboLocale);
55
56    // Panel na dane wejściowe
57    GridPane gridPane = new GridPane();
58    gridPane.add(new Label("Stopa oprocentowania"), 0, 0);
59    gridPane.add(tfInterestRate, 1, 0);
60    gridPane.add(tfFormattedInterestRate, 2, 0);
61    gridPane.add(new Label("Liczba lat"), 0, 1);
62    gridPane.add(tfNumberOfYears, 1, 1);
63    gridPane.add(tfFormattedNumberOfYears, 2, 1);
64    gridPane.add(new Label("Kwota kredytu"), 0, 2);
65    gridPane.add(tfLoanAmount, 1, 2);
66    gridPane.add(tfFormattedLoanAmount, 2, 2);
67
68    // Panel na dane wyjściowe
69    GridPane gridPaneOutput = new GridPane();
70    gridPaneOutput.add(new Label("Miesięczna rata"), 0, 0);
71    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
72    gridPaneOutput.add(new Label("Kwota do spłaty"), 0, 1);
73    gridPaneOutput.add(tfTotalPayment, 1, 1);
```

```
74
75      // Wyrównywanie pól tekstowych
76      tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
77      tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
78      tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
79      tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
80      tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);
81
82      // Blokowanie możliwości edycji
83      tfFormattedInterestRate.setEditable(false);
84      tfFormattedNumberOfYears.setEditable(false);
85      tfFormattedLoanAmount.setEditable(false);
86      tfTotalPayment.setEditable(false);
87      tfMonthlyPayment.setEditable(false);
88
89      VBox vBox = new VBox(5);
90      vBox.getChildren().addAll(hBox,
91        new Label("Wpisz roczne oprocentowanie, " +
92          "liczbę lat spłaty i kwotę kredytu"), gridPane,
93        new Label("Do spłaty"), gridPaneOutput, btCompute);
94
95      // Tworzenie sceny i umieszczanie jej w oknie
96      Scene scene = newScene(vBox, 400, 300);
97      primaryStage.setTitle("NumberFormatDemo"); // Ustawianie nagłówka
98      primaryStage.setScene(scene); // Umieszczanie sceny w oknie
99      primaryStage.show(); // Wyświetlanie okna
100
101     // Rejestrowanie odbiorników
102     cboLocale.setOnAction(e -> {
103       locale = locales[cboLocale
104         .getSelectionModel().getSelectedIndex()];
105       computeLoan();
106     });
107
108     btCompute.setOnAction(e -> computeLoan());
109   }
110
111   /** Obliczanie rat i wyświetlanie wyników zgodnie z ustawieniami regionalnymi */
112   private void computeLoan() {
113     // Pobieranie danych wejściowych od użytkownika
114     double loan = new Double(tfLoanAmount.getText()).doubleValue();
115     double interestRate =
116       new Double(tfInterestRate.getText()).doubleValue() / 1240;
117     int numberOfYears =
118       new Integer(tfNumberOfYears.getText()).intValue();
119
120     // Obliczanie rat
121     double monthlyPayment = loan * interestRate/
122       (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
123     double totalPayment = monthlyPayment * numberOfYears * 12;
124
125     // Pobieranie obiektów formatujących
126     NumberFormat percentFormatter =
127       NumberFormat.getPercentInstance(locale);
128     NumberFormat currencyForm =
```

```
129       NumberFormat.getCurrencyInstance(locale);
130     NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
131     percentFormatter.setMinimumFractionDigits(2);
132
133     // Wyświetlanie sformatowanych danych wejściowych
134     tfFormattedInterestRate.setText(
135       percentFormatter.format(interestRate * 12));
136     tfFormattedNumberOfYears.setText
137       (numberForm.format(numberOfYears));
138     tfFormattedLoanAmount.setText(currencyForm.format(loan));
139
140     // Wyświetlanie wyników jako wartości pieniężnych
141     tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
142     tfTotalPayment.setText(currencyForm.format(totalPayment));
143   }
144 }
```



RYSUNEK 34.9. Ustawienia regionalne określają format liczb wyświetlanych w kalkulatorze kredytów

Metoda `computeLoan` (wiersze 112. – 143.) pobiera od użytkownika dane wejściowe (stopę oprocentowania, liczbę lat spłaty i kwotę kredytu), oblicza miesięczną ratę i łączną kwotę do spłaty i wyświetla roczne oprocentowanie jako wartość procentową, liczbę lat spłaty jako zwykłą liczbę oraz kwotę kredytu, miesięczną ratę i łączną kwotę do spłaty jako wartości pieniężne zgodnie z ustawieniami regionalnymi.

Instrukcja `percentFormatter.setMinimumFractionDigits(2)` (wiersz 131.) ustawia minimalną liczbę cyfr w części ułamkowej na 2. Bez tej instrukcji wartość `0.075` byłaby wyświetlana jako `7%` zamiast `7.5%`.

34.4.1. Napisz kod formatujący wartość 12345,678 zgodnie z ustawieniami brytyjskimi. Kod ma zachować dwie cyfry w części ułamkowej.

34.4.2. Napisz kod formatujący liczbę 12345,678 jako wartość pieniężną zgodnie z ustawieniami amerykańskimi.

34.4.3. Napisz kod formatujący liczbę 0,345678 jako wartość procentową z przynajmniej trzema cyframi w części ułamkowej.

34.4.4. Napisz kod parsujący tekst `3456,78` do postaci liczbowej.

34.4.5. Napisz kod, który za pomocą klasy `DecimalFormat` formatuje liczbę 12345,678 z użyciem wzorca `"0.0000#"`.

34.5. Pakiety zasobów

Aby zapisać niestandardowe informacje zależne od ustawień regionalnych, możesz użyć pakietów zasobów.

Klasa `NumberFormatDemo` z poprzedniego przykładu wyświetla liczby, wartości pieniężne i wartości procentowe zgodnie z ustawieniami regionalnymi. Jednak wszystkie łańcuchy znaków, nagłówki i etykiety przycisków są wyświetlane po polsku. W tym podrozdziale zobaczysz, jak używać pakietów zasobów do zapisywania lokalnych tekstów komunikatów, nagłówków, etykiet przycisków itd.

Pakiet zasobów jest plikiem klasy Javy lub plikiem tekstowym z informacjami zależnymi od ustawień regionalnych. Te informacje mogą być dynamicznie pobierane przez programy w Javie. Gdy potrzebne są zasoby zależne od ustawień regionalnych (na przykład tekst komunikatu), program może wczytać je z pakietu zasobów zgodnie z używanymi ustawieniami. Pozwala to pisać kod programu niezależny od ustawień regionalnych i odizolować w pakiecie zasobów większość (lub nawet całość) informacji zależnych od tych ustawień.

Dzięki pakietom zasobów można pisać programy, w których aspekty zależne od ustawień regionalnych są oddzielone od aspektów niezależnych od tych ustawień. W programach można wtedy łatwo obsługiwać różne ustawienia regionalne i dodawać obsługę nowych państw.

Zasoby należy umieszczać w klasach rozszerzających klasę `ResourceBundle` lub jedną z jej podklas. Pakiety zasobów zawierają pary *klucz-wartość*. Każdy klucz w unikatowy sposób identyfikuje zależny od ustawień regionalnych obiekt w zasobie. Za pomocą tego klucza można pobrać dany obiekt. `ListResourceBundle` jest wygodną podklasą klasy `ResourceBundle` często używaną do upraszczania tworzenia pakietów zasobów. Oto przykładowy pakiet zasobów zawierający cztery klucze (rozszerzana jest tu klasa `ListResourceBundle`):

```
// MyResource.java: plik zasobów
public class MyResource extends java.util.ListResourceBundle {
  static final Object[][] contents = {
    {"nationalFlag", "us.gif"},
    {"nationalAnthem", "us.au"},
    {"nationalColor", Color.red},
    {"annualGrowthRate", new Double(7.8)}
  };
  public Object[][] getContents() {
    return contents;
  }
}
```

Kluczami są łańcuchy znaków (wielkość znaków ma tu znaczenie). W tym przykładzie klucze to: `nationalFlag`, `nationalAnthem`, `nationalColor` i `annualGrowthRate`. Wartościami mogą być obiekty dowolnego typu.

Jeśli wszystkie zasoby to łańcuchy znaków, można je umieścić w wygodnym pliku tekstowym o rozszerzeniu *.properties*. Typowy plik tego rodzaju wygląda tak:

```
#Wed Jul 01 07:23:24 EST 1998
nationalFlag=us.gif
nationalAnthem=us.au
```

Aby w programie pobrać wartości z pakietu zasobów, najpierw trzeba utworzyć obiekt typu `ResourceBundle`, używając którejś z następujących metod statycznych:

```
public static final ResourceBundle getBundle(String baseName)
  throws MissingResourceException

public static final ResourceBundle getBundle
  (String baseName, Locale locale) throws MissingResourceException
```

Pierwsza z tych metod zwraca obiekt typu `ResourceBundle` dla domyślnych ustawień regionalnych. Druga zwraca obiekt typu `ResourceBundle` dla podanych ustawień regionalnych. Parametr `baseName` to nazwa bazowa zestawu klas opisujących informacje dla danych ustawień regionalnych. Nazwy tych klas znajdziesz w tabeli 34.3.

TABELA 34.3. Konwencje nazewnicze dla pakietów zasobów

1. NazwaBazowa_język_kraj_dialekt.class
2. NazwaBazowa_język_kraj.class
3. NazwaBazowa_język.class
4. NazwaBazowa.class
5. NazwaBazowa_język_kraj_dialekt.properties
6. NazwaBazowa_język_kraj.properties
7. NazwaBazowa_język.properties
8. NazwaBazowa.properties

Na przykład plik *MyResource_en_BR.class* przechowuje zasoby specyficzne dla Wielkiej Brytanii, plik *MyResource_en_US.class* zawiera zasoby specyficzne dla Stanów Zjednoczonych, a w pliku *MyResource_en.class* znajdują się zasoby dla wszystkich krajów anglojęzycznych.

Metoda `getBundle` próbuje wczytać klasę pasującą do ustawień regionalnych na podstawie języka, kraju i dialektu, szukając plików w kolejności podanej w tabeli 34.3. Pliki przeszukiwane w tej kolejności tworzą *sekwencję zasobów*. Jeśli metoda `getBundle` nie znajdzie odpowiedniego pliku w sekwencji zasobów, zgłosi wyjątek `Missing ↪ResourceException` (jest to podklasa klasy `RuntimeException`).

Po utworzeniu obiektu pakietu zasobów możesz użyć metody `getObject`, aby pobrać wartość na podstawie klucza. Oto przykład:

```
ResourceBundle res = ResourceBundle.getBundle("MyResource");
String flagFile = (String)res.getObject("nationalFlag");
String anthemFile = (String)res.getObject("nationalAnthem");
Color color = (Color)res.getObject("nationalColor");
  double growthRate = (Double)res.getObject("annualGrowthRate");
```

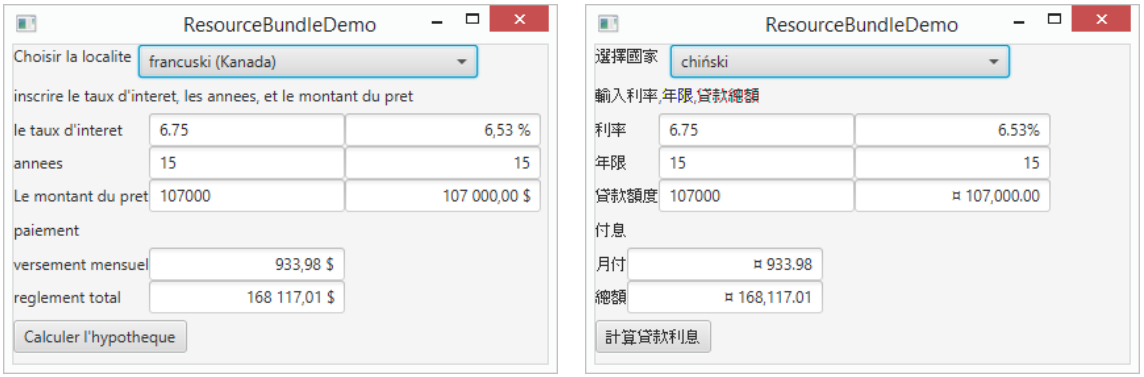
Wskazówka

Jeśli wartością jest łańcuch znaków, możesz użyć wygodnej metody `getString` zamiast `getObject`. Metoda `getString` rzutuje zwracaną przez metodę `getObject` wartość na typ `String`.

Co się stanie, jeśli szukany obiekt zasobu nie jest zdefiniowany w pakiecie zasobów? Java stosuje mechanizm inteligentnego wyszukiwania i szuka wtedy potrzebnego obiektu w plikach nadrzędnych w sekwencji zasobów. Ten proces jest kontynuowany do czasu znalezienia obiektu lub sprawdzenia wszystkich plików nadrzędnych. Nieudane wyszukiwanie skutkuje wyjątkiem `MissingResourceException`.

Zmodyfikuj teraz program `NumberFormatDemo`, aby wyświetlał komunikaty, nagłówki i etykiety przycisków w wielu językach (rysunek 34.10).

Wymaga to udostępnienia pakietu zasobów dla każdego uwzględnianego języka. Przyjmij, że program ma obsługiwać cztery języki: polski, angielski, chiński i francuski. Oto pakiet zasobów dla języka polskiego (*MyResource_pl.properties*):



RYSUNEK 34.10. Program wyświetla łańcuchy znaków w wielu językach

```
#MyResource_pl.properties dla języka polskiego
Number_Of_Years=Liczba lat
Total_Payment=Spłaty w sumie
Enter_Interest_Rate=Wpisz oprocentowanie, liczbę lat i kwotę kredytu
Payment=Do spłaty
Compute=Oblicz
Annual_Interest_Rate=Stopa oprocentowania
Loan_Amount=Kwota kredytu
Choose_a_Locale=Ustawienia regionalne
Monthly_Payment=Miesięczna rata
```

To pakiet dla języka angielskiego (*MyResource_en.properties*):

```
#MyResource.properties dla języka angielskiego
Number_Of_Years=Years
Total_Payment=Total Payment
Enter_Interest_Rate=Enter Interest Rate, Years, and Loan Amount
Payment=Payment
Compute=Compute
Annual_Interest_Rate=Interest Rate
Number_Formatting=ResourceBundleDemo
Loan_Amount=Loan Amount
Choose_a_Locale=Choose a Locale
Monthly_Payment=Monthly Payment
Icon=usIcon.gif
```

A to pakiet zasobów dla języka chińskiego (*MyResource_zh.properties*):

```
#MyResource_zh.properties dla języka chińskiego
Choose_a_Locale = \u9078\u64c7\u570b\u5bb6
Enter_Interest_Rate =
  \u8f38\u5165\u5229\u7387,\ u5e74\u9650,\ u8cbo8\u6b3e\u7e3d\u984d
Annual_Interest_Rate = \u5229\u7387
Number_Of_Years = \u5e74\u9650
Loan_Amount = \u8cbo8\u6b3e\u984d\u5ea6
Payment = \u4ed8\u606f
Monthly_Payment = \u6708\u4ed8
Total_Payment = \u7e3d\u984d
Compute = \u8a08\u7b97\u8cbo8\u6b3e\u5229\u606f
```

A to pakiet zasobów dla języka francuskiego (*MyResource_fr.properties*):

```
#MyResource_fr.properties dla języka francuskiego
Number_Of_Years=annees
Annual_Interest_Rate=le taux d'interet
Loan_Amount=Le montant du pret
Enter_Interest_Rate=inscrire le taux d'interet, les annees, et le
montant du pret
Payment=paiement
Compute=Calculer l'hypotheque
Number_Formatting=demonstration du formatting des chiffres
Choose_a_Locale=Choisir la localite
Monthly_Payment=versement mensuel
Total_Payment=reglement total
```

Plik pakietu zasobów należy umieścić w katalogu z daną klasą (na przykład *c:\kod* w przypadku programów z tej książki). Sam program pokazany jest na listingu 34.7.

LISTING 34.7. ResourceBundleDemo.java

```
 1 import java.util.*;
 2 import java.text.NumberFormat;
 3 import javafx.application.Application;
 4 import javafx.geometry.Pos;
 5 import javafx.scene.Scene;
 6 import javafx.scene.control.Button;
 7 import javafx.scene.control.ComboBox;
 8 import javafx.scene.control.Label;
 9 import javafx.scene.control.TextField;
10 import javafx.scene.layout.GridPane;
11 import javafx.scene.layout.HBox;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class ResourceBundleDemo extends Application {
16   private ResourceBundle res
17     = ResourceBundle.getBundle("MyResource");
18
19   // Tworzenie etykiet
20   private Label lblInterestRate =
21     new Label(res.getString("Annual_Interest_Rate"));
22   private Label lblNumberOfYears =
23     new Label(res.getString("Number_Of_Years"));
24   private Label lblLoanAmount =
25     new Label(res.getString("Loan_Amount"));
26   private Label lblMonthlyPayment =
27     new Label(res.getString("Monthly_Payment"));
28   private Label lblTotalPayment =
29     new Label(res.getString("Total_Payment"));
30   private Label lblPayment =
31     new Label(res.getString("Payment"));
32   private Label lblChooseALocale =
33     new Label(res.getString("Choose_a_Locale"));
34   private Label lblEnterInterestRate =
35     new Label(res.getString("Enter_Interest_Rate"));
36
```

```
37  // Pole kombi do wybierania dostępnych ustawień regionalnych
38  private ComboBox<String> cboLocale = new ComboBox<>();
39
40  // Pola tekstowe ze stopą oprocentowania, latami spłaty i kwotą kredytu
41  private TextField tfInterestRate = new TextField("6.75");
42  private TextField tfNumberOfYears = new TextField("15");
43  private TextField tfLoanAmount = new TextField("107000");
44  private TextField tfFormattedInterestRate = new TextField();
45  private TextField tfFormattedNumberOfYears = new TextField();
46  private TextField tfFormattedLoanAmount = new TextField();
47
48  // Pola tekstowe na ratę kredytu i łączną kwotę do spłaty
49  private TextField tfTotalPayment = new TextField();
50  private TextField tfMonthlyPayment = new TextField();
51
52  // Przycisk do uruchamiania obliczeń
53  private Button btCompute = new Button("Compute");
54
55  // Aktualne ustawienia regionalne
56  private Locale locale = Locale.getDefault();
57
58  // Deklarowanie tablicy locales do przechowywania dostępnych ustawień regionalnych
59  private Locale locales[] = Calendar.getAvailableLocales();
60
61  /** Inicjowanie pola kombi */
62  public voidinitializeComboBox() {
63    // Dodawanie nazw ustawień regionalnych w polu kombi
64    for (int i = 0; i < locales.length; i++)
65      cboLocale.getItems().add(locales[i].getDisplayName());
66  }
67
68  @Override // Przesłanianie metody start z klasy Application
69  public void start(Stage primaryStage) {
70    initializeComboBox();
71
72    // Panel z polem kombi do wybierania ustawień regionalnych
73    HBox hBox = new HBox(5);
74    hBox.getChildren().addAll(lblChooseALocale, cboLocale);
75
76    // Panel do przechowywania danych wejściowych
77    GridPane gridPane = new GridPane();
78    gridPane.add(lblInterestRate, 0, 0);
79    gridPane.add(tfInterestRate, 1, 0);
80    gridPane.add(tfFormattedInterestRate, 2, 0);
81    gridPane.add(lblNumberOfYears, 0, 1);
82    gridPane.add(tfNumberOfYears, 1, 1);
83    gridPane.add(tfFormattedNumberOfYears, 2, 1);
84    gridPane.add(lblLoanAmount, 0, 2);
85    gridPane.add(tfLoanAmount, 1, 2);
86    gridPane.add(tfFormattedLoanAmount, 2, 2);
87
88    // Panel do przechowywania danych wyjściowych
89    GridPane gridPaneOutput = new GridPane();
90    gridPaneOutput.add(lblMonthlyPayment, 0, 0);
91    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
```

```
92      gridPaneOutput.add(lblTotalPayment, 0, 1);
93      gridPaneOutput.add(tfTotalPayment, 1, 1);
94
95      // Wyrównywanie zawartości pól tekstowych
96      tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
97      tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
98      tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
99      tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
100     tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);
101
102     // Blokowanie możliwości edycji
103     tfFormattedInterestRate.setEditable(false);
104     tfFormattedNumberOfYears.setEditable(false);
105     tfFormattedLoanAmount.setEditable(false);
106     tfTotalPayment.setEditable(false);
107     tfMonthlyPayment.setEditable(false);
108
109     VBox vBox = new VBox(5);
110     vBox.getChildren().addAll(hBox, lblEnterInterestRate,
111       gridPane, lblPayment, gridPaneOutput, btCompute);
112
113     // Tworzenie sceny i umieszczanie jej w oknie
114     Scene scene = new Scene(vBox, 400, 300);
115     primaryStage.setTitle("ResourceBundleDemo"); // Ustawianie nagłówka okna
116     primaryStage.setScene(scene); // Umieszczanie sceny w oknie
117     primaryStage.show(); // Wyświetlanie okna
118
119     // Rejestrowanie odbiorników
120     cboLocale.setOnAction(e -> {
121       locale = locales[cboLocale
122         .getSelectionModel().getSelectedIndex()];
123       updateStrings();
124       computeLoan();
125     });
126
127     btCompute.setOnAction(e -> computeLoan());
128   }
129
130   /** Obliczanie rat i kwoty do spłaty oraz wyświetlanie wyników zgodnie z ustawieniami regionalnymi */
131   private void computeLoan() {
132     // Pobieranie danych wejściowych od użytkownika
133     double loan = new Double(tfLoanAmount.getText()).doubleValue();
134     double interestRate =
135       new Double(tfInterestRate.getText()).doubleValue() / 1240;
136     int numberOfYears =
137       new Integer(tfNumberOfYears.getText()).intValue();
138
139     // Obliczanie rat
140     double monthlyPayment = loan * interestRate/
141       (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
142     double totalPayment = monthlyPayment * numberOfYears * 12;
143
144     // Pobieranie obiektów typu NumberFormat
145     NumberFormat percentFormatter =
146       NumberFormat.getPercentInstance(locale);
```

```
147    NumberFormat currencyForm =
148      NumberFormat.getCurrencyInstance(locale);
149    NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
150    percentFormatter.setMinimumFractionDigits(2);
151
152    // Wyświetlanie sformatowanych danych wejściowych
153    tfFormattedInterestRate.setText(
154      percentFormatter.format(interestRate * 12));
155    tfFormattedNumberOfYears.setText
156      (numberForm.format(numberOfYears));
157    tfFormattedLoanAmount.setText(currencyForm.format(loan));
158
159    // Wyświetlanie wyników jako wartości pieniężnych
160    tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
161    tfTotalPayment.setText(currencyForm.format(totalPayment));
162  }
163
164  /** Aktualizowanie zasobów tekstowych */
165  private void updateStrings() {
166    res = ResourceBundle.getBundle("MyResource", locale);
167    lblInterestRate.setText(res.getString("Annual_Interest_Rate"));
168    lblNumberOfYears.setText(res.getString("Number_Of_Years"));
169    lblLoanAmount.setText(res.getString("Loan_Amount"));
170    lblTotalPayment.setText(res.getString("Total_Payment"));
171    lblMonthlyPayment.setText(res.getString("Monthly_Payment"));
172    btCompute.setText(res.getString("Compute"));
173    lblChooseALocale.setText(res.getString("Choose_a_Locale"));
174    lblEnterInterestRate.setText(
175      res.getString("Enter_Interest_Rate"));
176    lblPayment.setText(res.getString("Payment"));
177  }
178 }
```

Tekstowe pakiety zasobów są implementowane jako pliki tekstowe o rozszerzeniu *.properties* i umieszczane w tym samym miejscu co pliki klas programu. Obiekty typu `ListResourceBundle` są udostępniane jako pliki klas Javy. Ponieważ te ostatnie są implementowane za pomocą kodu źródłowego w Javie, nowe i zmodyfikowane obiekty tego rodzaju trzeba przed użyciem ponownie skompilować. Obiekty typu `PropertyResourceBundle` nie wymagają ponownej kompilacji po zmodyfikowaniu lub dodaniu tłumaczeń w aplikacji. Jednak klasa `ListResourceBundle` zapewnia znacznie wyższą wydajność niż `PropertyResourceBundle`.

Jeśli program nie znajdzie pakietu zasobów lub potrzebnego obiektu w pakiecie, zgłosi wyjątek `MissingResour`
`↪ceException`. Ponieważ `MissingResourceException` jest podklasą klasy `RuntimeException`, nie musisz bezpośrednio przechwytywać takich wyjątków w kodzie.

Ten program działa tak samo jak kod z listingu 34.6, *NumberFormatDemo.java*, ale zawiera kod do obsługi zasobów tekstowych. Metoda `updateString` (wiersze 165. – 177.) wyświetla łańcuchy znaków zgodnie z ustawieniami regionalnymi. Ta metoda jest wywoływana po wybraniu nowych ustawień regionalnych w polu kombi.

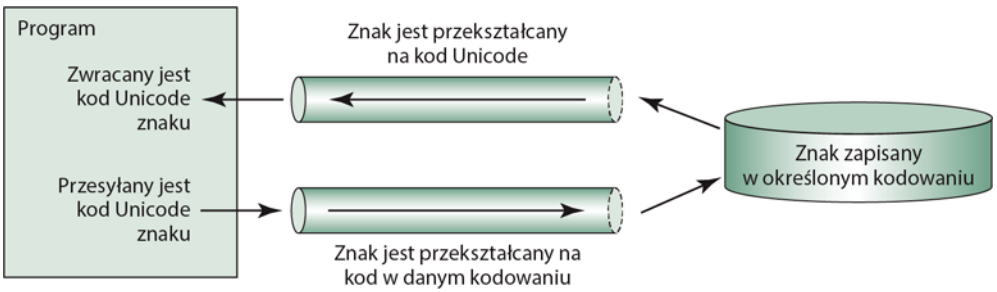
34.5.1. W jaki sposób metoda `getBundle` znajduje pakiet zasobów?

34.5.2. W jaki sposób metoda `getObject` znajduje zasób?

34.6. Kodowanie znaków

Możesz podać kodowanie plikowych operacji I/O, aby wczytywać i zapisywać znaki Unicode.

Programy w Javie używają kodowania Unicode. Gdy wczytujesz znak za pomocą tekstowych operacji I/O, zwracany jest kod Unicode tego znaku. W pliku dla znaków używane może być kodowanie inne niż Unicode. Java automatycznie przekształca znaki na kodowanie Unicode. Gdy zapisujesz znak za pomocą tekstowych operacji I/O, Java automatycznie konwertuje kod Unicode znaku na kodowanie używane w pliku. Ilustruje to rysunek 34.11.



RYSUNEK 34.11. Kodowanie w pliku może być inne niż kodowanie używane w programie

Kodowanie dla tekstowych operacji I/O możesz podać, używając konstruktora z klasy `Scanner` lub `PrintWriter`:

```
public Scanner(File file, String encodingName)
public PrintWriter(File file, String encodingName)
```

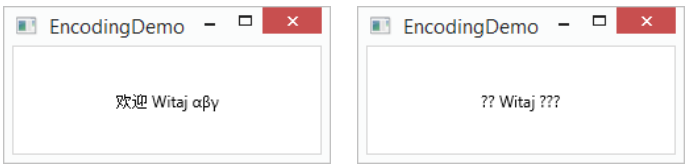
Listę schematów kodowania obsługiwanych w Javie znajdziesz na stronach *http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html* i *http://mindprod.com/jgloss/encoding.html*. Możesz na przykład użyć kodowania `GB18030` z uproszczonymi znakami chińskimi, `Big5` z tradycyjnymi znakami chińskimi, `Cp939` ze znakami japońskimi, `Cp933` ze znakami koreańskimi i `Cp838` ze znakami tajskimi.

Kod z listingu 34.8 tworzy plik z użyciem kodowania GB18030 (wiersz 8.). Tekst trzeba następnie wczytywać za pomocą tego samego kodowania (wiersz 12.). Dane wyjściowe przedstawia rysunek 34.12a.

LISTING 34.8. EncodingDemo.java

```
 1 import java.util.*;
 2 import java.io.*;
 3 import javafx.application.Application;
 4 import javafx.scene.Scene;
 5 import javafx.scene.layout.StackPane;
 6 import javafx.stage.Stage;
 7 import javafx.scene.text.Text;
 8
 9 public class EncodingDemo extends Application {
10   @Override // Przesłanianie metody start z klasy Application
11   public void start(Stage primaryStage) throws Exception {
12     try (
13       PrintWriter output = new PrintWriter("temp.txt", "GB18030");
14     ) {
15       output.print("\u6B22\u8FCE Witaj \u03b1\u03b2\u03b3");
16     }
17
```

```
18      try (
19        Scanner input = new Scanner(new File("temp.txt"), "GB18030");
20      ) {
21        StackPane pane = new StackPane();
22        pane.getChildren().add(new Text(input.nextLine()));
23
24        // Tworzenie sceny i umieszczanie jej w oknie
25        Scene scene = new Scene(pane, 200, 200);
26        primaryStage.setTitle("EncodingDemo"); // Ustawianie nagłówka okna
27        primaryStage.setScene(scene); // Umieszczanie sceny w oknie
28        primaryStage.show(); // Wyświetlanie okna
29      }
30    }
31  }
```



Rysunek 34.12. Możesz wybrać schemat kodowania dla pliku

Jeśli w wierszach 13. i 19. nie podasz kodowania, używane będzie kodowanie domyślne z systemu. W Stanach Zjednoczonych domyślnie używane jest kodowanie ASCII, gdzie kody mają po 8 bitów. W Javie używane jest 16-bitowe kodowanie Unicode. Jeśli kod Unicode nie jest zgodny z kodem ASCII, w pliku zapisywany jest znak `'?'`. Dlatego gdy zapiszesz w pliku ASCII sekwencję `\u6B22`, w pliku pojawi się znak ?. Po ponownym wczytaniu pliku zobaczysz znak ? (rysunek 34.12b).

Aby wyświetlić domyślne kodowanie systemu, użyj następującej instrukcji:

```
System.out.println(System.getProperty("file.encoding"));
```

W systemie Windows domyślnie używane jest kodowanie `Cp1252` (jest to odmiana kodowania ASCII).

34.6.1. Jak podać schemat kodowania na potrzeby pliku tekstowego?
34.6.2. Co się stanie, jeśli zapiszesz znak Unicode w pliku tekstowym w formacie ASCII?
34.6.3. Jak sprawdzić nazwę domyślnego kodowania z systemu?

Najważniejsze pojęcia

ustawienia regionalne
pakiet zasobów
schemat kodowania pliku

Podsumowanie rozdziału

1. Java jest pierwszym językiem zaprojektowanym od podstaw z myślą o obsłudze umiędzynarodawiania. Dzięki temu można dostosować program do wielu państw i języków bez konieczności wprowadzania skomplikowanych zmian w kodzie.

2. W programach w Javie używane są znaki *Unicode*. Kodowanie Unicode ułatwia pisanie w Javie programów operujących łańcuchami w dowolnym języku.
3. Java udostępnia klasę `Locale` zawierającą informacje o danych ustawieniach regionalnych. Obiekt typu `Locale` określa, w jaki sposób wyświetlane mają być informacje zależne od ustawień regionalnych (na przykład daty, czas i liczby) oraz jak wykonywać operacje zależne od takich ustawień (na przykład sortowanie łańcuchów znaków). Klasy do formatowania dat, czasu i liczb oraz sortowania łańcuchów znaków znajdują się w pakiecie `java.text`.
4. W różnych ustawieniach regionalnych używane są inne konwencje wyświetlania dat i czasu. Do formatowania dat i czasu zgodnie z ustawieniami regionalnymi możesz użyć klasy `java.text.DataFormat` i jej podklas.
5. Aby sformatować liczbę zgodnie z domyślnymi lub określonymi ustawieniami regionalnymi, użyj jednej z metod fabrycznych z klasy `NumberFormat`; otrzymasz wtedy obiekt formatujący. Metody `getInstance` i `getNumberInstance` zwracają obiekt formatujący dla zwykłych liczb, metoda `getCurrencyInstance` zwraca obiekt formatujący dla wartości pieniężnych, a metoda `getPercentInstance` zwraca obiekt formatujący dla wartości procentowych.
6. W Javie dostępna jest klasa `ResourceBundle` do wydzielania od programu informacji specyficznych dla ustawień regionalnych (na przykład komunikatów o stanie i etykiet komponentów GUI). Te informacje są zapisywane niezależnie od kodu źródłowego i mogą być używane oraz wczytywane dynamicznie w czasie wykonywania programu za pomocą obiektów typu `ResourceBundle` (nie trzeba ich zapisywać na stałe w programie).
7. Gdy tworzysz obiekt typu `PrintWriter` lub `Scanner`, możesz podać kodowanie pliku tekstowego.

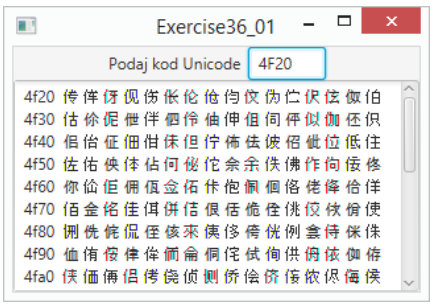
Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

Ćwiczenia programistyczne

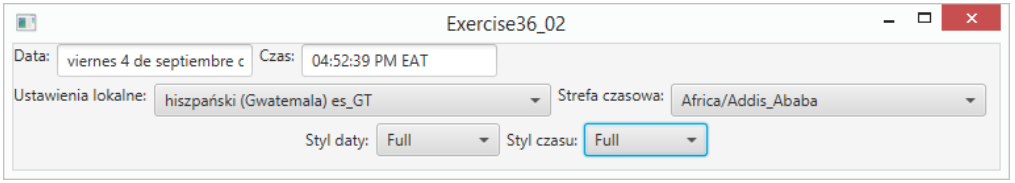
Podrozdziały 34.1 – 34.2

***34.1.** *Przeglądarka znaków Unicode*. Napisz aplikację z GUI, która wyświetla znaki Unicode (rysunek 34.13). Użytkownik ma podać kod Unicode w pliku tekstowym i wcisnąć klawisz *Enter*, aby wyświetlić zestaw znaków Unicode począwszy od podanego kodu. Znaki Unicode należy wyświetlać w 20-wierszowym obszarze tekstowym z możliwością przewijania. Program w każdym wierszu ma wyświetlać 16 znaków poprzedzonych kodem Unicode pierwszego znaku z danego wiersza.



Rysunek 34.13. Program wyświetlający znaki Unicode

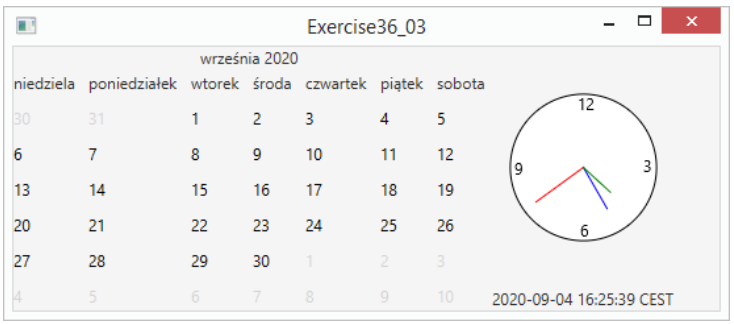
**34.2. *Wyświetlanie daty i czasu.* Napisz program, który wyświetla bieżącą datę i czas w sposób pokazany na rysunku 34.14. Program ma umożliwiać wybranie w polach kombi ustawień regionalnych, strefy czasowej, stylu dla daty i stylu dla czasu.



Rysunek 34.14. Program wyświetla bieżącą datę i czas

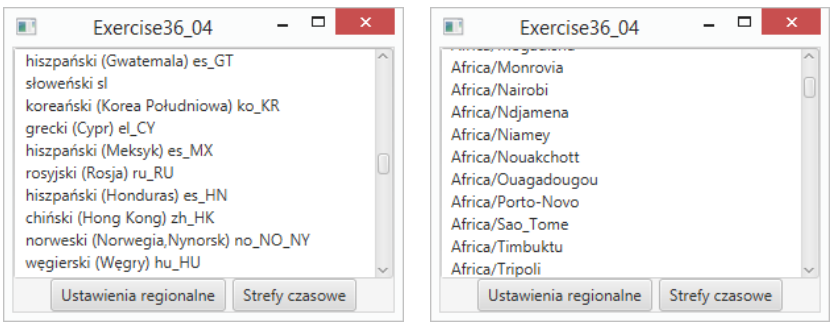
Podrozdział 34.3

34.3. *Umieszczanie kalendarza i zegara w panelu.* Napisz program, który wyświetla bieżącą datę w kalendarzu i bieżący czas na zegarze (rysunek 34.15). Program ma działać jako samodzielna aplikacja.



Rysunek 34.15. Kalendarz i zegar wyświetlają bieżącą datę i aktualny czas

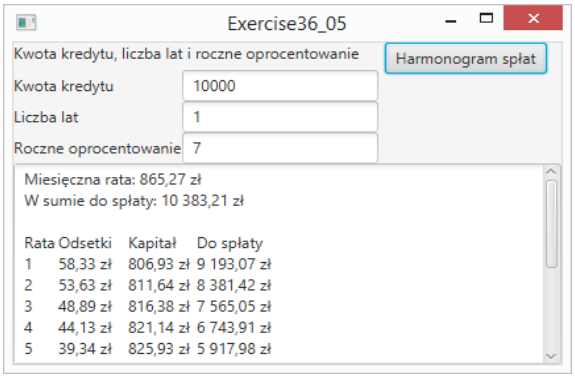
34.4. *Znajdowanie identyfikatorów dostępnych ustawień regionalnych i stref czasowych.* Napisz dwa programy do wyświetlania identyfikatorów dostępnych ustawień regionalnych i stref czasowych za pomocą przycisków (rysunek 34.16).



Rysunek 34.16. Program po kliknięciu przycisku wyświetla dostępne ustawienia regionalne i strefy czasowe

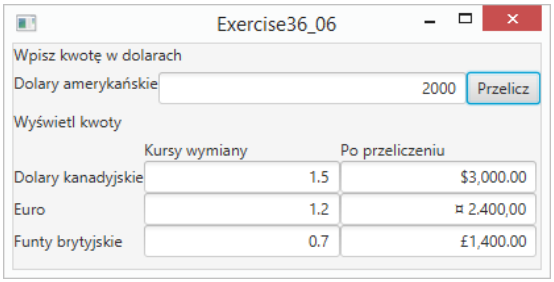
Podrozdział 34.4

***34.5.** *Obliczanie harmonogramu spłat kredytu*. Zmodyfikuj rozwiązanie ćwiczenia 4.22, używając architektury JavaFX (rysunek 34.17). Program ma umożliwiać użytkownikowi wpisanie kwoty kredytu, okresu kredytowania i stopy oprocentowania, a zwracać część odsetkową, część kapitałową i wartość do spłaty sformatowane jako wartości pieniężne.



Rysunek 34.17. Program wyświetlający harmonogram spłat kredytu

34.6. *Przeliczanie dolarów na inne waluty*. Napisz program, który przelicza dolary amerykańskie na dolary kanadyjskie, euro i funty brytyjskie (rysunek 34.18). Użytkownik ma wpisać wartość w dolarach i kurs wymiany, a następnie kliknąć przycisk *Przelicz*, aby wyświetlić wartość po konwersji.



Rysunek 34.18. Program przelicza dolary amerykańskie na dolary kanadyjskie, euro i funty brytyjskie

34.7. *Obliczanie rat kredytu*. Zmodyfikuj listing 2.8, *ComputeLoan.java*, aby kod wyświetlał miesięczną ratę i łączną kwotę do spłaty jako wartości pieniężne.

34.8. *Używanie klasy* `DecimalFormat`. Zmodyfikuj rozwiązanie ćwiczenia 5.8, aby wyświetlać temperaturę z najwyżej dwiema cyframi w części ułamkowej. Użyj do tego klasy `DecimalFormat`.

Podrozdział 34.5

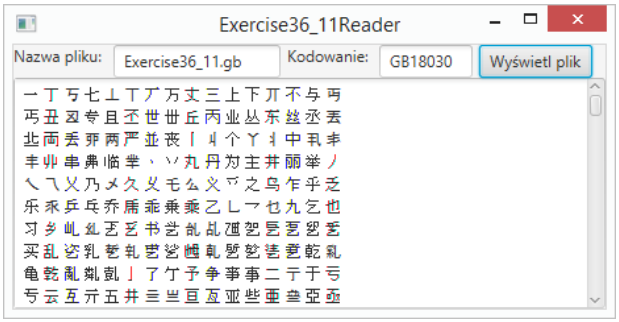
***34.9.** *Używanie pakietów zasobów*. Zmodyfikuj kod wyświetlający kalendarz z punktu 34.3.6, „Przykład — wyświetlanie kalendarza". Nowa wersja ma udostępniać pakiet zasobów z etykietami *Wybierz ustawienia regionalne* i *Kalendarz* w językach francuskim, niemieckim, chińskim lub w innych wybranych przez Ciebie językach.

**34.10. *Flaga i hymn.* Zmodyfikuj listing 16.13, *ImageAudioAnimation.java*, używając pakietu zasobów do pobierania plików graficznych i dźwiękowych.

Wskazówka: po wybraniu nowego kraju zapisz powiązane z nim ustawienia regionalne. Program powinien szukać w pliku zasobów flagi i hymnu dla tych ustawień regionalnych.

Podrozdział 34.6

**34.11. *Określanie kodowania w pliku.* Napisz program `Exercise36_11Writer`, który zapisuje 1307 × 16 chińskich znaków Unicode (począwszy od `\uE00`) w pliku *Exercise36_11.gb*, używając kodowania GB18030. Wyświetlaj po 16 znaków w wierszu i rozdzielaj znaki spacjami. Napisz program `Exercise36_11Reader`, który wczytuje wszystkie znaki z pliku za pomocą podanego kodowania. Program z rysunku 34.19 wyświetla plik z użyciem kodowania GB18030.



RYSUNEK 34.19. Program wyświetla plik, używając określonego kodowania

ROZDZIAŁ

35

Drzewa 2-3-4 i B-drzewa

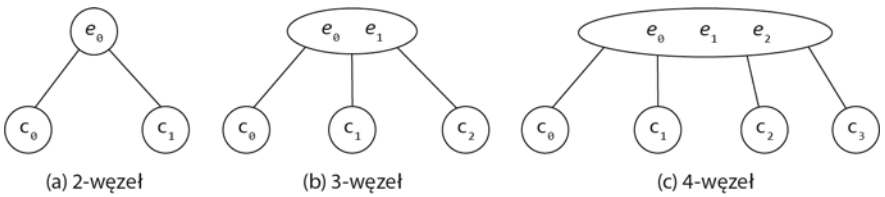
Cele

- Wyjaśnienie, czym są drzewa 2-3-4 (podrozdział 35.1).
- Zaprojektowanie klasy `Tree24` implementującej interfejs `Tree` (podrozdział 35.2).
- Wyszukiwanie elementów w drzewach 2-3-4 (podrozdział 35.3).
- Wstawianie elementów w drzewach 2-3-4 i wyjaśnienie podziału węzła (podrozdział 35.4).
- Usuwanie elementów w drzewach 2-3-4 i omówienie operacji przeniesienia i złączania (podrozdział 35.5).
- Przechodzenie drzew 2-3-4 (podrozdział 35.6).
- Zaimplementowanie i przetestowanie klasy `Tree24` (podrozdziały 35.7 – 35.8).
- Przeanalizowanie złożoności drzew 2-3-4 (podrozdział 35.9).
- Używanie B-drzew do indeksowania dużych zbiorów danych (podrozdział 35.10).

35.1. Wprowadzenie

Drzewo 2-3-4 to w pełni zrównoważone drzewo wyszukiwań, w którym wszystkie liście znajdują się na tym samym poziomie[1].

W drzewie 2-3-4 węzeł może zawierać jeden, dwa lub trzy elementy. Wewnętrzny *2-węzeł* zawiera jeden element i dwoje dzieci. Wewnętrzny *3-węzeł* ma dwa elementy i troje dzieci. Wewnętrzny *4-węzeł* ma trzy elementy i czworo dzieci (rysunek 35.1).

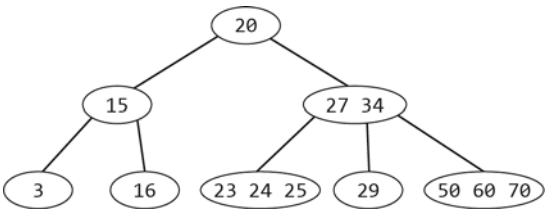


Rysunek 35.1. Wewnętrzne węzły drzewa 2-3-4 mają dwoje, troje lub czworo dzieci

Każde dziecko jest poddrzewem 2-3-4 (które może być puste). Korzeń nie ma rodzica, a liście nie mają dzieci. Elementy w takim drzewie są unikatowe i uporządkowane w następujący sposób:

$$E(c_0) < e_0 < E(c_1) < e_1 < E(c_2) < e_2 < E(c_3)$$

gdzie $E(c_k)$ oznacza elementy z c_k. Na rysunku 35.2 pokazane jest przykładowe drzewo 2-3-4. c_k jest nazywane *lewym poddrzewem* e_k, a c_{k+1} to *prawe poddrzewo* e_k.



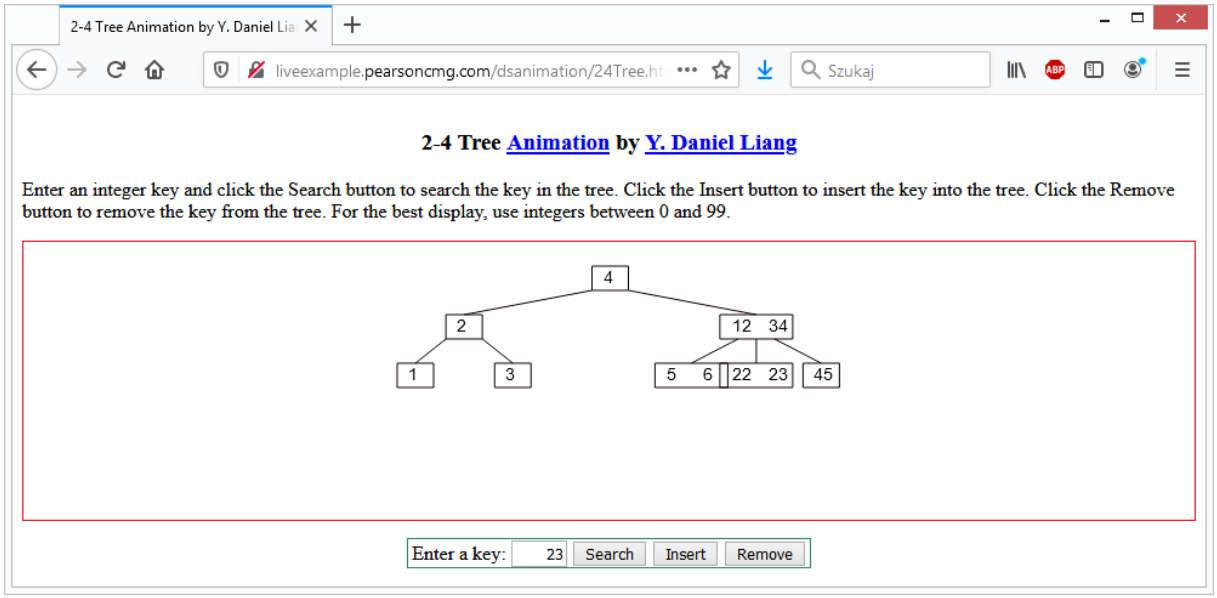
Rysunek 35.2. Drzewo 2-3-4 jest kompletnym drzewem wyszukiwań

W drzewie binarnym każdy węzeł zawiera jeden element. Drzewo 2-3-4 jest zwykle krótsze od analogicznego drzewa binarnego, ponieważ w drzewie 2-3-4 węzeł może obejmować także dwa lub trzy elementy.

Uwaga edukacyjna

Otwórz stronę *http://liveexample.pearsoncmg.com/dsanimation/24Tree.html*, aby zobaczyć działanie drzew 2-3-4 (rysunek 35.3).

[1] W witrynie poświęconej oryginalnemu wydaniu książki jest to rozdział 42. — *przyp. tłum.*



RYSUNEK 35.3. Narzędzie do wyświetlania animacji umożliwia wstawianie, usuwanie i wyszukiwanie elementów drzew 2-3-4

35.2. Projektowanie klas na potrzeby drzew 2-3-4

Klasa `Tree24` będzie definiować drzewa 2-3-4 i udostępniać metody do wyszukiwania, wstawiania i usuwania elementów.

Na rysunku 35.4 opisana jest klasa `Tree24` implementująca interfejs `Tree`. Ten interfejs został zdefiniowany na listingu 27.3, *Tree.java*. Klasa `Tree24Node` definiuje trzy węzły. Elementy węzła są przechowywane na liście `elements`, a wskaźniki do dzieci są zapisane na liście `child` (rysunek 35.5).

35.2.1. Czym jest drzewo 2-3-4? Czym są 2-węzeł, 3-węzeł i 4-węzeł?

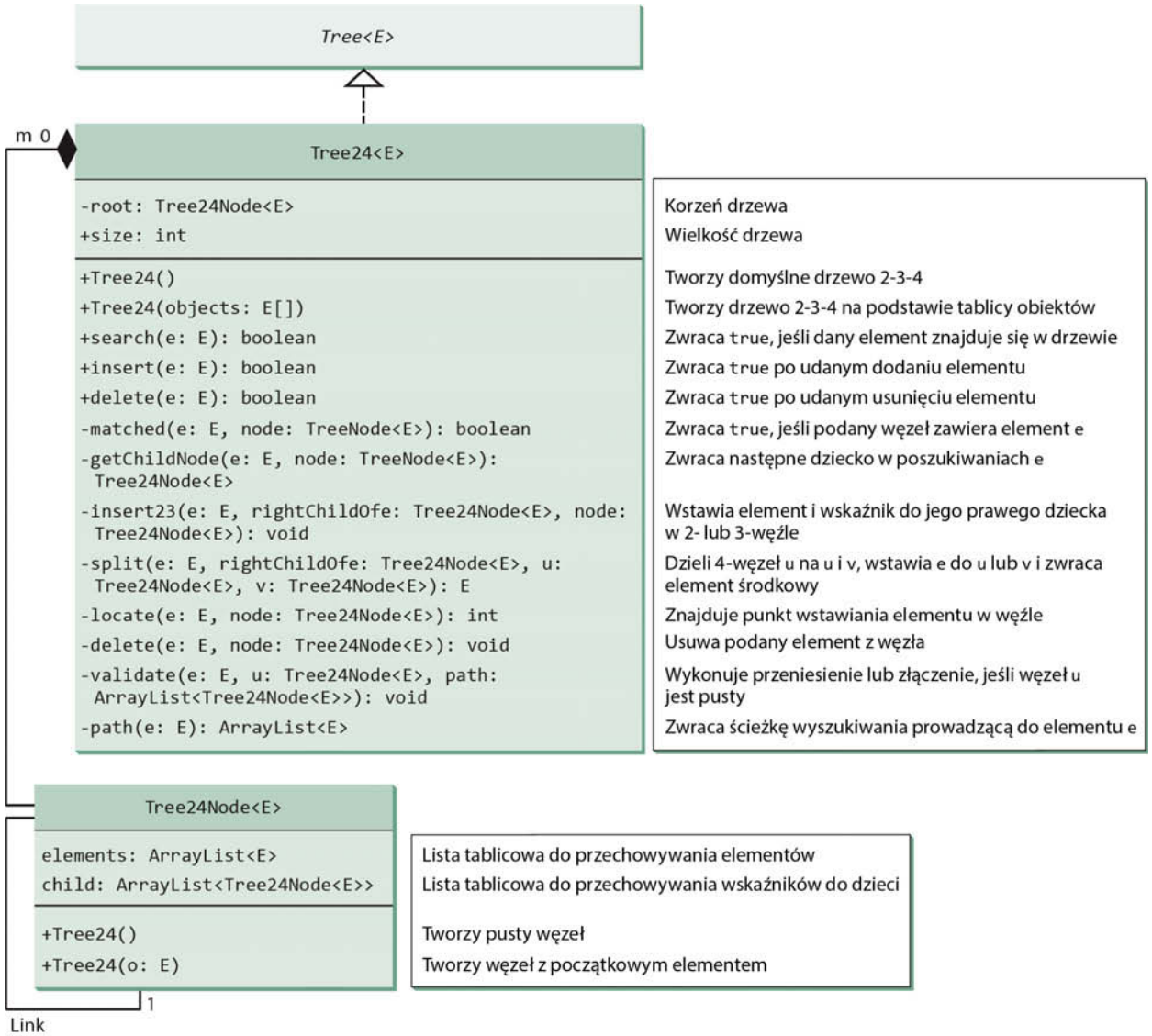
35.2.2. Opisz pola klasy `Tree24` i pola klasy `Tree24Node`.

35.2.3. Jaka jest minimalna liczba elementów drzewa 2-3-4 o wysokości 5? Jaka jest maksymalna liczba elementów drzewa 2-3-4 o wysokości 5?

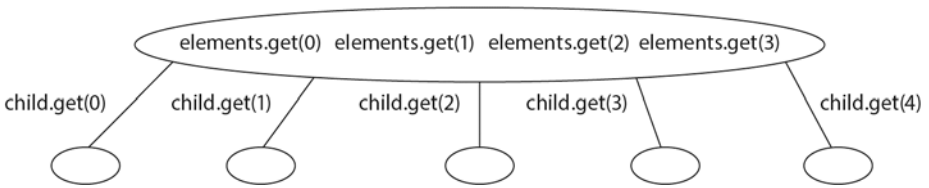
35.3. Wyszukiwanie elementu

Wyszukiwanie elementu w drzewie 2-3-4 przypomina wyszukiwanie elementu w drzewie binarnym. Różnica polega na tym, że w drzewie 2-3-4 oprócz szukania elementów wzdłuż ścieżki trzeba szukać ich w węzłach.

Aby znaleźć element w drzewie 2-3-4, należy zacząć od korzenia i zmierzać w dół. Jeśli elementu nie ma w danym węźle, należy przejść do odpowiedniego poddrzewa. Proces ten jest powtarzany do czasu znalezienia pasującego elementu lub dotarcia do pustego poddrzewa. Opis algorytmu znajdziesz na listingu 35.1.



RYSUNEK 35.4. Klasa Tree24 implementuje interfejs Tree



RYSUNEK 35.5. Węzeł drzewa 2-3-4 przechowuje elementy i wskaźniki do dzieci na listach tablicowych

LISTING 35.1. Wyszukiwanie elementu w drzewie 2-3-4

```
1 boolean search(E e) {
2   current = root; // Rozpoczynanie od korzenia
3
4   while (current != null) {
5     if (match(e, current)) { // Element znajduje się w węźle
6       return true; // Element został znaleziony
7     }
8     else {
9       current = getChildNode(e, current); // Wyszukiwanie w poddrzewie
10    }
11  }
12  return false; // Elementu nie ma w drzewie
13 }
```

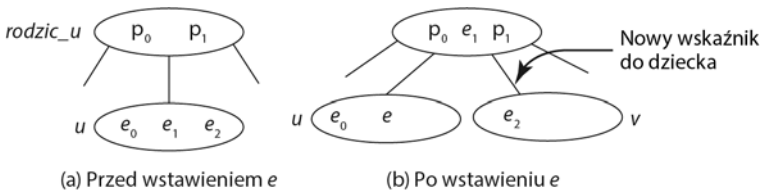
Metoda `match(e, current)` sprawdza, czy element `e` znajduje się w bieżącym węźle. Metoda `getChildNode(e, current)` zwraca korzeń poddrzewa na potrzeby dalszego wyszukiwania. Początkowo `current` prowadzi do korzenia (wiersz 2.). Należy powtarzać wyszukiwanie elementu w bieżącym węźle do czasu, gdy `current` będzie równe `null` (wiersz 4.) lub element zostanie znaleziony w danym węźle.

35.4. Wstawianie elementu w drzewie 2-3-4

Wstawianie elementu wymaga znalezienia liścia i wstawienia w nim elementu.

Aby wstawić element *e* w drzewie 2-3-4, należy znaleźć liść, w którym ten element ma się znaleźć. Jeśli liściem jest 2- lub 3-węzeł, wystarczy wstawić w nim element. W 4-węźle wstawienie nowego elementu spowoduje *przepełnienie*. Należy wtedy przeprowadzić operację *podziału*:

- Niech *u* będzie 4-węzłem, który jest *liściem*, gdzie należy wstawić dany element. Niech *rodzic_u* (`parentOfu` w kodzie) będzie rodzicem węzła *u* (rysunek 35.6a).
- Utwórz nowy węzeł *v* i przenieś e_2 do *v*.
- Jeśli $e < e_1$, wstaw *e* do *u*; w przeciwnym razie wstaw *e* do *v*. Jeśli $e_0 < e < e_1$, to *e* jest wstawiane do *u* (rysunek 35.6b).
- Wstaw e_1 wraz z jego prawym dzieckiem (*v*) do rodzica (rysunek 35.6b).

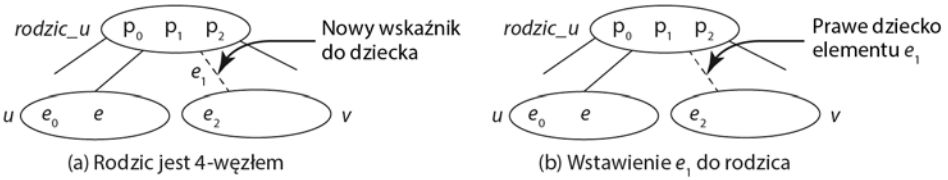


RYSUNEK 35.6. Operacja podziału tworzy nowy węzeł i wstawia środkowy element do rodzica

Na rysunku 35.6 rodzic jest 3-węzłem. Jest więc miejsce na wstawienie *e* do rodzica. Co się stanie, jeśli rodzic będzie 4-węzłem, tak jak na rysunku 35.7? Wymaga to podziału rodzica. Ten proces przebiega tak samo jak podział 4-węzła będącego liściem, przy czym trzeba wstawić element wraz z jego prawym dzieckiem.

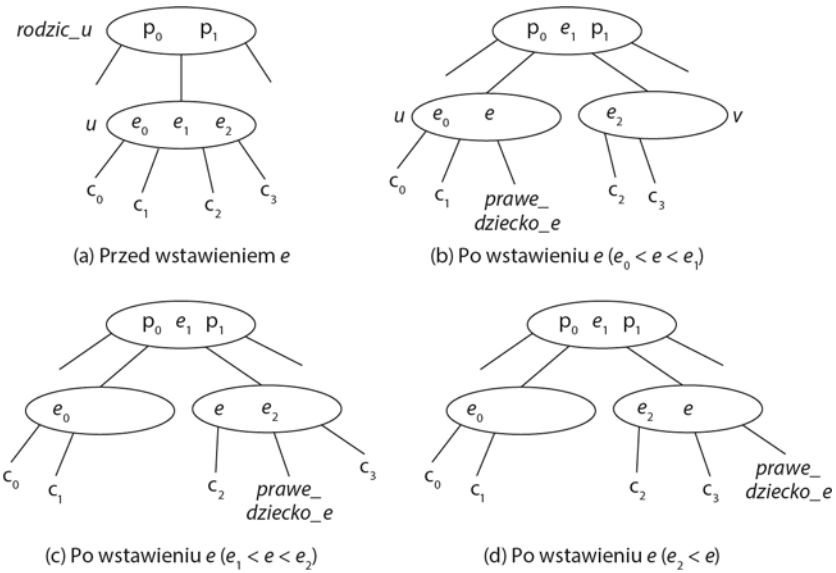
Ten algorytm można zmodyfikować:

- Niech *u* będzie 4-węzłem (*liściem lub nieliściem*), w którym należy wstawić element. Niech *rodzic_u* będzie rodzicem *u* (rysunek 35.8a).



Rysunek 35.7. Jeśli rodzicem jest 4-węzeł, proces wstawiania należy kontynuować

- Utwórz nowy węzeł, *v*, i przenieś do niego e_2 oraz jego dzieci c_2 i c_3.
- Jeśli $e < e_1$, wstaw *e* i wskaźnik do jego prawego dziecka do *u*; w przeciwnym razie wstaw *e* wraz ze wskaźnikiem do prawego dziecka do *v*. Rysunki 35.8.b, c i d reprezentują sytuacje $e_0 < e < e_1$, $e_1 < e < e_2$ i $e_2 < e$.
- Rekurencyjnie wstaw e_1 wraz z jego prawym dzieckiem (*v*) do rodzica.



Rysunek 35.8. Węzeł wewnętrzny można podzielić, aby wyeliminować przepełnienie

Algorytm wstawiania elementu przedstawiony jest na listingu 35.2.

Listing 35.2. Wstawianie elementu w drzewie 2-3-4

```
1 public boolean insert(E e) {
2   if (root == null)
3     root = new Tree24Node<E>(e); // Tworzenie nowego korzenia dla elementu
4   else {
5     znajdź leafNode, gdzie należy wstawić e
6     insert(e, null, leafNode); // Prawe dziecko elementu e to null
7   }
8
9   size++; // Zwiększanie wielkości
10  return true; // Element został wstawiony
```

```
11 }
12
13 private void insert(E e, Tree24Node<E> rightChildOfe,
14     Tree24Node<E> u) {
15   if (u jest 2- lub 3-węzłem) { // u jest 2- lub 3-węzłem
16     insert23(e, rightChildOfe, u); // Wstawianie e do węzła u
17   }
18   else { // Podział 4-węzła u
19     Tree24Node<E> v = new Tree24Node<E>(); // Tworzenie nowego węzła
20     E median = split(e, rightChildOfe, u, v); // Podział u
21
22     if (u == root) { // u jest korzeniem
23       root = new Tree24Node<E>(median); // Nowy korzeń
24       root.child.add(u); // u jest lewym dzieckiem elementu środkowego
25       root.child.add(v); // v jest prawym dzieckiem elementu środkowego
26     }
27     else {
28       Pobieranie rodzica węzła u, parentOfu;
29       insert(median, v, parentOfu); // Wstawianie elementu środkowego do rodzica
30    }
31   }
32 }
```

Wywołanie `insert(E e, Tree24Node<E> rightChildOfe, Tree24Node<E> u)` wstawia element `e` wraz z prawym dzieckiem do `u`. Gdy algorytm wstawia `e` do liścia, prawe dziecko `e` to `null` (wiersz 6.). W 2- i 3-węzłach wystarczy wstawić element do węzła (wiersze 15. – 17.). W 4-węźle należy wywołać metodę `split` i podzielić węzeł (wiersz 20.). Metoda `split` zwraca element środkowy. Należy rekurencyjnie wywołać metodę `insert`, by wstawić element środkowy do rodzica (wiersz 29.). Rysunek 35.9 ilustruje wstawianie elementów 34, 3, 50, 20, 15, 16, 25, 27, 29 i 24 do drzewa 2-3-4.

35.5. Usuwanie elementów z drzewa 2-3-4

Usuwanie elementu wymaga znalezienia węzła, który zawiera ten element, i usunięcia elementu z węzła.

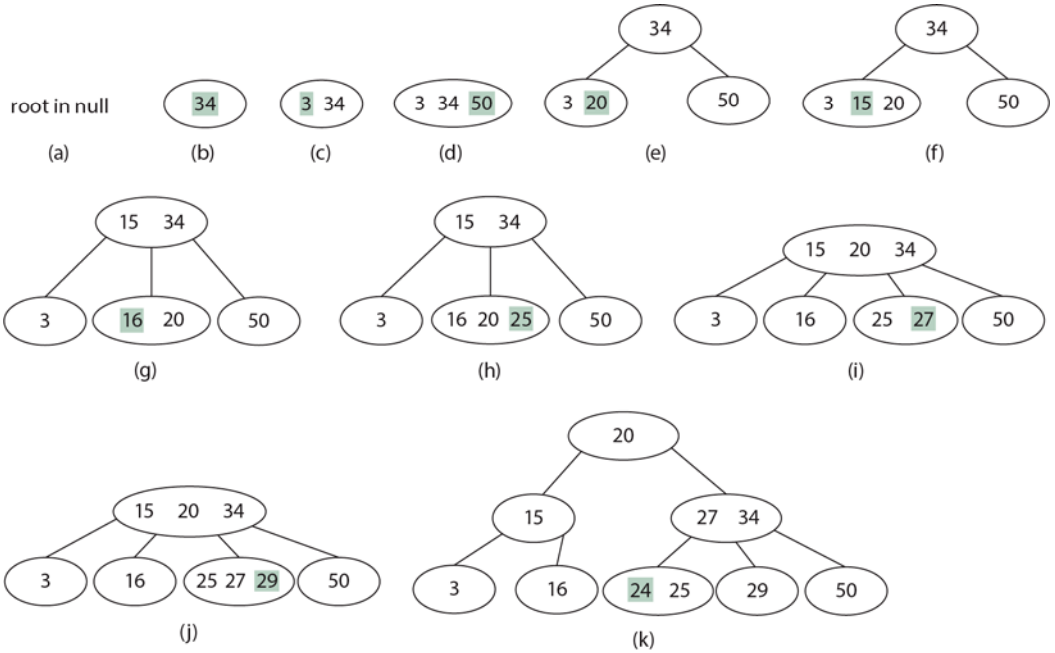
Aby usunąć element z drzewa 2-3-4, najpierw znajdź ten element w drzewie, by zlokalizować zawierający go węzeł. Jeśli elementu nie ma w drzewie, metoda zwraca `false`. Niech *u* będzie węzłem zawierającym dany element, a *rodzic_u* — rodzicem *u*. Rozważ trzy scenariusze:

Scenariusz 1. *u* jest 3- lub 4-węzłem będącym liściem. Usuń *e* z *u*.

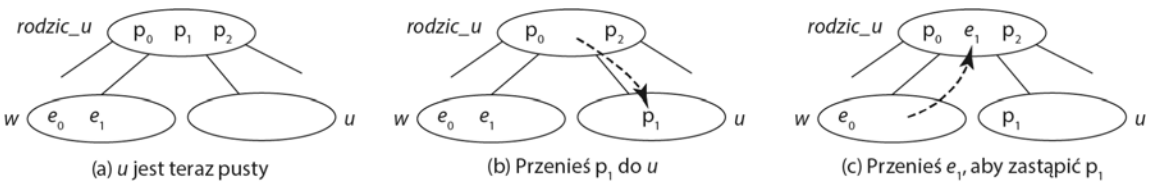
Scenariusz 2. *u* jest 2-węzłem będącym liściem. Usuń *e* z *u*. Teraz *u* jest pusty. Ta sytuacja to *niedopełnienie*. Aby je wyeliminować, rozważ dwie możliwości:

Scenariusz 2.1. Bezpośrednim lewym lub prawym bratem węzła *u* jest 3- lub 4-węzeł. Nazwij go *w* (rysunek 35.10a); przyjmij, że *w* jest lewym bratem węzła *u*. Wykonaj operację *przeniesienia*, która przeniesie element z *rodzic_u* do *u* (rysunek 35.10b) i przeniesie element z *w*, aby zastąpić element zabrany z *rodzic_u* (rysunek 35.10c).

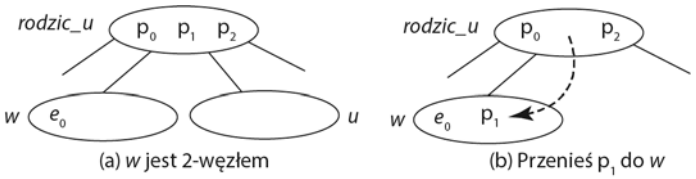
Scenariusz 2.2. Bezpośredni lewy i prawy brat węzła *u* są 2-węzłami (jeśli oba istnieją; *u* może też mieć tylko jednego brata). Niech tym bratem będzie *w* (rysunek 35.11a); przyjmij, że *w* jest lewym bratem węzła *u*. Wykonaj operację *złączenia*, która usuwa *u* i przenosi elementy z *rodzic_u* do *w* (rysunek 35.11b). Jeśli *rodzic_u* stanie się pusty, powtórz rekurencyjnie scenariusz 2., aby wykonać przeniesienie lub złączenie węzła *rodzic_u*.



Rysunek 35.9. Zmiany w drzewie w trakcie dodawania elementów 34, 3, 50, 20, 15, 16, 25, 27, 29 i 24 do pustego drzewa



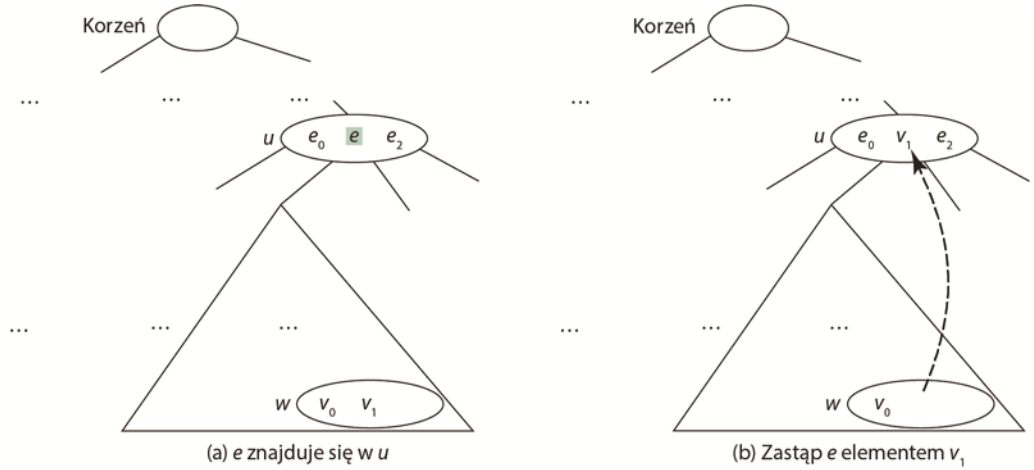
Rysunek 35.10. Operacja przeniesienia zapełnia pusty węzeł u



Rysunek 35.11. Operacja złączenia usuwa pusty węzeł u

Scenariusz 3. *u* nie jest liściem. Znajdź pierwszy od prawej liść w lewym poddrzewie *e*. Niech tym liściem będzie węzeł *w* (rysunek 35.12a). Przenieś ostatni element z *w*, aby zastąpić *e* w *u* (rysunek 35.12b). Jeśli *w* stanie się pusty, wykonaj operację przeniesienia lub złączenia dla *w*.

Algorytm usuwania elementu jest opisany na listingu 35.3.



RYSUNEK 35.12. Element w węźle wewnętrznym jest zastępowany elementem z liścia

LISTING 35.3. Usuwanie elementu w drzewie 2-3-4

```
1 /** Usuwanie podanego elementu w drzewie */
2 public boolean delete(E e) {
3   Znajdź węzeł zawierający element e
4   if (węzeł został znaleziony) {
5     delete(e, node); // Usuwanie elementu e z węzła
6     size--; // Po usunięciu jednego elementu
7     return true; // Element został usunięty
8   }
9
10  return false; // Elementu nie ma w drzewie
11 }
12
13 /** Usuwanie podanego elementu z węzła */
14 private void delete(E e, Tree24Node<E> node) {
15   if (e znajduje się w liściu) {
16     // Pobieranie ścieżki z korzenia do e
17     ArrayList<Tree24Node<E>> path = path(e);
18
19     Usuwanie e z węzła;
20
21     // Wykrywanie niedopełnienia w ścieżce i eliminowanie go
22     validate(e, node, path); // Sprawdzanie niedopełnienia w węźle
23   }
24   else { // e znajduje się w węźle wewnętrznym
25     Znajdź pierwszy od prawej węzeł w lewym poddrzewie węzła u;
26     Pobierz największy element ze znalezionego węzła;
27
28     // Pobieranie ścieżki z korzenia do e
29     ArrayList<Tree24Node<E>> path = path(rightmostElement);
30
31     Zastąp element w węźle pobranym największym elementem
32
```

```
33      // Wykrywanie niedopełnienia w węzłach ze ścieżki i eliminowanie go
34      validate(rightmostElement, rightmostNode, path);
35    }
36  }
37
38  /** Wykonywanie przeniesienia lub złączenia, jeśli jest to konieczne */
39  private void validate(E e, Tree24Node<E> u,
40      ArrayList<Tree24Node<E>> path) {
41    for (int i = path.size() - 1; i >= 0; i--) {
42      if (u nie jest pusty)
43        return; // Gotowe — nie trzeba wykonywać przeniesienia ani złączania
44
45      Tree24Node<E> parentOfu = path.get(i - 1); // Pobieranie rodzica węzła u
46
47      // Sprawdzanie obu braci
48      if (lewy brat węzła u ma więcej niż jeden element) {
49        Przenieś do u element z lewego brata
50      }
51      else if (prawy brat węzła u ma więcej niż jeden element) {
52        Przenieś do u element z prawego brata
53      }
54      else if (u ma lewego brata) { // Złączanie z lewym bratem
55        Złącz u z lewym bratem
56        u = parentOfu; // Powrót do pętli, aby sprawdzić rodzica
57      }
58      else { // Złączanie z prawym bratem (jeśli ten istnieje)
59        Złączanie u z prawym bratem
60        u = parentOfu; // Powrót do pętli, aby sprawdzić rodzica
61      }
62    }
63  }
```

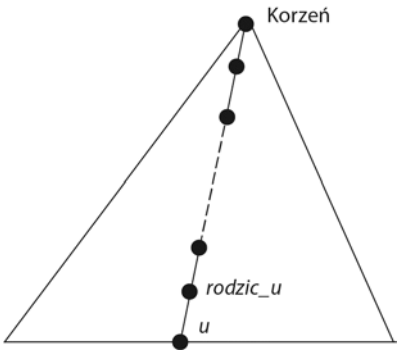
Metoda `delete(E e)` znajduje węzeł zawierający element `e` i wywołuje metodę `delete(E e, Tree24Node<E> node)` (wiersz 5.), aby usunąć element z węzła.

Jeśli węzeł jest liściem, należy pobrać ścieżkę z korzenia do `e` (wiersz 17.), usunąć `e` z węzła (wiersz 19.) i wywołać metodę `validate`, aby wykryć i naprawić puste węzły (wiersz 22.). Gdy występuje pusty węzeł, wywołanie `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` wykonuje przeniesienie lub złączenie. Ponieważ te operacje mogą spowodować, że rodzic węzła `u` stanie się pusty, potrzebna jest ścieżka z rodzicami wzdłuż ścieżki z korzenia do `u` (rysunek 35.13).

Jeśli węzeł nie jest liściem, należy znaleźć największy element w lewym poddrzewie węzła (wiersze 25. i 26.), pobrać ścieżkę z korzenia do tego elementu (wiersz 29.), zastąpić `e` w węźle znalezionym elementem (wiersz 31.) i wywołać metodę `validate` w celu naprawienia pierwszego od prawej węzła z poddrzewa, jeżeli stał się pusty (wiersz 34.).

Wywołanie `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` sprawdza, czy `u` jest pusty, i wykonuje przeniesienie lub złączenie, aby naprawić pusty węzeł. Jeśli węzeł nie jest pusty, metoda `validate` kończy pracę (wiersz 43.). Gdy węzeł jest pusty, trzeba rozważyć następujące scenariusze:

1. Jeśli `u` ma lewego brata z więcej niż jednym elementem, należy przenieść element z lewego brata do `u` (wiersz 49.).
2. W przeciwnym razie jeżeli `u` ma prawego brata z więcej niż jednym elementem, należy przenieść element z prawego brata do `u` (wiersz 52.).



RYSUNEK 35.13. Węzły wzdłuż ścieżki mogą w wyniku przeniesienia lub złączenia stać się puste

3. W przeciwnym razie jeśli u ma lewego brata, należy złączyć u z lewym bratem (wiersz 55.) i przypisać `parent➥Ofu` do u (wiersz 56.).
4. W przeciwnym razie u musi mieć prawego brata. Złącz u z prawym bratem (wiersz 59.) i przypisz `parentOfu` do u (wiersz 60.).

Wykonywany jest kod dla tylko jednego z tych scenariuszy. Następnie w nowej iteracji w razie potrzeby program wykonuje przeniesienie lub złączanie nowego węzła u. Na rysunku 35.14 pokazane są kroki potrzebne do usunięcia elementów 20, 15, 3, 6 i 34 z drzewa 2-3-4 z rysunku 35.9.k.

35.5.1. Jak znaleźć element w drzewie 2-3-4?

35.5.2. Jak wstawić element w drzewie 2-3-4?

35.5.3. Jak usunąć element w drzewie 2-3-4?

35.5.4. Przedstaw zmiany w drzewie 2-3-4 w trakcie wstawiania elementów: 1, 2, 3, 4, 10, 9, 7, 5, 8 i 6 (w tej kolejności).

35.5.5. Pokaż zmiany w drzewie zbudowanym w poprzednim zadaniu, gdy będziesz usuwać elementy: 1, 2, 3, 4, 10, 9, 7, 5, 8 i 6 (w tej kolejności).

35.5.6. Pokaż zmiany w B-drzewie stopnia 6. w trakcie wstawiania elementów: 1, 2, 3, 4, 10, 9, 7, 5, 8, 6, 17, 25, 18, 26, 14, 52, 63, 74, 80, 19 i 27 (w tej kolejności).

35.5.7. Pokaż zmiany w drzewie zbudowanym w poprzednim zadaniu, gdy będziesz usuwać elementy: 1, 2, 3, 4, 10, 9, 7, 5, 8 i 6 (w tej kolejności).

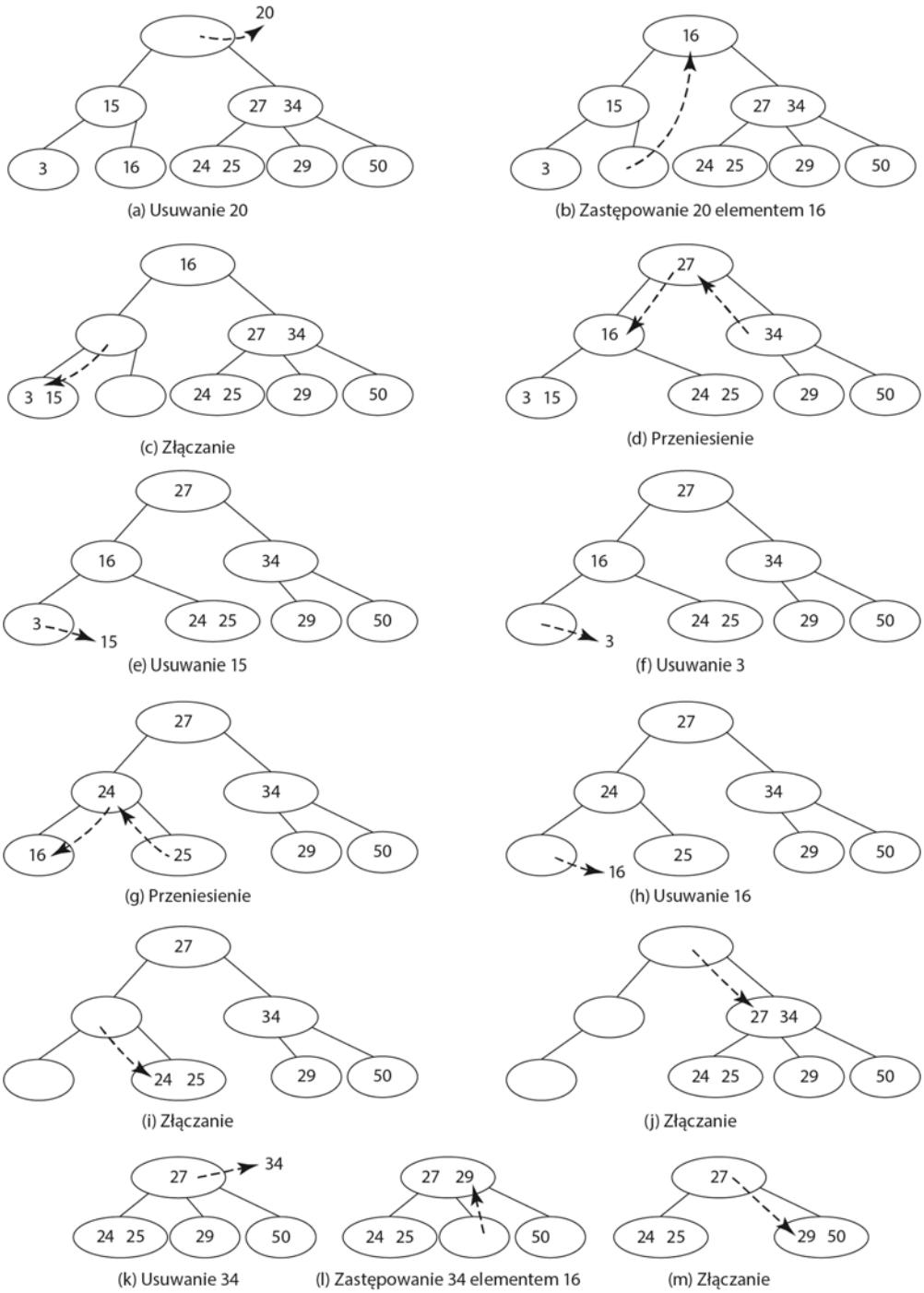
35.6. Odwiedzanie elementów w drzewie 2-3-4

Elementy drzewa 2-3-4 można odwiedzać w porządku inorder, preorder i postorder.

W drzewach 2-3-4 przydatne jest odwiedzanie elementów w porządku inorder, preorder i postorder. W metodzie inorder elementy są odwiedzane w porządku rosnącym. W metodzie preorder odwiedzane są elementy korzenia, a następnie rekurencyjnie poddrzewa od lewego do prawego. W metodzie postorder najpierw rekurencyjnie odwiedzane są poddrzewa od lewego do prawego, a następnie elementy w korzeniu.

Dla drzewa 2-3-4 z rysunku 35.9.k porządek inorder wygląda tak:

3 15 16 20 24 25 27 29 34 50



Rysunek 35.14. Zmiany w trakcie usuwania elementów 20, 15, 3, 6 i 34 z drzewa 2-3-4

porządek preorder wygląda tak:

20 15 3 16 27 34 24 25 29 50

a porządek postorder jest taki:

3 16 1 24 25 29 50 27 34 20

35.7. Implementowanie klasy Tree24

W tym podrozdziale pokazana jest kompletna implementacja klasy `Tree24`.

Na listingu 35.4 znajduje się kompletny kod źródłowy klasy `Tree24`.

LISTING 35.4. Tree24.java

```
1 import java.util.ArrayList;
2
3 public class Tree24<E extends Comparable<E>> implements Tree<E> {
4   private Tree24Node<E> root;
5   private int size;
6
7   /** Tworzy domyślne drzewo 2-3-4 */
8   public Tree24() {
9   }
10
11  /** Tworzy drzewo 2-3-4 na podstawie tablicy obiektów */
12  public Tree24(E[] elements) {
13    for (int i = 0; i < elements.length; i++)
14      insert(elements[i]);
15  }
16
17  @Override /* Wyszukuje element w drzewie */
18  public boolean search(E e) {
19    Tree24Node<E> current = root; // Rozpoczynanie od korzenia
20
21    while (current != null) {
22      if (matched(e, current)) { // Element znajduje się w danym węźle
23        return true; // Element został znaleziony
24      }
25      else {
26        current = getChildNode(e, current); // Wyszukiwanie w poddrzewie
27      }
28    }
29
30    return false; // Element nie znajduje się w drzewie
31  }
32
33  /** Zwraca true, jeśli element znajduje się w danym węźle */
34  private boolean matched(E e, Tree24Node<E> node) {
35    for (int i = 0; i < node.elements.size(); i++)
36      if (node.elements.get(i).equals(e))
37        return true; // Element został znaleziony
38
39    return false; // Element nie znajduje się w danym węźle
```

```
40  }
41
42  /** Znajduje dziecko, w którym szukany będzie element e */
43  private Tree24Node<E> getChildNode(E e, Tree24Node<E> node) {
44    if (node.child.size() == 0)
45      return null; // Węzeł jest liściem
46
47    int i = locate(e, node); // Znajdowanie punktu wstawiania dla e
48    return node.child.get(i); // Zwraca dziecko
49  }
50
51  @Override /** Wstawia element e do drzewa.
52              * Zwraca true po udanym wstawieniu elementu
53              */
54  public boolean insert(E e) {
55    if (root == null)
56      root = new Tree24Node<E>(e); // Tworzenie nowego korzenia, by zapisać w nim element
57    else {
58      // Znajdowanie liścia, gdzie można wstawić e
59      Tree24Node<E> leafNode = null;
60      Tree24Node<E> current = root;
61      while (current != null)
62        if (matched(e, current)) {
63          return false; // Znaleziono taki sam element; nowy element nie jest wstawiany
64        }
65        else {
66          leafNode = current;
67          current = getChildNode(e, current);
68        }
69
70      // Wstawianie elementu e w liściu
71      insert(e, null, leafNode); // Prawe dziecko elementu e to null
72    }
73
74    size++; // Zwiększanie wielkości
75    return true; // Element został wstawiony
76  }
77
78  /** Wstawianie elementu e do węzła u */
79  private void insert(E e, Tree24Node<E> rightChildOfe,
80      Tree24Node<E> u) {
81    // Pobieranie ścieżki wyszukiwania prowadzącej do elementu e
82    ArrayList<Tree24Node<E>> path = path(e);
83
84    for (int i = path.size() - 1; i >= 0; i--) {
85      if (u.elements.size() < 3) { // u jest 2- lub 3-węzłem
86        insert23(e, rightChildOfe, u); // Wstawianie e w węźle u
87        break; // Dalsze wstawianie w rodzicu u nie jest konieczne
88      }
89      else {
90        Tree24Node<E> v = new Tree24Node<E>(); // Tworzenie nowego węzła
91        E median = split(e, rightChildOfe, u, v); // Podział u
92
93        if (u == root) {
```

```
94          root = new Tree24Node<E>(median); // Nowy korzeń
95          root.child.add(u); // u jest lewym dzieckiem elementu środkowego
96          root.child.add(v); // v jest prawym dzieckiem elementu środkowego
97          break; // Dalsze wstawianie w rodzicu u nie jest konieczne
98        }
99        else {
100         // W następnej iteracji pętli używane będą nowe wartości
101         e = median; // Element wstawiany w rodzicu
102         rightChildOfe = v; // Prawe dziecko tego elementu
103         u = path.get(i - 1); // Nowy węzeł do wstawiania elementu
104       }
105     }
106   }
107 }
108
109 /** Wstawia element do 2- lub 3-węzła i zwraca punkt wstawiania */
110 private void insert23(E e, Tree24Node<E> rightChildOfe,
111     Tree24Node<E> node) {
112   int i = this.locate(e, node); // Znajdowanie miejsca wstawiania
113   node.elements.add(i, e); // Wstawianie elementu w węźle
114   if (rightChildOfe != null)
115     node.child.add(i + 1, rightChildOfe); // Wstawianie wskaźnika do dziecka
116 }
117
118 /** Dzieli 4-węzeł u na u i v oraz wstawia e do u lub v */
119 private E split(E e, Tree24Node<E> rightChildOfe,
120     Tree24Node<E> u, Tree24Node<E> v) {
121   // Przenoszenie ostatniego elementu z węzła u do v
122   v.elements.add(u.elements.remove(2));
123   E median = u.elements.remove(1);
124
125   // Podział dzieci w węźle, który nie jest liściem.
126   // Przeniesienie dwojga ostatnich dzieci z u do v
127   if (u.child.size() = 0) {
128     v.child.add(u.child.remove(2));
129     v.child.add(u.child.remove(2));
130   }
131
132   // Wstawianie e do 2- lub 3-węzła u lub v
133   if (e.compareTo(median) < 0)
134     insert23(e, rightChildOfe, u);
135   else
136     insert23(e, rightChildOfe, v);
137
138   return median; // Zwracanie elementu środkowego
139 }
140
141 /** Zwracanie ścieżki wyszukiwania prowadzącej do elementu e */
142 private ArrayList<Tree24Node<E> = path(E e) {
143   ArrayList<Tree24Node<E>= list = new ArrayList<Tree24Node<E>=();
144   Tree24Node<E> current = root; // Rozpoczynanie od korzenia
145
146   while (current != null) {
147     list.add(current); // Dodawanie węzła do listy
```

```
148      if (matched(e, current)) {
149        break; // Element został znaleziony
150      }
151      else {
152        current = getChildNode(e, current);
153      }
154    }
155
156    return list; // Zwracanie tablicy węzłów
157  }
158
159  @Override /** Usuwa określony element z drzewa */
160  public boolean delete(E e) {
161    // Znajdowanie węzła zawierającego element e
162    Tree24Node<E> node = root;
163      while (node != null)
164        if (matched(e, node)) {
165          delete(e, node); // Usuwanie elementu e z węzła
166          size--; // Po usunięciu jednego elementu
167          return true; // Element został usunięty
168        }
169        else {
170          node = getChildNode(e, node);
171        }
172
173    return false; // Element nie występuje w drzewie
174  }
175
176  /** Usuwa podany element z węzła */
177  private void delete(E e, Tree24Node<E> node) {
178    if (node.child.size() == 0) { // e znajduje się w liściu
179      // Pobieranie ścieżki z korzenia do e
180      ArrayList<Tree24Node<E>> path = path(e);
181
182      node.elements.remove(e); // Usuwanie elementu e
183
184      if (node == root) { // Przypadek specjalny
185        if (node.elements.size() == 0)
186          root = null; // Drzewo jest puste
187        return; // Gotowe
188      }
189
190      validate(e, node, path); // Wykrywanie niedopełnienia węzła
191    }
192    else {// e znajduje się w węźle wewnętrznym
193      // Znajdowanie pierwszego od prawej węzła w lewym poddrzewie danego węzła
194      int index = locate(e, node); // Indeks elementu e w węźle
195      Tree24Node<E> current = node.child.get(index);
196      while (current.child.size() > 0) {
197        current = current.child.get(current.child.size() - 1);
198      }
199      E rightmostElement =
200        current.elements.get(current.elements.size() - 1);
201
202      // Pobieranie ścieżki z korzenia do e
```

```
203      ArrayList<Tree24Node<E>> path = path(rightmostElement);
204
205      // Zastępowanie usuniętego elementu elementem pierwszym od prawej
206      node.elements.set(index, current.elements.remove(
207        current.elements.size() - 1));
208
209      validate(rightmostElement, current, path); // Wykrywanie niedopełnienia
210    }
211  }
212
213  /** Wykonuje przeniesienie i złączanie (jeśli to konieczne) */
214  private void validate(E e, Tree24Node<E> u,
215      ArrayList<Tree24Node<E>> path) {
216    for (int i = path.size() - 1; u.elements.size() == 0; i--) {
217      Tree24Node<E> parentOfu = path.get(i - 1); // Pobieranie rodzica elementu u
218      int k = locate(e, parentOfu); // Indeks elementu e w rodzicu
219
220      // Sprawdzanie braci
221      if (k > 0&& parentOfu.child.get(k - 1).elements.size() > 1) {
222        leftSiblingTransfer(k, u, parentOfu);
223      }
224      else if (k + 1 < parentOfu.child.size() &&
225               parentOfu.child.get(k + 1).elements.size() > 1) {
226        rightSiblingTransfer(k, u, parentOfu);
227      }
228      else if (k - 1 == 0) { // Złączanie z lewym bratem
229        // Pobieranie lewego brata węzła u
230        Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
231
232        // Złączanie z lewym bratem węzła u
233        leftSiblingFusion(k, leftNode, u, parentOfu);
234
235        // Koniec, gdy korzeń staje się pusty
236        if (parentOfu == root && parentOfu.elements.size() == 0) {
237          root = leftNode;
238          break;
239        }
240
241        u = parentOfu; // Powrót do pętli w celu sprawdzenia rodzica
242      }
243      else { // Złączanie z prawym bratem (który musi istnieć)
244        // Pobieranie lewego brata węzła u
245        Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
246
247        // Złączanie z prawym bratem węzła u
248        rightSiblingFusion(k, rightNode, u, parentOfu);
249
250        // Koniec, gdy korzeń staje się pusty
251        if (parentOfu == root && parentOfu.elements.size() == 0) {
252          root = rightNode;
253          break;
254        }
255
256        u = parentOfu; // Powrót do pętli w celu sprawdzenia rodzica
257      }
```

```
258     }
259   }
260
261   /** Znajdowanie punktu wstawiania elementu w węźle */
262   private int locate(E o, Tree24Node<E> node) {
263     for (int i = 0; i < node.elements.size(); i++) {
264       if (o.compareTo(node.elements.get(i)) <= 0) {
265         return i;
266       }
267     }
268
269     return node.elements.size();
270   }
271
272   /** Przeniesienie z lewego brata */
273   private void leftSiblingTransfer(int k,
274       Tree24Node<E> u, Tree24Node<E> parentOfu) {
275     // Przenoszenie elementu z rodzica do u
276     u.elements.add(0, parentOfu.elements.get(k - 1));
277
278     // Przenoszenie elementu z lewego węzła do rodzica
279     Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
280     parentOfu.elements.set(k - 1,
281       leftNode.elements.remove(leftNode.elements.size() - 1));
282
283     // Przenoszenie wskaźnika do dziecka z lewego brata do danego węzła
284     if (leftNode.child.size() > 0)
285       u.child.add(0, leftNode.child.remove(
286         leftNode.child.size() - 1));
287   }
288
289   /** Przenoszenie z prawego dziecka */
290   private void rightSiblingTransfer(int k,
291       Tree24Node<E> u, Tree24Node<E> parentOfu) {
292     // Przenoszenie elementu z rodzica do u
293     u.elements.add(parentOfu.elements.get(k));
294
295     // Przenoszenie elementu z prawego węzła do rodzica
296     Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
297     parentOfu.elements.set(k, rightNode.elements.remove(0));
298
299     // Przenoszenie wskaźnika do dziecka z prawego brata do danego węzła
300     if (rightNode.child.size() > 0)
301       u.child.add(rightNode.child.remove(0));
302   }
303
304   /** Złączanie z lewym bratem */
305   private void leftSiblingFusion(int k, Tree24Node<E> leftNode,
306       Tree24Node<E> u, Tree24Node<E> parentOfu) {
307     // Przenoszenie elementu z rodzica do lewego brata
308     leftNode.elements.add(parentOfu.elements.remove(k - 1));
309
310     // Usuwanie wskaźnika do pustego węzła
311     parentOfu.child.remove(k);
312
```

```
313     // Modyfikowanie wskaźników do dzieci, jeśli węzeł nie jest liściem
314     if (u.child.size() > 0)
315       leftNode.child.add(u.child.remove(0));
316   }
317
318   /** Złączanie z prawym bratem */
319   private void rightSiblingFusion(int k, Tree24Node<E> rightNode,
320       Tree24Node<E> u, Tree24Node<E> parentOfu) {
321     // Przenoszenie elementu z rodzica do prawego brata
322     rightNode.elements.add(0, parentOfu.elements.remove(k));
323
324     // Usuwanie wskaźnika do pustego węzła
325     parentOfu.child.remove(k);
326
327     // Modyfikowanie wskaźników do dzieci, jeśli węzeł nie jest liściem
328     if (u.child.size() > 0)
329       rightNode.child.add(0, u.child.remove(0));
330   }
331
332   /** Pobiera liczbę węzłów w drzewie */
333   public int getSize() {
334     return size;
335   }
336
337   /** Przechodzi drzewo w porządku preorder, zaczynając od korzenia */
338   public void preorder() {
339     preorder(root);
340   }
341
342   /** Przechodzi drzewo w porządku preorder, zaczynając od poddrzewa */
343   private void preorder(Tree24Node<E> root) {
344     if (root == null) return;
345     for (int i = 0; i < root.elements.size(); i++)
346       System.out.print(root.elements.get(i) + " ");
347
348     for (int i = 0; i < root.child.size(); i++)
349       preorder(root.child.get(i));
350   }
351
352   /** Przechodzi drzewo w porządku inorder, zaczynając od korzenia */
353   public void inorder() {
354     // Ćwiczenie dla czytelników
355   }
356
357   /** Przechodzi drzewo w porządku postorder, zaczynając od korzenia */
358   public void postorder() {
359     // Ćwiczenie dla czytelników
360   }
361
362   /** Zwraca true, jeśli drzewo jest puste */
363   public boolean isEmpty() {
364     return root == null;
365   }
366
367   @Override /** Usuwa wszystkie elementy z drzewa */
```

```
368  public void clear() {
369    root = null;
370    size = 0;
371  }
372
373  /** Zwraca iterator do przetwarzania elementów drzewa */
374  public java.util.Iterator iterator() {
375    // Ćwiczenie dla czytelników
376    return null;
377  }
378
379  /** Definiuje węzeł drzewa 2-3-4 */
380  protected static class Tree24Node<E extends Comparable<E>> {
381    // Węzły mają maksymalnie trzy elementy
382    ArrayList<E> elements = new ArrayList<E>(3);
383    // Każdy może mieć do czworga dzieci
384    ArrayList<Tree24Node<E>> child
385      = new ArrayList<Tree24Node<E>>(4);
386
387    /** Tworzy pusty węzeł typu Tree24Node */
388    Tree24Node() {
389    }
390
391    /** Tworzy węzeł typu Tree24Node z początkowym elementem */
392    Tree24Node(E o) {
393      elements.add(o);
394    }
395  }
396 }
```

Klasa `Tree24` obejmuje pola `root` i `size` (wiersze 4. i 5.). Pole `root` wskazuje korzeń, a pole `size` przechowuje liczbę elementów w drzewie.

Klasa `Tree24` ma dwa konstruktory: bezargumentowy (wiersze 8. i 9.), który tworzy puste drzewo, i konstruktor tworzący drzewo na podstawie tablicy elementów (wiersze 12. – 15.).

Metoda `search` (wiersze 18. – 31.) szuka elementów w drzewie. Zwraca wartość `true` (wiersz 23.), jeśli element znajduje się w drzewie, lub wartość `false`, gdy wyszukiwanie dociera do pustego poddrzewa (wiersz 30.).

Metoda `matched(e, node)` (wiersze 34. – 40.) sprawdza lokalizację elementu `e` w węźle.

Metoda `getChildNode(e, node)` (wiersze 43. – 49.) zwraca korzeń poddrzewa, w którym należy szukać elementu `e`.

Metoda `insert(E e)` (wiersze 54. – 76.) wstawia element w drzewie. Jeśli drzewo jest puste, tworzony jest nowy korzeń (wiersz 56.). Ta metoda znajduje liść, w którym element ma być umieszczony, i wywołuje metodę `insert(e, null, leafNode)` w celu wstawienia tego elementu (wiersz 71.).

Metoda `insert(e, rightChildOfe, u)` wstawia element do węzła `u` (wiersze 79. – 107.). Najpierw wywołuje ona metodę `path(e)` (wiersz 82.), aby pobrać ścieżkę z korzenia do węzła `u`. W każdej iteracji pętli `for` analizowany jest element `u` i jego rodzic — `parentOfu` (wiersze 84. – 106.). Jeśli `u` jest 2- lub 3-węzłem, należy wywołać metodę `insert23(e, rightChildOfe, u)`, by dodać do `u` element `e` i wskaźnik do dziecka `rightChildOfe` (wiersz 86.). Podział nie jest wtedy konieczny (wiersz 87.). Gdy `u` nie jest 2- lub 3-węzłem, należy utworzyć nowy węzeł `v` (wiersz 90.) i wywołać metodę `split(e, rightChildOfe, u, v)` (wiersz 91.) w celu podziału `u` na `u` i `v`. Metoda `split` wstawia `e` do `u` lub do `v`, a następnie zwraca element środkowy pierwotnego węzła `u`. Jeżeli `u` jest korzeniem, należy utworzyć nowy korzeń zawierający element środkowy oraz ustawić `u` i `v` jako lewe i prawe dziecko elementu środkowego

(wiersze 95. i 96.). Gdy `u` nie jest korzeniem, element środkowy jest w następnej iteracji wstawiany do `parentOfu` (wiersze 101. – 103.).

Metoda `insert23(e, rightChildOfe, node)` wstawia do węzła element `e` wraz ze wskaźnikiem do prawego dziecka (wiersze 110. – 116.). Ta metoda najpierw wywołuje metodę `locate(e, node)` (wiersz 112.), aby znaleźć punkt wstawiania, a następnie wstawia `e` do węzła (wiersz 113.). Jeśli wskaźnik `rightChildOfe` jest różny od `null`, zostaje wstawiony na listę dzieci danego węzła (wiersz 115.).

Metoda `split(e, rightChildOfe, u, v)` dzieli 4-węzeł `u` (wiersze 119. – 139.). Przebiega to tak: (1) przeniesienie ostatniego elementu z `u` do `v` i usunięcie środkowego elementu z `u` (wiersze 122. i 123.); (2) przeniesienie dwóch ostatnich wskaźników do dzieci z `u` do `v` (wiersze 127. – 130.), jeśli `u` nie jest liściem; (3) jeżeli `e < median`, wstawianie `e` do `u`; w przeciwnym razie wstawianie `e` do `v` (wiersze 133. – 136.); (4) zwrócenie elementu środkowego (wiersz 138.).

Metoda `path(e)` zwraca listę `ArrayList` z węzłami (począwszy od korzenia) sprawdzanymi w trakcie szukania `e` (wiersze 142. – 157.). Jeśli `e` znajduje się w drzewie, ostatni węzeł ścieżki zawiera element `e`. W przeciwnym razie ostatni węzeł jest miejscem, w którym należy wstawić `e`.

Metoda `delete(E e)` usuwa element z drzewa (wiersze 160. – 174.). Najpierw znajduje ona węzeł zawierający `e`, a następnie wywołuje metodę `delete(e, node)`, by usunąć `e` z tego węzła (wiersz 165.). Jeśli element nie znajduje się w drzewie, metoda zwraca `false` (wiersz 173.).

Metoda `delete(e, node)` usuwa element z węzła `u` (wiersze 177. – 211.). Jeśli ten węzeł jest liściem, należy pobrać ścieżkę prowadzącą do `e` (wiersz 180.), usunąć `e` (wiersz 182.), przypisać `null` do korzenia, jeżeli drzewo stało się puste (wiersze 184. – 188.), i wywołać metodę `validate`, aby wykonać przeniesienie lub złączenie dla pustych węzłów (wiersz 190.). Jeżeli węzeł nie jest liściem, należy znaleźć element pierwszy od prawej (wiersze 194. – 200.), pobrać ścieżkę prowadzącą do `e` (wiersz 203.), zastąpić `e` elementem pierwszym od prawej (wiersze 206. i 207.) i wywołać metodę `validate`, aby wykonać przeniesienie lub złączenie dla pustych węzłów (wiersz 209.).

Metoda `validate(e, u, path)` gwarantuje, że drzewo będzie poprawnym drzewem 2-3-4 (wiersze 214. – 259.). Gdy `u` nie jest pustym węzłem, pętla `for` kończy pracę (wiersz 216.). Ciało pętli jest wykonywane, aby naprawić pusty węzeł `u` za pomocą przeniesienia lub złączenia. Jeśli istnieje lewy brat z więcej niż jednym elementem, należy przenieść element między lewym bratem i `u` (wiersz 222.). W przeciwnym razie jeżeli istnieje prawy brat z więcej niż jednym elementem, element jest przenoszony między prawym bratem i `u` (wiersz 226.). W przeciwnym razie jeśli istnieje lewy brat, należy złączyć `u` z lewym bratem (wiersze 230. – 239.) i w następnej iteracji pętli sprawdzić poprawność `parentOfu` (wiersz 241.). W przeciwnym razie trzeba złączyć `u` z prawym bratem.

Metoda `locate(e, node)` znajduje indeks `e` w węźle (wiersze 262. – 270.).

Metoda `leftSiblingTransfer(k, u, parentOfu)` przenosi element między lewym bratem i `u` (wiersze 273. – 287.). Metoda `rightSiblingTransfer(k, u, parentOfu)` przenosi element między prawym bratem i `u` (wiersze 290. – 302.). Metoda `leftSiblingFusion(k, leftNode, u, parentOfu)` złącza `u` z lewym bratem `leftNode` (wiersze 305. – 316.). Metoda `rightSiblingFusion(k, rightNode, u, parentOfu)` złącza `u` z prawym bratem `rightNode` (wiersze 319. – 330.).

Metoda `preorder()` wyświetla wszystkie elementy drzewa w porządku preorder (wiersze 338. – 350.).

Klasa wewnętrzna `Tree24Node` reprezentuje węzły drzewa (wiersze 374. – 389.).

35.8. Testowanie klasy Tree24

W tym podrozdziale napiszesz program testowy dla klasy `Tree24`.

Na listingu 35.5 przedstawiony jest program testowy. Ten program tworzy drzewo 2-3-4 i wstawia do niego elementy (wiersze 6. – 20.), a dalej usuwa te elementy (wiersze 22. – 56.).

LISTING 35.5. TestTree24.java

```
 1 public class TestTree24 {
 2   public static void main(String[] args) {
 3     // Tworzenie drzewa 2-3-4
 4     Tree24<Integer> tree = new Tree24<Integer>();
 5
 6     tree.insert(34);
 7     tree.insert(3);
 8     tree.insert(50);
 9     tree.insert(20);
10     tree.insert(15);
11     tree.insert(16);
12     tree.insert(25);
13     tree.insert(27);
14     tree.insert(29);
15     tree.insert(24);
16     System.out.print("\nPo wstawieniu 24:");
17     printTree(tree);
18     tree.insert(23);
19     tree.insert(22);
20     tree.insert(60);
21     tree.insert(70);
22     System.out.print("\nPo wstawieniu 70:");
23     printTree(tree);
24
25     tree.delete(34);
26     System.out.print("\nPo usunięciu 34:");
27     printTree(tree);
28
29     tree.delete(25);
30     System.out.print("\nPo usunięciu 25:");
31     printTree(tree);
32
33     tree.delete(50);
34     System.out.print("\nPo usunięciu 50:");
35     printTree(tree);
36
37     tree.delete(16);
38     System.out.print("\nPo usunięciu 16:");
39     printTree(tree);
40
41     tree.delete(3);
42     System.out.print("\nPo usunięciu 3:");
43     printTree(tree);
44
45     tree.delete(15);
46     System.out.print("\nPo usunięciu 15:");
47     printTree(tree);
48   }
49
50   public static <E extends Comparable<E>>
51     void printTree(Tree<E> tree) {
52     // Przechodzenie drzewa
53     System.out.print("\nPreorder: ");
54     tree.preorder();
```

```
55     System.out.print("\nLiczba węzłów: " + tree.getSize());
56     System.out.println();
57   }
58 }
```

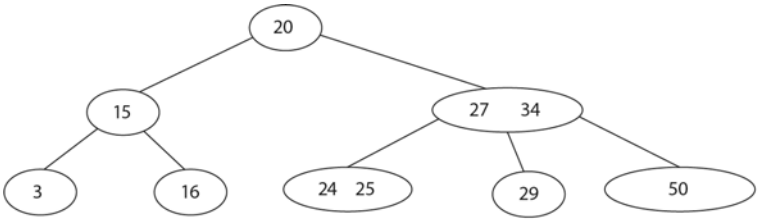
```
Po wstawieniu 24:
Preorder: 20 15 3 16 27 34 24 25 29 50
Liczba węzłów: 10

Po wstawieniu 70:
Preorder: 20 15 3 16 24 27 34 22 23 25 29 50 60 70
Liczba węzłów: 14

Po usunięciu 34:
Preorder: 20 15 3 16 24 27 50 22 23 25 29 60 70
Liczba węzłów: 13

Po usunięciu 25:
Preorder: 20 15 3 16 23 27 50 22 24 29 60 70
Liczba węzłów: 12

Po usunięciu 50:
Preorder: 20 15 3 16 23 27 60 22 24 29 70
Liczba węzłów: 11

Po usunięciu 16:
Preorder: 23 20 3 15 22 27 60 24 29 70
Liczba węzłów: 10

Po usunięciu 3:
Preorder: 23 20 15 22 27 60 24 29 70
Liczba węzłów: 9

Po usunięciu 15:
Preorder: 27 23 20 22 24 60 29 70
Liczba węzłów: 8
```

Rysunek 35.15 pokazuje zmiany w drzewie zachodzące wraz z dodawaniem elementów. Rysunek 35.15a przedstawia drzewo po dodaniu elementów 34, 3, 50, 20, 15, 16, 25, 27, 29 i 24, rysunek 35.15b — po dodaniu 23, 22, 60 i 70, 35.15c — po usunięciu 34, 35.15d — po usunięciu 25, 35.15e — po usunięciu 50, 35.15f — po usunięciu 16, 35.15g — po usunięciu 3, a 35.15h — po usunięciu 15.

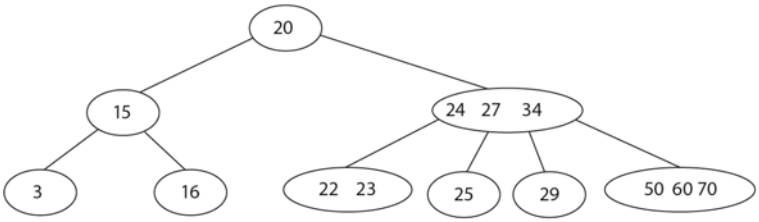
35.9. Analiza złożoności czasowej

W drzewach 2-3-4 wyszukiwanie, wstawianie i usuwanie mają złożoność O*(log*n*).*

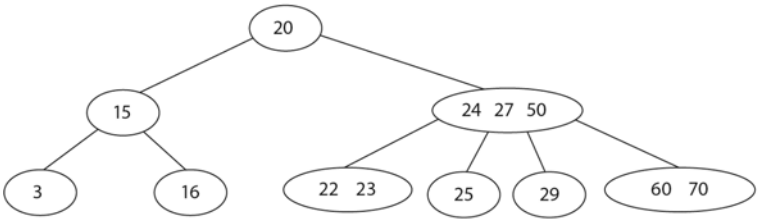
Ponieważ drzewo 2-3-4 jest zrównoważonym drzewem binarnym, jego wysokość to najwyżej *O*(log*n*). Metody `search`, `insert` i `delete` operują na węzłach wzdłuż ścieżki. Znalezienie elementu w węźle zajmuje stały czas. Dlatego metoda `search` ma złożoność *O*(log*n*). Jeśli chodzi o metodę `insert`, to czas podziału węzła jest stały, dlatego ta metoda też ma złożoność *O*(log*n*). W metodzie `delete` przenoszenie i złączanie zajmują stały czas, tak więc również jej złożoność wynosi *O*(log*n*).



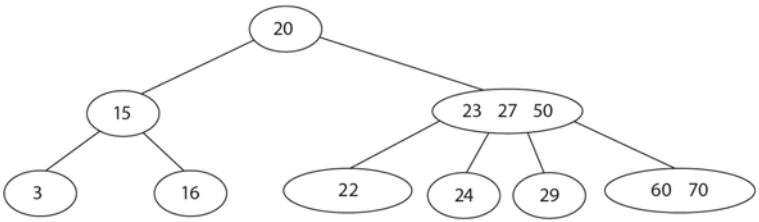
(a) Po wstawieniu 34, 3, 50, 20, 15, 16, 25, 27, 29 i 24 (w tej kolejności)



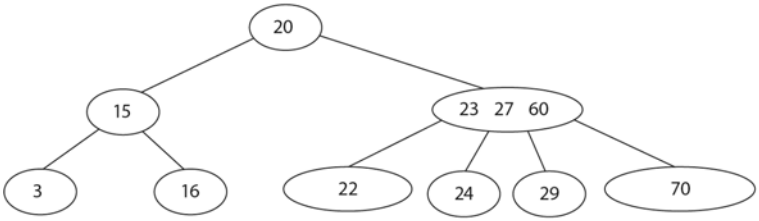
(b) Po wstawieniu 23, 22, 60 i 70



(c) Po usunięciu 34

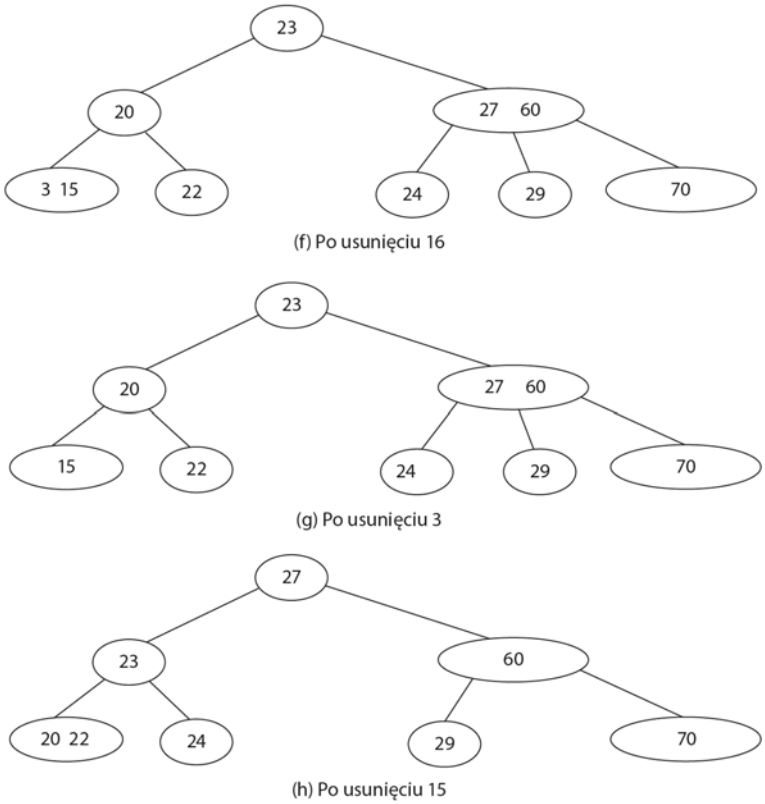


(d) Po usunięciu 25



(e) Po usunięciu 50

Ciąg dalszy rysunku na następnej stronie



RYSUNEK 35.15. Zmiany w drzewie w trakcie wstawiania i usuwania elementów

35.10. B-drzewo

B-drzewo to uogólnione drzewo 2-3-4.

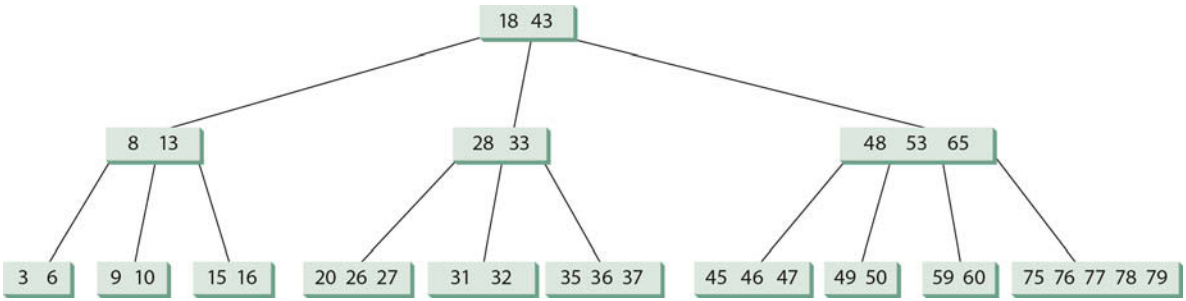
Do tej pory obowiązywało założenie, że cały zbiór danych jest przechowywany w pamięci. Co się dzieje, gdy zbiór danych jest za duży i nie mieści się w pamięci (dotyczy to większości baz danych, gdzie dane są przechowywane na dysku)? Przyjmij, że używasz drzewa AVL do przechowywania miliona rekordów z tabeli bazodanowej. Aby znaleźć rekord, średnio trzeba sprawdzić $\log_2 1\,000\,000 \approx 20$ węzłów. Nie jest to problem, jeśli wszystkie węzły znajdują się w pamięci. Jednak gdy węzły są zapisane na dysku, koniecznych jest 20 odczytów z niego. Dyskowe operacje I/O są kosztowne i tysiące razy wolniejsze od operacji w pamięci. Aby poprawić wydajność, trzeba ograniczyć liczbę dyskowych operacji I/O. Wydajną strukturą danych do wyszukiwania, wstawiania i usuwania danych przechowywanych w pamięci zewnętrznej (na przykład na dyskach twardych) jest B-drzewo będące uogólnioną postacią drzew 2-3-4.

B-drzewo stopnia d jest zdefiniowane tak:

1. Każdy węzeł oprócz korzenia ma od $\lceil d/2 \rceil - 1$ do $d - 1$ kluczy.
2. Korzeń może mieć do $d - 1$ kluczy.

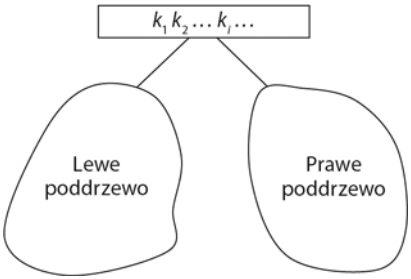
3. Węzeł niebędący liściem o k kluczach ma $k + 1$ dzieci.
4. Wszystkie liście mają tę samą głębokość.

Rysunek 35.16 przedstawia B-drzewo stopnia 6. Dla uproszczenia używane są tu klucze całkowitoliczbowe. Każdy klucz jest powiązany ze wskaźnikiem prowadzącym do rekordu w bazie danych. Na rysunku te wskaźniki zostały pominięte.



Rysunek 35.16. W B-drzewie stopnia 6. każdy węzeł oprócz korzenia może zawierać od dwóch do pięciu kluczy

Zauważ, że B-drzewo jest drzewem wyszukiwań. Klucze w każdym węźle są zapisane rosnąco. Każdy klucz w węźle wewnętrznym ma lewe i prawe poddrzewo (rysunek 35.17). Wszystkie klucze w lewym poddrzewie są mniejsze od klucza z rodzica, a wszystkie klucze w prawym poddrzewie są większe od klucza z rodzica.



Rysunek 35.17. Klucze w lewym (prawym) poddrzewie klucza k_i są mniejsze (większe) od k_i

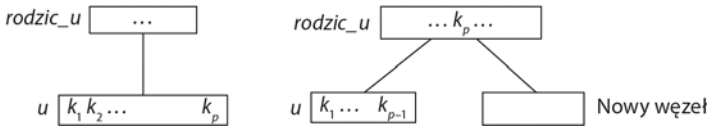
Podstawową jednostką dyskowych operacji I/O jest blok. W trakcie odczytu danych z dysku wczytywany jest cały blok zawierający szukane dane. Powinieneś dobrać odpowiedni stopień d, aby węzeł mieścił się w bloku dyskowym. Minimalizuje to liczbę dyskowych operacji I/O.

Drzewo 2-3-4 jest B-drzewem stopnia 4. Techniki wstawiania i usuwania danych w drzewach 2-3-4 można łatwo uogólnić na potrzeby B-drzew.

Wstawianie klucza do B-drzewa przebiega podobnie jak w drzewach 2-3-4. Najpierw znajdź liść, w którym należy wstawić klucz, i wstaw klucz. Jeśli po wstawieniu liść zawiera d kluczy, następuje przepełnienie. Aby rozwiązać ten problem, przeprowadź *podział* podobny jak w drzewie 2-3-4.

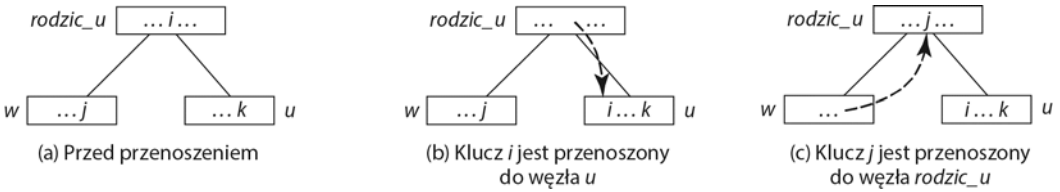
Niech u oznacza dzielony węzeł, a m — środkowy klucz węzła. Utwórz nowy węzeł i przenieś wszystkie klucze większe od m do nowego węzła. Wstaw m do rodzica węzła u. Teraz u staje się lewym dzieckiem m, a v prawym dzieckiem m (rysunek 35.18). Jeśli wstawienie m do rodzica węzła u spowoduje przepełnienie, należy w analogiczny sposób przeprowadzić podział rodzica.

Klucz *k* można usunąć z B-drzewa w taki sam sposób jak w drzewie 2-3-4. Najpierw znajdź węzeł *u* zawierający usuwany klucz. Rozważ dwa scenariusze:

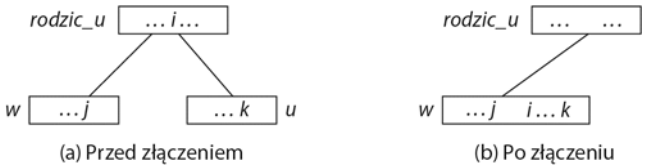


Rysunek 35.18. (a) Po wstawieniu nowego klucza do węzła u; (b) Środkowy klucz k_p jest wstawiany do rodzic_u

Scenariusz 1. Jeśli *u* jest liściem, usuń klucz z *u*. Jeżeli po usunięciu w *u* znajduje się mniej niż $\lceil d/2 \rceil - 1$ kluczy, następuje niedopełnienie. Aby je wyeliminować, przenieś element z mającego więcej niż $\lceil d/2 \rceil - 1$ kluczy brata *w* węzła *u*, jeśli taki brat istnieje (rysunek 35.19). W przeciwnym razie złącz *u* z bratem *w* (rysunek 35.20).



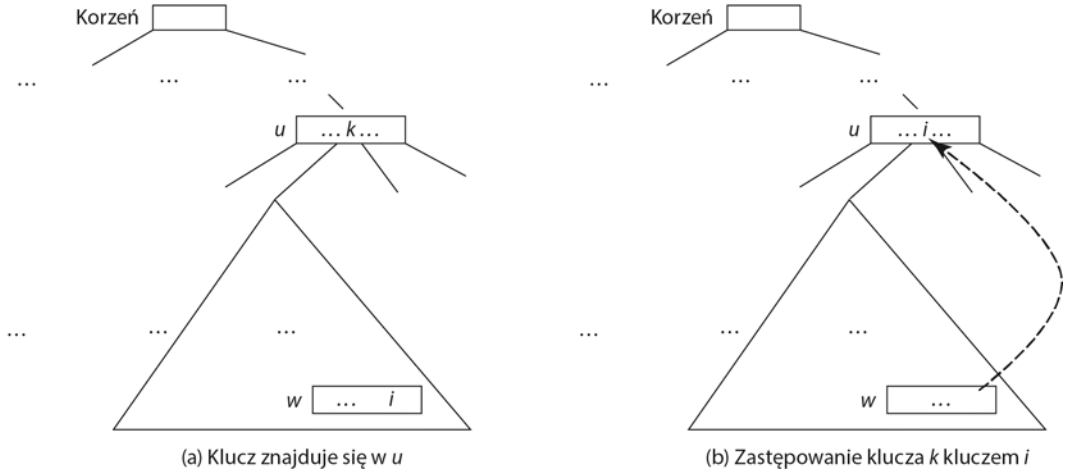
Rysunek 35.19. Klucz jest przenoszony z rodzic_u do u oraz z brata węzła u (w) do rodzic_u



Rysunek 35.20. W ramach złączania klucz i jest przenoszony z rodzic_u do w, a wszystkie klucze z u są przenoszone do w

Scenariusz 2. Węzeł *u* nie jest liściem. Znajdź pierwszy od prawej liść w lewym poddrzewie elementu *k*. Niech będzie to liść *w* (rysunek 35.21a). Przenieś ostatni klucz z *w*, aby zastąpić *k* w węźle *u* (rysunek 35.21b). Jeśli w węźle *w* wystąpi niedopełnienie, przeprowadź przenoszenie lub złączenie na węźle *w*.

Wydajność B-drzewa zależy od liczby dyskowych operacji I/O (czyli od liczby węzłów używanych w trakcie wykonywania zadania). Liczba węzłów używanych w trakcie wyszukiwania, wstawiania i usuwania elementów zależy od wysokości drzewa. W przypadku pesymistycznym każdy węzeł zawiera $\lceil d/2 \rceil - 1$ kluczy. Dlatego wysokość drzewa wynosi $\log_{\lceil d/2 \rceil} n$, gdzie *n* to liczba kluczy. W przypadku optymistycznym każdy węzeł zawiera *d* – 1 kluczy. Dlatego wysokość drzewa wynosi $\log_d n$. Rozważ B-drzewo stopnia 12. zawierające 10 milionów kluczy. Wysokość tego drzewa wynosi między $\log_6 10\,000\,000 \approx 7$ a $\log_{12} 10\,000\,000 \approx 9$. Dlatego w trakcie wyszukiwania, wstawiania i usuwania elementów maksymalna liczba odwiedzanych węzłów wynosi 42. Jeśli używasz drzewa AVL, maksymalna liczba odwiedzanych węzłów jest równa $\log_2 10\,000\,000 \approx 24$.



RYSUNEK 35.21. Klucz w węźle wewnętrznym jest zastępowany przez element z liścia

NAJWAŻNIEJSZE POJĘCIA

drzewo 2-3-4
drzewo 2-4
2-węzeł
3-węzeł
4-węzeł
B-drzewo
złączanie
podział
przeniesienie

PODSUMOWANIE ROZDZIAŁU

1. Drzewo 2-3-4 jest kompletnym zrównoważonym drzewem wyszukiwań. W drzewie 2-3-4 węzeł może zawierać jeden, dwa lub trzy elementy.
2. Wyszukiwanie elementu w drzewie 2-3-4 przypomina wyszukiwanie elementu w drzewie binarnym. Różnica dotyczy wyszukiwania elementu w węźle.
3. Aby wstawić element w drzewie 2-3-4, znajdź liść, w którym należy umieścić ten element. Jeśli liść jest 2- lub 3-węzłem, wystarczy wstawić element w tym węźle. Jeżeli węzeł jest 4-węzłem, należy go podzielić.
4. Proces usuwania elementu w drzewie 2-3-4 przebiega podobnie jak w drzewie binarnym. Różnica polega na tym, że trzeba wykonać przeniesienie lub złączanie dla pustych węzłów.
5. Wysokość drzewa 2-3-4 wynosi $O(\log n)$. Dlatego złożoność czasowa wyszukiwania, wstawiania i usuwania jest równa $O(\log n)$.
6. B-drzewo to uogólniona postać drzewa 2-3-4. Każdy węzeł (z wyjątkiem korzenia) w B-drzewie stopnia d może mieć od $\lceil d/2 \rceil - 1$ do $d - 1$ kluczy. Drzewa 2-3-4 są bardziej płaskie niż drzewa AVL, a B-drzewa są bardziej płaskie od drzew 2-3-4. B-drzewa są wydajnym narzędziem do tworzenia indeksów danych w systemach bazodanowych, w których duża ilość danych jest przechowywana na dyskach.

Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

ĆWICZENIA PROGRAMISTYCZNE

***35.1.** *Implementacja metody `inorder`*. Zaimplementuj potraktowaną jako ćwiczenie metodę `inorder` z klasy `Tree24`.

35.2. *Implementacja metody `postorder`*. Zaimplementuj potraktowaną jako ćwiczenie metodę `postorder` z klasy `Tree24`.

35.3. *Implementacja metody `iterator`*. Zaimplementuj potraktowaną jako ćwiczenie metodę `iterator` z klasy `Tree24`, używając porządku inorder.

***35.4.** *Graficzne wyświetlanie 2-3-4 drzewa*. Napisz program z GUI do wyświetlania 2-3-4 drzew.

*****35.5.** *Animacja ilustrująca drzewa 2-3-4*. Napisz program z GUI z animacjami działania metod `insert`, `delete` i `search` dla drzew 2-3-4 (rysunek 35.4).

****35.6.** *Wskaźnik do rodzica w klasie `Tree24Node`*. Zmodyfikuj definicję klasy `Tree24Node`, aby zawierała wskaźnik do rodzica węzła:

`Tree24Node<E>`	
`elements: ArrayList<E>`	Lista tablicowa do przechowywania elementów.
`child: ArrayList<Tree24Node<E>>`	Lista tablicowa do przechowywania wskaźników do dzieci.
parent: Tree24Node<E>	Wskazuje rodzica danego węzła.
`+Tree24()`	Tworzy pusty węzeł.
`+Tree24(o: E)`	Tworzy węzeł z początkowym elementem.

Dodaj w klasie `Tree24` dwie nowe metody:

```
public Tree24Node<E> getParent(Tree24Node<E> node)
// Zwraca rodzica podanego węzła
public ArrayList<Tree24Node<E>> getPath(Tree24Node<E> node)
// Zwraca listę tablicową ze ścieżką z podanego węzła do korzenia
```

Napisz program testowy, który dodaje do drzewa liczby 1, 2, …, 100 i wyświetla ścieżki do wszystkich liści.

*****35.7.** *Klasa `BTree`*. Zaprojektuj i zaimplementuj klasę reprezentującą B-drzewa.

ROZDZIAŁ

36

Drzewa czerwono-czarne

Cele

- Omówienie drzew czerwono-czarnych (podrozdział 36.1).
- Przekształcanie drzew czerwono-czarnych w drzewa 2-3-4 i w drugą stronę (podrozdział 36.2).
- Zaprojektowanie klasy `RBTree` rozszerzającej klasę `BST` (podrozdział 36.3).
- Wstawianie elementów do drzew czerwono-czarnych i rozwiązywanie w razie potrzeby problemu dwóch kolejnych czerwonych węzłów (podrozdział 36.4).
- Usuwanie elementów z drzew czerwono-czarnych i rozwiązywanie w razie potrzeby problemu „podwójnie czarnych" węzłów (podrozdział 36.5).
- Implementowanie i testowanie klasy `RBTree` (podrozdziały 36.6 i 36.7).
- Porównanie wydajności drzew AVL, drzew 2-3-4 i klasy `RBTree` (podrozdział 36.8).

36.1. Wprowadzenie

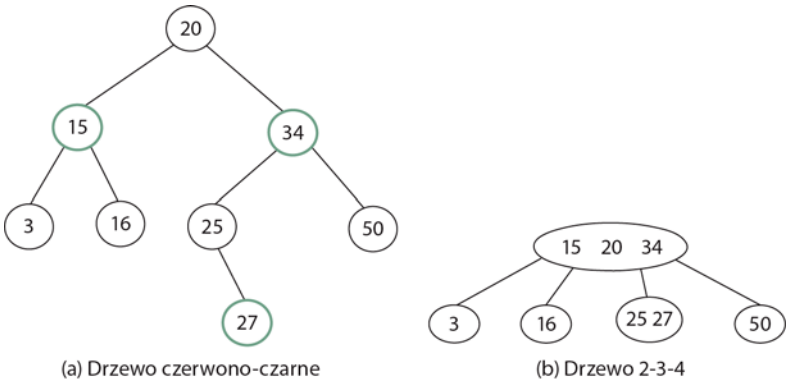
Drzewo czerwono-czarne jest zrównoważonym binarnym drzewem poszukiwań opartym na drzewie 2-3-4. Drzewo czerwono-czarne odpowiada drzewu 2-3-4[1].

Każdy węzeł w drzewie czerwono-czarnym ma *kolor* — czerwony lub czarny (rysunek 36.1a). Węzeł jest nazywany *zewnętrznym*, jeśli jego lewe lub prawe poddrzewo jest puste. Zauważ, że liść zawsze jest węzłem zewnętrznym, ale nie każdy węzeł zewnętrzny jest liściem. Na przykład węzeł 25 jest zewnętrzny, ale nie jest liściem. *Długość ścieżki czarnych węzłów* dla węzła to liczba czarnych węzłów w ścieżce z tego węzła do korzenia. Na przykład długość ścieżki węzłów czarnych dla węzłów 25 i 27 wynosi 2.

Drzewo czerwono-czarne ma następujące cechy:

1. Korzeń jest czarny.
2. Kolejne węzły nie mogą być czerwone.
3. Wszystkie węzły zewnętrzne mają tę samą długość ścieżki węzłów czarnych.

Drzewo czerwono-czarne z rysunku 36.1a ma te trzy cechy. Drzewo czerwono-czarne można przekształcić w drzewo 2-3-4 i na odwrót. Na rysunku 36.1b pokazane jest drzewo 2-3-4 odpowiadające drzewu czerwono-czarnemu z rysunku 36.1a.



Rysunek 36.1. Drzewo czerwono-czarne można zapisać jako drzewo 2-3-4 i na odwrót

36.2. Konwersja między drzewami czerwono-czarnymi a drzewami 2-3-4

W tym podrozdziale opisane jest powiązanie między drzewami czerwono-czarnymi a drzewami 2-3-4.

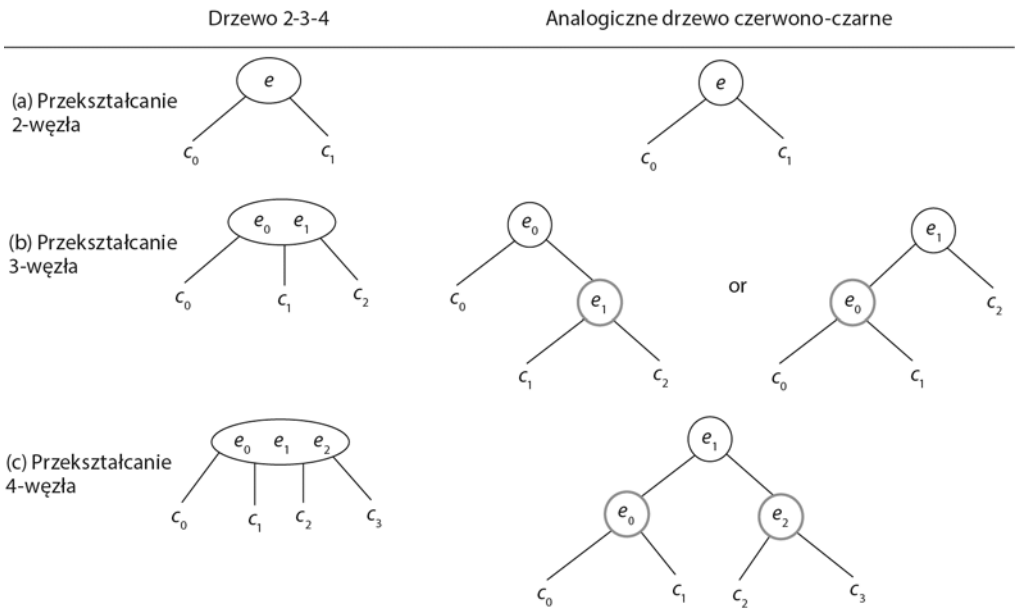
Możesz zaprojektować algorytmy wstawiania i usuwania dla drzew czerwono-czarnych, nie znając drzew 2-3-4. Jednak powiązanie między tymi typami drzew pozwala lepiej zrozumieć strukturę drzew czerwono-czarnych i operacje na nich. Dlatego w tym podrozdziale omówione są zależności między tymi rodzajami drzew.

[1] W witrynie poświęconej oryginalnemu wydaniu książki jest to rozdział 43. — *przyp. tłum.*

Aby przekształcić drzewo czerwono-czarne w drzewo 2-3-4, scal każdy czerwony węzeł z jego rodzicem, aby utworzyć 3- lub 4-węzeł. Na przykład czerwone węzły 15 i 34 należy scalić z ich rodzicem, aby powstał 4-węzeł, a czerwony węzeł 27 należy połączyć z jego rodzicem, by uzyskać 3-węzeł (rysunek 36.1b).

W celu przekształcenia drzewa 2-3-4 w drzewo czerwono-czarne w każdym węźle *u* wykonaj następujące transformacje:

1. Jeśli *u* jest 2-węzłem, użyj koloru czarnego (rysunek 36.2a).
2. Jeżeli *u* jest 3-węzłem z elementami e_0 i e_1, możliwe są dwa przekształcenia: albo użyj e_0 jako rodzica e_1, albo użyj e_1 jako rodzica e_0. W obu sytuacjach użyj dla rodzica koloru czarnego, a dla dziecka — czerwonego (rysunek 36.2b).
3. Jeśli *u* jest 4-węzłem z elementami e_0, e_1 i e_2, użyj e_1 jako rodzica e_0 i e_2. Dla e_1 użyj koloru czarnego, a dla e_0 i e_2 czerwonego (rysunek 36.2c).



Rysunek 36.2. Węzeł w drzewie 2-3-4 można przekształcić na węzły drzewa czerwono-czarnego

Przeprowadź teraz transformację drzewa 2-3-4 z rysunku 36.1b. Po przekształceniu 4-węzła drzewo wygląda tak jak na rysunku 36.3a. Rysunek 36.3b przedstawia drzewo po transformacji 3-węzła. Zauważ, że można ją przeprowadzić na różne sposoby, dlatego konwersja drzewa 2-3-4 na drzewo czerwono-czarne *nie jest unikatowa*. Inne drzewo powstałe po przekształceniu 3-węzła przedstawia rysunek 36.3c.

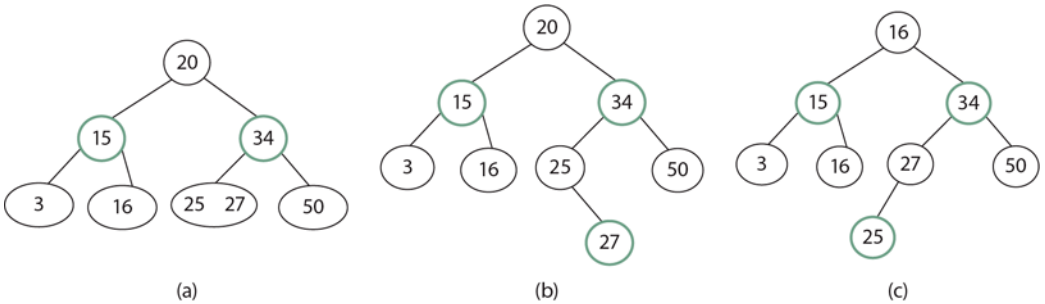
Można udowodnić, że pokazane przekształcenia prowadzą do otrzymania drzewa czerwono-czarnego spełniającego wszystkie trzy wymogi.

Właściwość 1. Korzeń jest czarny.

Dowód: Jeśli korzeniem drzewa 2-3-4 jest 2-węzeł, korzeń drzewa czerwono-czarnego jest czarny. Jeżeli korzeniem drzewa 2-3-4 jest 3- lub 4-węzeł, w wyniku transformacji korzeniem staje się czarny rodzic.

Właściwość 2. Dwa kolejne węzły nie mogą być czerwone.

Dowód: Ponieważ rodzic czerwonego węzła zawsze jest czarny, dwa kolejne węzły nie mogą być czerwone.



RYSUNEK 36.3. Konwersja drzewa 2-3-4 na drzewo czerwono-czarne nie jest unikatowa

Właściwość 3. Wszystkie węzły zewnętrzne mają tę samą długość ścieżki węzłów czarnych.

Dowód: Gdy przekształcasz węzeł drzewa 2-3-4 na węzły drzewa czerwono-czarnego, otrzymujesz jeden czarny węzeł i zero, jeden lub dwa węzły czerwone będące dziećmi węzła czarnego (zależnie od tego, czy pierwotny węzeł to 2-, 3- czy 4-węzeł). Tylko liść z drzewa 2-3-4 może dawać zewnętrzne węzły drzewa czerwono-czarnego. Ponieważ drzewo 2-3-4 jest idealnie zrównoważone, liczba czarnych węzłów w każdej ścieżce z korzenia do węzła zewnętrznego jest taka sama.

36.2.1. Czym jest drzewo czerwono-czarne? Czym jest węzeł zewnętrzny? Czym jest długość ścieżki węzłów czarnych?

36.2.2. Opisz właściwości drzewa czerwono-czarnego.

36.2.3. Jak przekształcić drzewo czerwono-czarne w drzewo 2-3-4? Czy taka konwersja jest unikatowa?

36.2.4. Jak przekształcić drzewo 2-3-4 w drzewo czerwono-czarne? Czy taka konwersja jest unikatowa?

36.3. Projektowanie klas drzew czerwono-czarnych

W tym podrozdziale zaprojektujesz klasę dla drzew czerwono-czarnych.

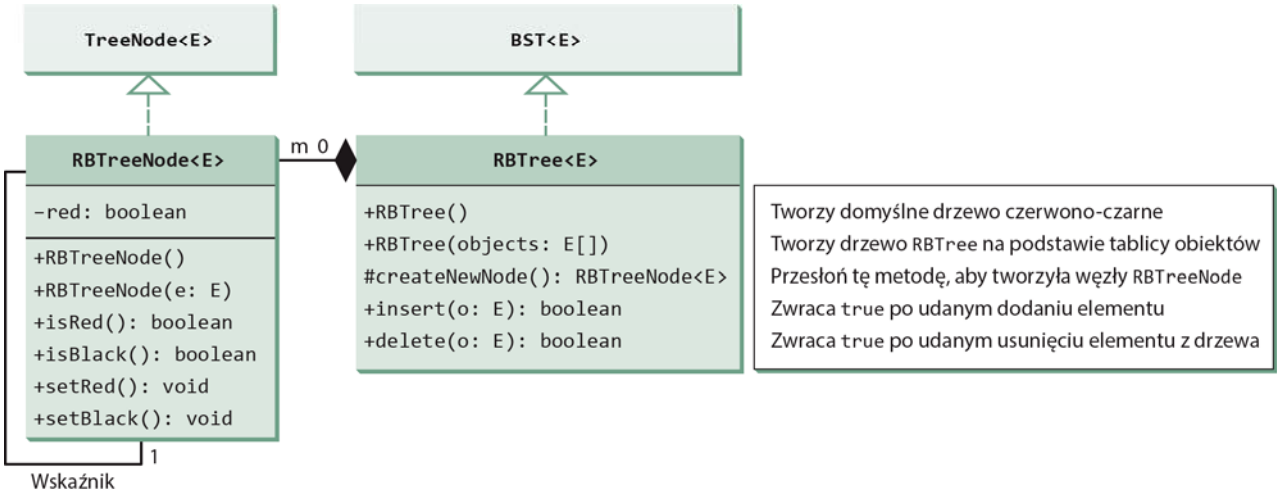
Drzewo czerwono-czarne jest binarnym drzewem poszukiwań. Dlatego na potrzeby drzew czerwono-czarnych możesz zdefiniować klasę `RBTree` rozszerzającą klasę BST (rysunek 36.4). Klasy `BST` i `TreeNode` są zdefiniowane w punkcie 26.2.5.

Każdy węzeł w drzewie czerwono-czarnym ma kolor. Ponieważ używane są kolory czerwony i czarny, do ich zapisu można użyć typu `boolean`. Klasa `RBTreeNode` może rozszerzać klasę `BST.TreeNode` o właściwość reprezentującą kolor. Dla wygody można też udostępnić metody do sprawdzania i ustawiania koloru. Zauważ, że `TreeNode` jest statyczną klasą wewnętrzną w klasie `BST`. `RBTreeNode` będzie statyczną klasą wewnętrzną w klasie `RBTree`. Klasa `TreeNode` zawiera pola `element`, `left` i `right`, które są dziedziczone w klasie `RBTreeNode`. Dlatego w klasie `RBTreeNode` znajdą się cztery pola (rysunek 36.5).

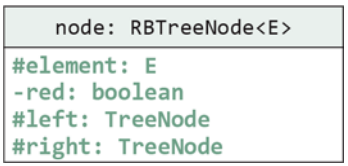
W klasie `BST` metoda `createNewNode()` tworzy obiekt typu `TreeNode`. W klasie `RBTree` ta metoda jest przesłonięta, aby tworzyła węzły `RBTreeNode`. Zauważ, że w klasie `BST` typ wartości zwracanej przez metodę `createNewNode()` to `TreeNode`, ale w klasie `RBTree` typem zwracanej wartości jest `RBTreeNode`. Jest to poprawne, ponieważ `RBTreeNode` jest podtypem typu `TreeNode`.

Wyszukiwanie elementu w drzewie czerwono-czarnym przebiega tak samo jak w zwykłym drzewie binarnym. Dlatego metoda `search` zdefiniowana w klasie `BST` działa także w klasie `RBTree`.

Metody `insert` i `delete` są przesłonięte, aby wstawiały, usuwały i kolorowały elementy oraz w razie potrzeby zmieniały strukturę drzewa, aby zachować trzy właściwości drzew czerwono-czarnych.



RYSUNEK 36.4. Klasa RBTree rozszerza klasę BST o nową implementację metod insert i delete



RYSUNEK 36.5. Klasa RBTreeNode zawiera pola element, red, left i right

Uwaga edukacyjna

Otwórz stronę *http://liveexample.pearsoncmg.com/dsanimation/RBTree.html*, aby zobaczyć, jak działają drzewa czerwono-czarne (rysunek 36.6).

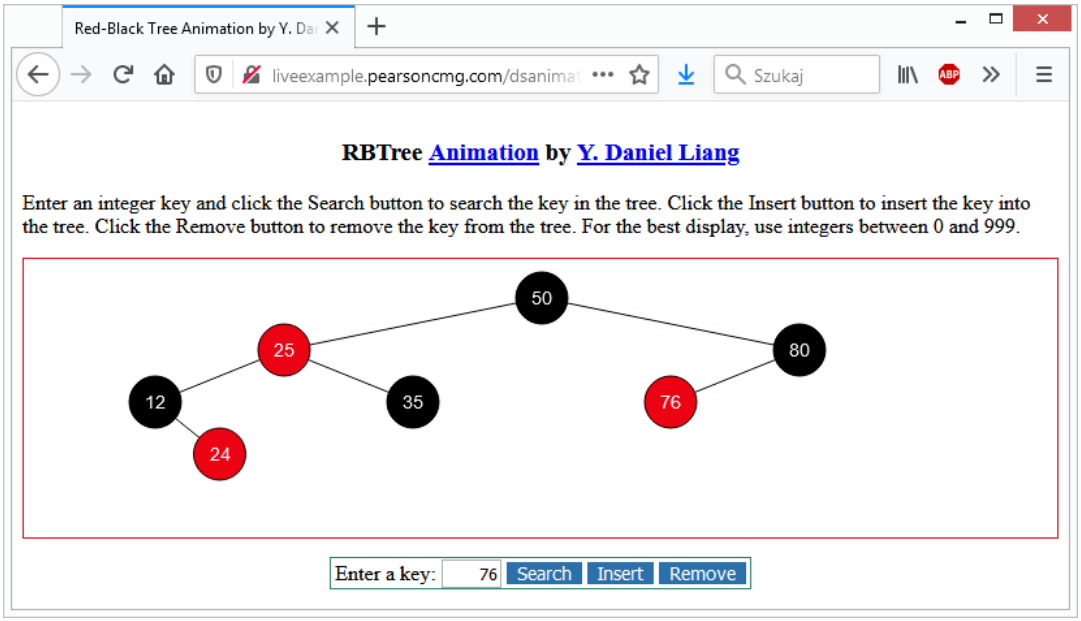
36.4. Przesłanianie metody insert

W tym podrozdziale omówione jest wstawianie elementów w drzewach czerwono-czarnych.

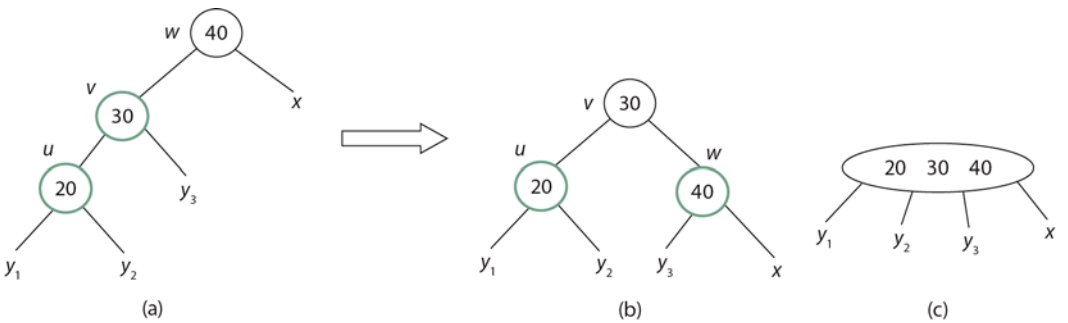
Nowy element zawsze jest wstawiany jako liść. Jeśli nowym elementem jest korzeń, należy przypisać mu kolor czarny. W przeciwnym razie należy użyć koloru czerwonego. Jeżeli rodzic nowego węzła jest czerwony, narusza to drugą właściwość drzew czerwono-czarnych. Jest to problem *dwóch kolejnych czerwonych węzłów*.

Niech *u* oznacza wstawiany nowy węzeł, *v* rodzica węzła *u*, *w* rodzica węzła *v*, a *x* brata węzła *v*. Aby rozwiązać problem dwóch kolejnych czerwonych węzłów, rozważ dwa scenariusze:

Scenariusz 1. Węzeł *x* jest czarny lub równy `null`. Są cztery możliwe konfiguracje węzłów *u*, *v*, *w* i *x* (rysunki 36.7a, 36.8a, 36.9a i 36.10a). Wtedy *u*, *v* i *w* tworzą 4-węzeł w analogicznym drzewie 2-3-4 (rysunki 36.7c, 36.8c, 36.9c i 36.10c), ale są reprezentowanie nieprawidłowo w drzewie czerwono-czarnym. Aby poprawić błąd, należy zmienić strukturę i kolory trzech węzłów *u*, *v* i *w* (rysunki 36.7b, 36.8b, 36.9b i 36.10b). Zauważ, że *x*, y_1, y_2 i y_3 mogą być równe `null`.

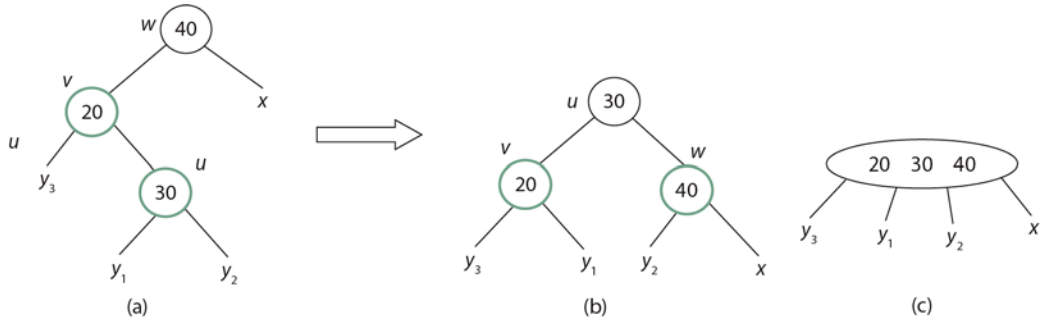


Rysunek 36.6. Narzędzie do wyświetlania animacji umożliwia wstawianie, usuwanie i wyszukiwanie elementów w drzewach czerwono-czarnych

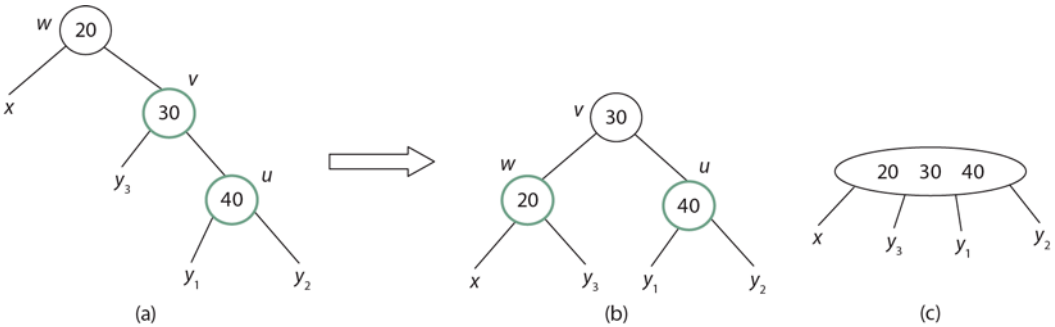


Rysunek 36.7. Scenariusz 1.1: u < v < w

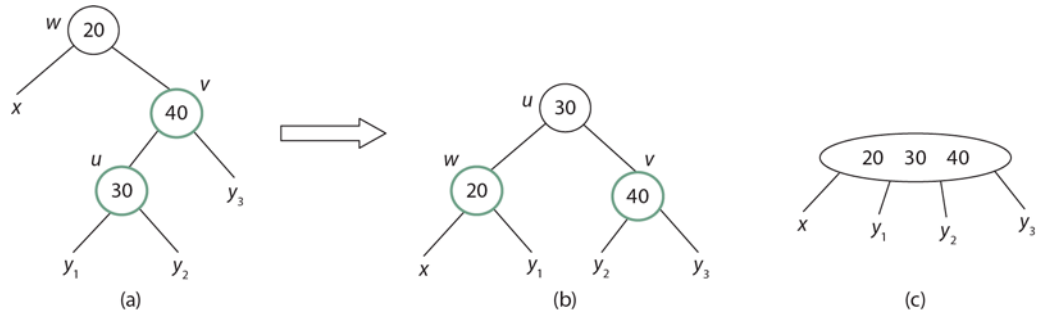
Scenariusz 2. Węzeł *x* jest czerwony. Są cztery możliwe konfiguracje węzłów *u*, *v*, *w* i *x* (rysunki 36.11a, 36.11b, 36.11c i 36.11d). Wszystkie one odpowiadają przepełnieniu w powiązanym 4-węźle drzewa 2-3-4 (rysunek 36.12a). Należy podzielić węzeł, aby rozwiązać błąd przepełnienia w drzewie 2-3-4 (rysunek 36.12b), i odpowiednio pokolorować węzły, by wyeliminować problem w drzewie czerwono-czarnym. Węzły *w* i *u* pokoloruj na czerwono, a dla dwojga dzieci węzła *w* użyj koloru czarnego. Niech *u* będzie lewym dzieckiem węzła *v* (rysunek 36.11a). Rysunek 36.11c przedstawia węzły po zmianie kolorów. Teraz *w* jest czerwony, a jeśli jego rodzic jest czarny, problem dwóch kolejnych czerwonych węzłów został wyeliminowany. W przeciwnym razie problem pojawia się w węźle *w*. Należy rekurencyjnie zastosować ten sam proces, by wyeliminować problem dwóch kolejnych czerwonych węzłów w węźle *w*.



Rysunek 36.8. Scenariusz 1.2: v < u < w



Rysunek 36.9. Scenariusz 1.3: w < v < u

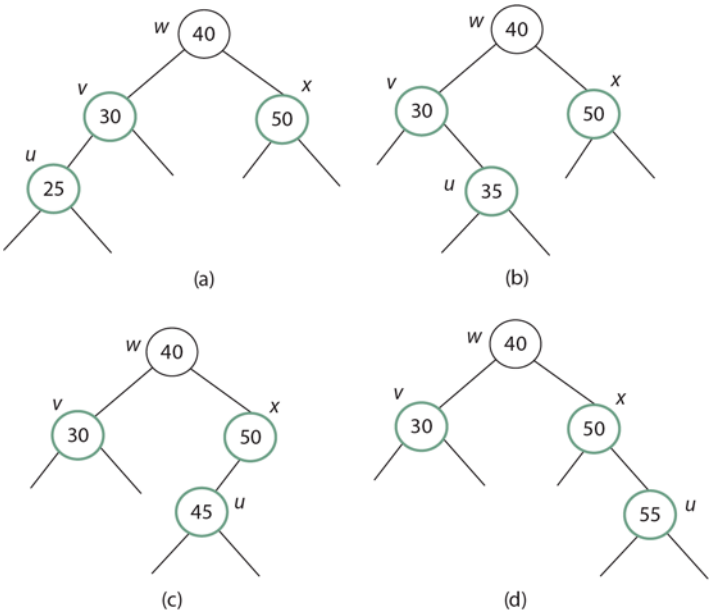


Rysunek 36.10. Scenariusz 1.4: w < u < v

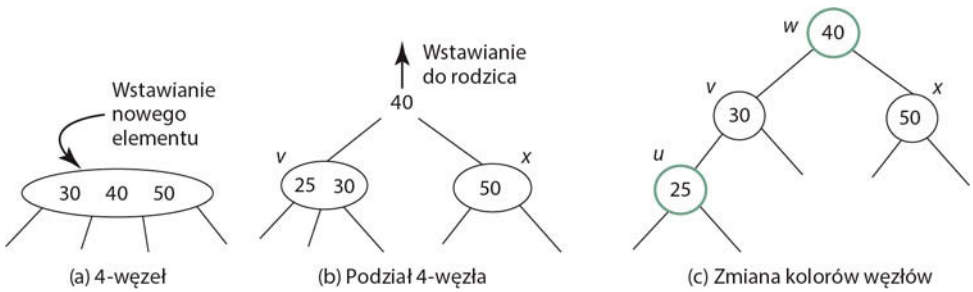
Szczegółowy algorytm wstawiania elementu jest opisany na listingu 36.1.

Listing 36.1. Wstawianie elementu w drzewie czerwono-czarnym

```
1 public boolean insert(E e) {
2   boolean successful = super.insert(e);
3   if (!successful)
4     return false; // e znajduje się już w drzewie
5   else {
6     ensureRBTree(e);
```



Rysunek 36.11. W scenariuszu 2. występują cztery możliwe konfiguracje



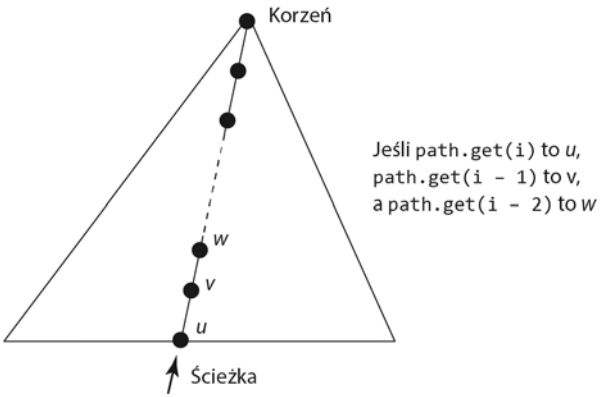
Rysunek 36.12. Podział 4-węzła to odpowiednik zmiany kolorów węzłów w drzewie czerwono-czarnym

```
 7   }
 8
 9   return true; // e został wstawiony
10 }
11
12 /** Gwarantuje zachowanie właściwości drzewa czerwono-czarnego */
13 private void ensureRBTree(E e) {
14   Pobieranie ścieżki z korzenia do elementu e.
15   int i = path.size() – 1; // Indeks bieżącego węzła w ścieżce
16   Pobieranie węzłów u i v ze ścieżki; u to węzeł zawierający e, a v
17     jest rodzicem u.
18   Kolorowanie u na czerwono;
19
20   if (u == root) // Jeśli e jest wstawiany w korzeniu, należy pokolorować korzeń na czarno
21     u.setBlack();
```

```
22   else if (v.isRed())
23     fixDoubleRed(u, v, path, i); // Usuwanie problemu dwóch kolejnych czerwonych węzłów w u
24 }
25
26 /** Rozwiązuje problem dwóch kolejnych czerwonych węzłów w węźle u */
27 private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
28     ArrayList<TreeNode<E>> path, int i) {
29   Pobierz w ze ścieżki (w jest dziadkiem węzła u).
30
31   // Pobieranie węzła x (brata węzła v)
32   RBTreeNode<E> x = (w.left == v) ?
33     (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);
34
35   if (x == null || x.isBlack()) {
36     // Scenariusz 1. x (brat węzła v) jest czarny
37     if (w.left == v && v.left == u) {
38       // Scenariusz 1.1. u < v < w, zmiana struktury i kolorów węzłów
39     }
40     else if (w.left == v && v.right == u) {
41       // Scenariusz 1.2. v < u < w, zmiana struktury i kolorów węzłów
42     }
43     else if (w.right == v && v.right == u) {
44       // Scenariusz 1.3. w < v < u, zmiana struktury i koloru węzłów
45     }
46     else {
47       // Scenariusz 1.4. w < u < v, zmiana struktury i koloru węzłów
48     }
49   }
50   else { // Scenariusz 2. x (brat węzła v) jest czerwony
51     Pokoloruj w i u na czerwono.
52     Pokoloruj dwoje dzieci węzła w na czarno.
53
54     if (w jest korzeniem) {
55       Pokoloruj w na czarno;
56     }
57     else if (rodzic węzła w jest czerwony) {
58       // Przechodzenie wzdłuż ścieżki, aby usunąć nowy problem dwóch kolejnych czerwonych węzłów
59       u = w;
60       v = rodzic węzła w;
61       fixDoubleRed(u, v, path, i – 2); // i – 2 powoduje przejście w górę ścieżki
62     }
63   }
64 }
```

Metoda `insert(E e)` (wiersze 1. – 10.) wywołuje metodę `insert` z klasy `BST`, aby utworzyć nowy liść z elementem (wiersz 2.). Jeśli dany element już znajduje się w drzewie, metoda zwraca `false` (wiersz 4.). W przeciwnym razie wywoływana jest metoda `ensureRBTree(e)` (wiersz 6.), która gwarantuje zachowanie właściwości drzewa czerwono-czarnego.

Metoda `ensureRBTree(E e)` (wiersze 13. – 24.) pobiera ścieżkę z korzenia do węzła `e` (wiersz 14.), co ilustruje rysunek 36.13. Ta ścieżka jest ważna w implementacji algorytmu. Ze ścieżki pobierane są węzły `u` i `v` (wiersze 16. i 17.) Jeśli `u` jest korzeniem, należy pokolorować go na czarno (wiersze 20. i 21.). Jeżeli `v` jest czerwony, w węźle `u` występuje problem dwóch kolejnych czerwonych węzłów. Należy wywołać metodę `fixDoubleRed`, aby wyeliminować ten problem.



RYSUNEK 36.13. Ścieżka składa się z węzłów od u do korzenia

Metoda `fixDoubleRed` (wiersze 27. – 63.) eliminuje problem dwóch kolejnych czerwonych węzłów. Ta metoda najpierw pobiera ze ścieżki węzły `w` (rodzica węzła `v`; wiersz 29.) i `x` (brata węzła `v`; wiersze 32. i 33.). Jeśli `x` jest pusty lub czarny, należy zmienić strukturę i kolor trzech węzłów `u`, `v` i `w`, aby usunąć problem (wiersze 35. – 49.). Jeżeli `x` jest czerwony, należy zmienić kolor węzłów `u`, `v`, `w` i `x` (wiersze 51. i 52.). Jeśli `w` jest korzeniem, pokoloruj go na czarno (wiersze 54. – 56.). Gdy rodzic węzła `w` jest czerwony, w `w` występuje problem dwóch kolejnych czerwonych węzłów. Wywołaj metodę `fixDoubleRed` dla nowych węzłów `u` i `v`, aby usunąć ten problem (wiersz 61.). Zauważ, że teraz `i - 2` wskazuje nowy węzeł `u` w ścieżce. Ta zmiana jest niezbędna, aby znaleźć w ścieżce nowy węzeł `w` i jego rodzica.

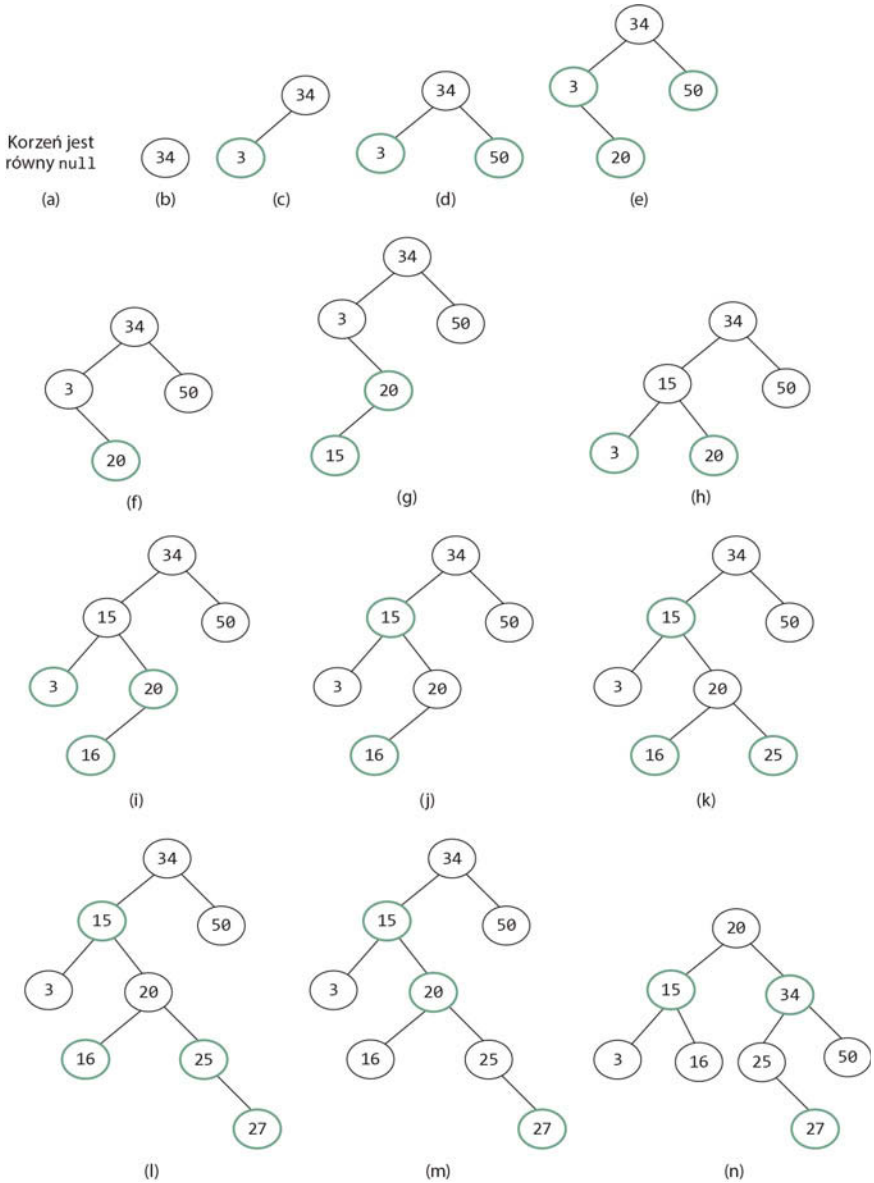
Na rysunku 36.14 pokazane jest wstawianie elementów 34, 3, 50, 20, 15, 16, 25 i 27 do pustego drzewa czerwono-czarnego. Przy wstawianiu 20 (d) zachodzi scenariusz 2.; należy zmienić kolor 3 i 50 na czarny. Przy wstawianiu 15 (g) zachodzi scenariusz 1.4; należy zmienić strukturę i kolor węzłów 15, 20 i 3. Przy wstawianiu 16 (i) zachodzi scenariusz 2.; należy zmienić kolor 3 i 20 na czarny, a 15 i 16 na czerwony. Przy wstawianiu 27 (l) zachodzi scenariusz 2.; należy zmienić kolor 16 i 25 na czarny, a 20 i 27 na czerwony. Teraz w węźle 20 występuje problem dwóch kolejnych czerwonych węzłów. Zachodzi scenariusz 1.2 i należy zmienić strukturę oraz kolor węzłów. Nowe drzewo jest pokazane na rysunku (n).

36.5. Przesłanianie metody delete

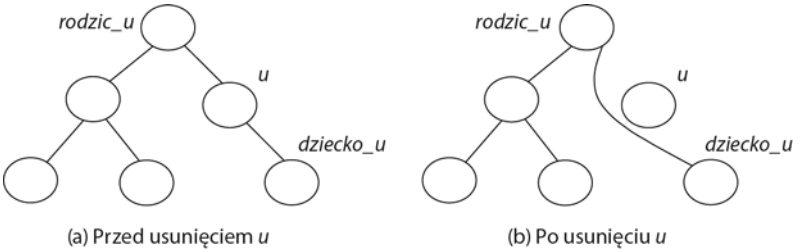
W tym podrozdziale opisane jest usuwanie elementów z drzew czerwono-czarnych.

Aby usunąć element w drzewie czerwono-czarnym, najpierw znajdź ten element w drzewie, aby zlokalizować zawierający go węzeł. Jeśli elementu nie ma w drzewie, metoda zwraca `false`. Niech *u* będzie węzłem zawierającym usuwany element. Jeżeli *u* jest węzłem wewnętrznym mającym lewe i prawe dziecko, należy znaleźć pierwszy od prawej węzeł w lewym poddrzewie węzła *u* i zastąpić element w *u* elementem ze znalezionego węzła. Dalej omawiane jest tylko usuwanie węzłów zewnętrznych.

Niech *u* będzie usuwanym węzłem zewnętrznym. Ponieważ *u* jest węzłem zewnętrznym, ma najwyżej jedno dziecko (*dziecko_u*; `childOfu` w kodzie). To dziecko może być równe `null`. Niech *rodzic_u* (`parentOfu` w kodzie) będzie rodzicem węzła *u* (rysunek 36.15a). Usuń *u*, łącząc *dziecko_u* z *rodzic_u* (rysunek 36.15b).



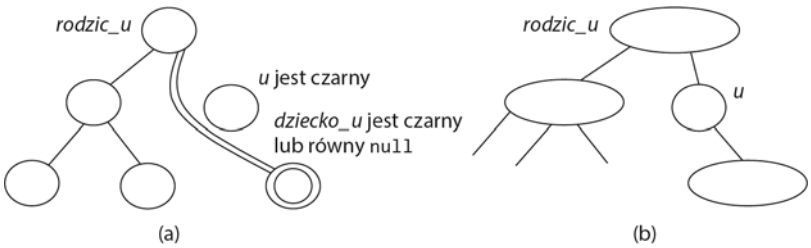
Rysunek 36.14. Wstawianie elementu do drzewa czerwono-czarnego: (a) początkowo drzewo jest puste; (b) wstawianie 34; (c) wstawianie 3; (d) wstawianie 50; (e) wstawienie 20 powoduje problem dwóch kolejnych czerwonych węzłów; (f) po zmianie koloru (scenariusz 2.); (g) wstawienie 15 powoduje problem dwóch kolejnych czerwonych węzłów; (h) po zmianie struktury i kolorów (scenariusz 1.4); (i) wstawienie 16 powoduje problem dwóch kolejnych czerwonych węzłów; (j) po zmianie kolorów (scenariusz 2.); (k) wstawienie 25; (l) wstawienie 27 powoduje problem dwóch kolejnych czerwonych węzłów w węźle 27; (m) po zmianie kolorów problem dwóch kolejnych czerwonych węzłów występuje w węźle 20 (scenariusz 2.); (n) po zmianie struktury i kolorów (scenariusz 1.2)



Rysunek 36.15. u jest węzłem zewnętrznym, a dziecko_u może być równe null

Rozważ następujący scenariusz:

- Jeśli *u* jest czerwony, nie trzeba robić nic więcej.
- Jeżeli *u* jest czarny, a *dziecko_u* czerwony, należy pokolorować *dziecko_u* na czarno, aby zachować długość ścieżki czarnych węzłów.
- W przeciwnym razie należy oznaczyć *dziecko_u* jako węzeł *podwójnie czarny* (rysunek 36.16a). Jest to problem *węzła podwójnie czarnego*. Oznacza on, że długość ścieżki czarnych węzłów jest za mała o 1, co wynika z usunięcia czarnego węzła u.



Rysunek 36.16. (a) dziecko_u jest węzłem podwójnie czarnym; (b) u odpowiada pustemu węzłowi w drzewie 2-3-4

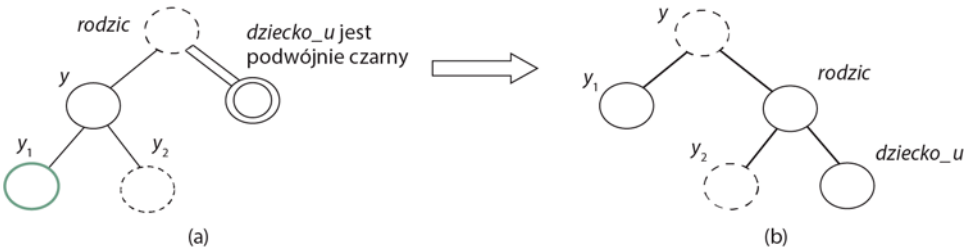
Węzeł podwójnie czarny w drzewie czerwono-czarnym to odpowiednik pustego węzła u (niedopełnienia) w analogicznym drzewie 2-3-4 (rysunek 36.16b). By rozwiązać problem węzła podwójnie czarnego, należy przeprowadzić przeniesienie i złączanie. Rozważ trzy sytuacje:

Scenariusz 1. Brat *y* węzła *dziecko_u* jest czarny i ma czerwone dziecko. Są tu cztery możliwe konfiguracje (rysunki 36.17a, 36.18a, 36.19a, 36.20a). Kółko z przerywanych kresek oznacza, że węzeł jest albo czerwony, albo czarny. Aby wyeliminować problem podwójnie czarnego węzła, zmień strukturę i kolor węzłów (rysunki 36.17b, 36.18b, 36.19b, 36.20b).

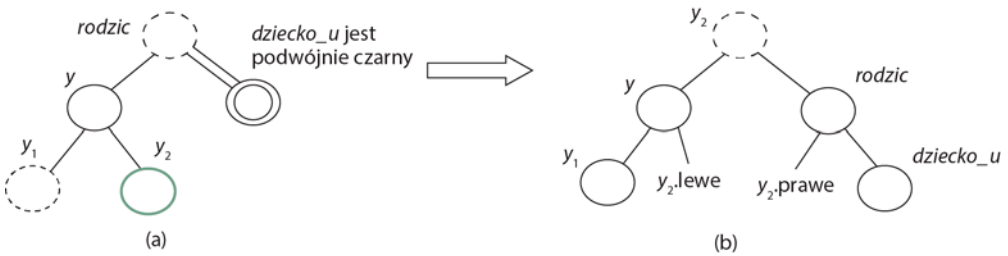
Uwaga

Scenariusz 1. odpowiada operacji *przeniesienia* w drzewie 2-3-4. na rysunku 36.21a pokazane jest drzewo 2-3-4 odpowiadające drzewu z rysunku 36.17a. To drzewo 2-3-4 jest przekształcane za pomocą przeniesienia w drzewo z rysunku 36.21b.

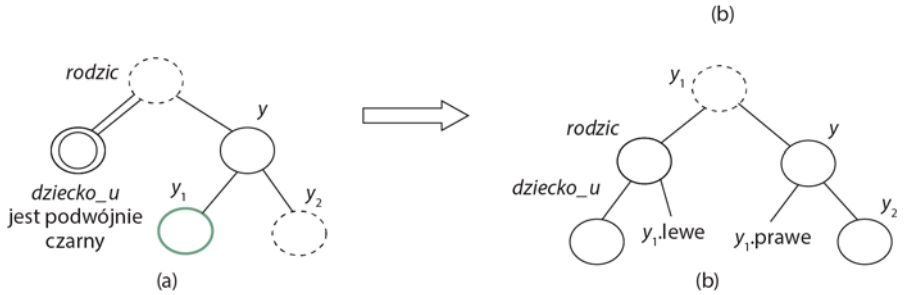
Scenariusz 2. Brat *y* węzła *dziecko_u* jest czarny, a jego dzieci są czarne lub równe null. Wtedy należy zmienić kolor węzła *y* na czerwony. Jeżeli *rodzic* jest czerwony, należy zmienić jego kolor na czarny, co kończy pracę (rysunek 36.22). Jeżeli *rodzic* jest czarny, należy oznaczyć go jako węzeł podwójnie czarny (rysunek 36.23). Problem *przechodzi* w ten sposób do rodzica.



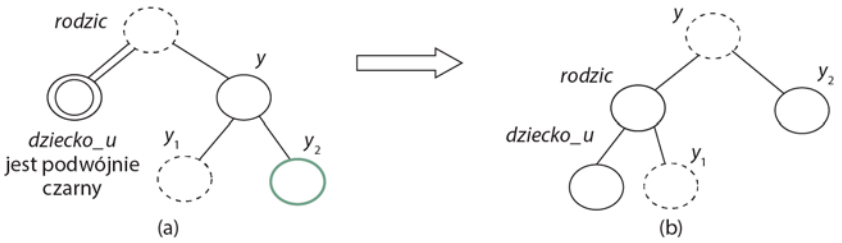
Rysunek 36.17. Scenariusz 1.1. Brat y węzła dziecko_u jest czarny, a y1 jest czerwony



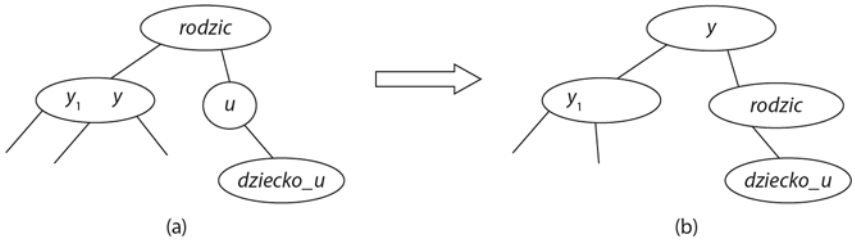
Rysunek 36.18. Scenariusz 1.2. Brat y węzła dziecko_u jest czarny, a y2 jest czerwony



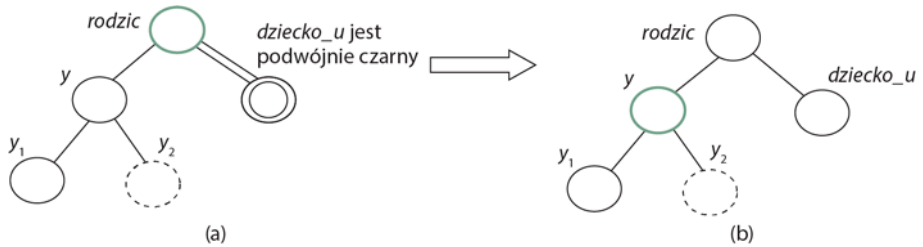
Rysunek 36.19. Scenariusz 1.3. Brat y węzła dziecko_u jest czarny, a y1 jest czerwony



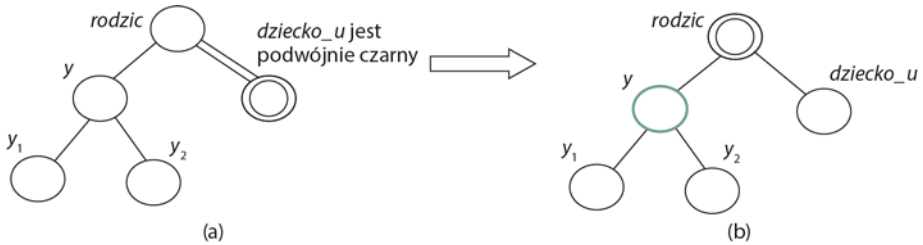
Rysunek 36.20. Scenariusz 1.4. Brat y węzła dziecko_u jest czarny, a y2 jest czerwony



Rysunek 36.21. Scenariusz 1. odpowiada operacji przeniesienia w analogicznym drzewie 2-3-4



Rysunek 36.22. Scenariusz 2. Jeśli rodzic jest czerwony, zmiana koloru eliminuje problem podwójnie czarnego węzła



Rysunek 36.23. Scenariusz 2. Jeśli rodzic jest czarny, zmiana koloru powoduje przeniesienie problemu podwójnie czarnego węzła do rodzica

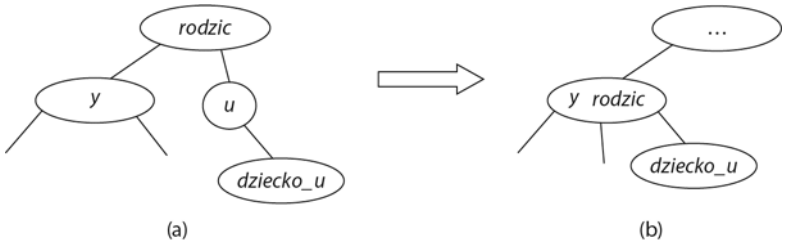
Uwaga

Na rysunkach 36.22 i 36.23 *rodzic_u* jest prawym dzieckiem węzła *rodzic*. Gdy *rodzic_u* jest lewym dzieckiem węzła *rodzic*, zmiana kolorów odbywa się w identyczny sposób.

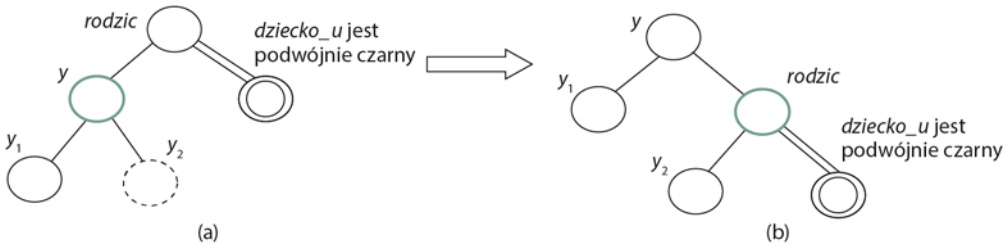
Uwaga

Scenariusz 2. odpowiada operacji *złączania* dla drzew 2-3-4. Rysunek 36.24a przedstawia drzewo 2-3-4 odpowiadające drzewu z rysunku 36.22a. To drzewo 2-3-4 jest przekształcane w drzewo z rysunku 36.24b za pomocą złączenia.

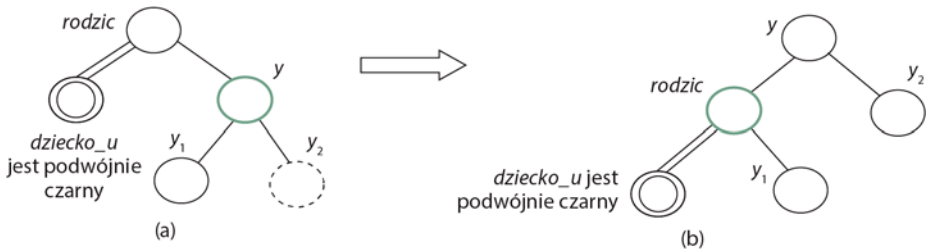
Scenariusz 3. Brat *y* węzła *dziecko_u* jest czerwony. Wtedy należy przeprowadzić *dostosowanie*. Jeśli *y* jest lewym dzieckiem węzła *rodzic*, niech *y1* i *y2* będą lewym i prawym dzieckiem węzła *y* (rysunek 36.25). Jeżeli *y* jest prawym dzieckiem węzła *rodzic*, niech *y1* i *y2* będą lewym i prawym dzieckiem węzła *y* (rysunek 36.26). W obu przypadkach pokoloruj węzeł *y* na czarno, a węzeł *rodzic* na czerwono. Węzeł *dziecko_u* wciąż jest podwójnie czarny. Po dostosowaniu brat węzła *dziecko_u* jest czarny i można przejść do scenariusza 1. lub 2.



RYSUNEK 36.24. Scenariusz 2. odpowiada złączeniu w analogicznym drzewie 2-3-4



RYSUNEK 36.25. Scenariusz 3.1. Węzeł y jest czerwonym lewym dzieckiem węzła rodzic



RYSUNEK 36.26. Scenariusz 3.2. Węzeł y jest czerwonym prawym dzieckiem węzła rodzic

W scenariuszu 1. pojedyncza zmiana struktury i koloru eliminuje problem podwójnie czarnego węzła. W scenariuszu 2. problem podwójnie czarnego węzła nie może wystąpić, ponieważ *rodzic* jest teraz czerwony. Dlatego pojedyncze uwzględnienie scenariusza 1. lub 2. kończy scenariusz 3.

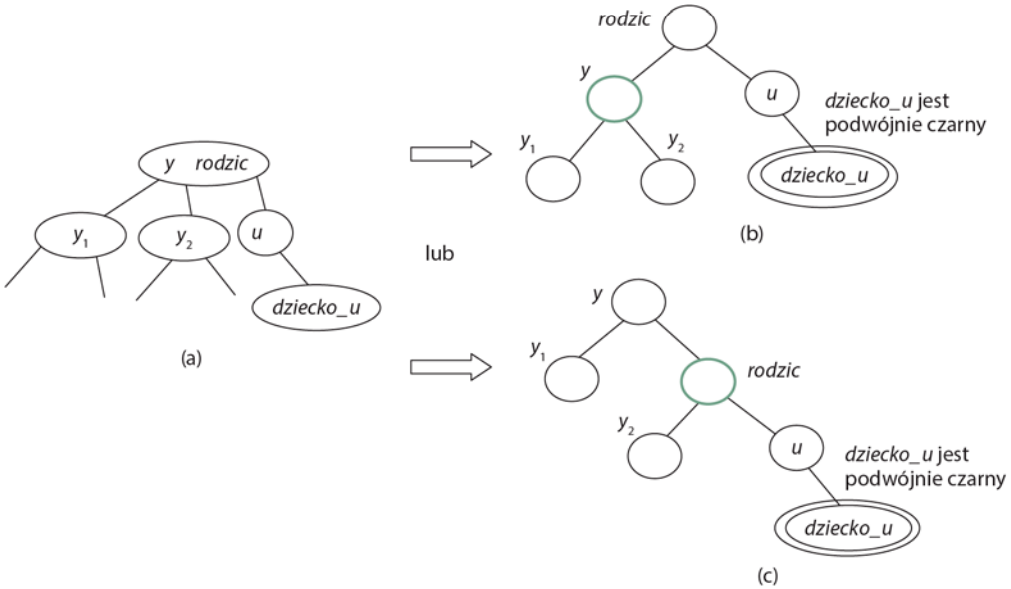
Uwaga

Scenariusz 3. wynika z tego, że 3-węzeł można przekształcić w drzewo czerwono-czarne na dwa sposoby (rysunek 36.27).

Na listingu 36.2 pokazany jest szczegółowy algorytm usuwania elementu oparty na wcześniejszych analizach.

LISTING 36.2. Usuwanie elementu drzewa czerwono-czarnego

```
1 public boolean delete(E e) {
2   Znajdowanie usuwanego węzła
3   if (węzła nie znaleziono)
4     return false;
5
```



RYSUNEK 36.27. 3-węzeł można przekształcić w węzły drzewa czerwono-czarnego na dwa sposoby

```
  if (węzeł jest wewnętrzny) {
    Znajdowanie pierwszego od prawej węzła w poddrzewie węzła;
    Zastępowanie elementu w węźle elementem z węzła pierwszego od prawej;
    Teraz do usunięcia wyznaczony jest węzeł pierwszy od prawej;
  }

  Pobieranie ścieżki z korzenia do usuwanego węzła;

  // Usuwanie ostatniego węzła w ścieżce i w razie potrzeby rozwiązanie problemu na następnym poziomie
  deleteLastNodeInPath(path);

  size--; // Po usunięciu 1 elementu
  return true; // Element został usunięty
}

/** Usuwanie ostatniego węzła w ścieżce */
public void deleteLastNodeInPath(ArrayList<TreeNode<e>> path) {
  Pobierz ostatni węzeł u w ścieżce;
  Pobierz parentOfu i grandparentOfu w ścieżce;
  Pobierz childOfu na podstawie u;
  Usuń węzeł u. Powiąż childOfu z parentOfu

  // Zmiana koloru węzłów i w razie potrzeby usunięcie problemu podwójnie czarnego węzła
  if (childOfu == root || u.isRed())
    return; // Jeśli childOfu jest korzeniem lub u jest czerwony, można zakończyć pracę
  else if (childOfu != null && childOfu.isRed())
    childOfu.setBlack(); // Przypisz kolor czarny; gotowe
  else // u jest czarny, childOfu jest czarny lub równy null
    // Wyeliminuj problem podwójnie czarnego węzła w parentOfu
    fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
```

```
}

/** Eliminuje problem podwójnie czarnego węzła w rodzicu */
private void fixDoubleBlack(
    RBTreeNode<E> grandparent, RBTreeNode<E> parent,
    RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
  Pobierz y, y1 i y2

  if (y.isBlack() && y1 != null && y1.isRed()) {
    if (parent.right == db) {
      // Scenariusz 1.1. y jest czarnym lewym bratem, a y1 jest czerwony
      Zmień strukturę i kolory rodzica, y i y1, aby usunąć problem;
    }
    else {
      // Scenariusz 1.3. y to czarny prawy brat, a y1 to czerwony
      Zmień strukturę i kolory rodzica, y1 i y, aby usunąć problem;
    }
  }
  else if (y.isBlack() && y2 != null && y2.isRed()) {
    if (parent.right == db) {
      // Scenariusz 1.2. y to czarny lewy brat, a y2 to czerwony
      Zmień strukturę i kolory rodzica, y2, i y, aby usunąć problem;
    }
    else {
      // Scenariusz 1.4. y to czarny prawy brat, a y2 to czerwony
      Zmień strukturę i kolory rodzica, y i y2, aby usunąć problem;
    }
  }
  else if (y.isBlack()) {
    // Scenariusz 2. y jest czarny, a jego dzieci są czarne lub równe null
    Zmień kolor y na czerwony;

    if (parent.isRed())
      parent.setBlack(); // Gotowe
    else if (parent != root) {
      // Przeniesienie problemu podwójnie czarnego węzła do rodzica.
      // Rekurencyjne usuwanie nowych wystąpień tego problemu
      db = parent;
      parent = grandparent;
      grandparent =
        (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
      fixDoubleBlack(grandparent, parent, db, path, i - 1);
    }
  }
  else if (y.isRed()) {
    if (parent.right == db) {
      // Scenariusz 3.1. y jest czerwonym lewym dzieckiem węzła parent
      parent.left = y2;
      y.right = parent;
    }
    else {
      // Scenariusz 3.2. y jest czerwonym prawym dzieckiem węzła parent
      parent.right = y.left;
      y.left = parent;
    }
```

```
91
92      parent.setRed(); // Węzeł parent jest kolorowany na czerwono
93      y.setBlack(); // Węzeł y jest kolorowany na czarno
94      connectNewParent(grandparent, parent, y); // y to nowy węzeł parent
95      fixDoubleBlack(y, parent, db, path, i - 1);
96    }
97  }
```

Metoda `delete(E e)` (wiersze 1. – 19.) znajduje węzeł zawierający element `e` (wiersz 2.). Jeśli nie ma takiego węzła, należy zwrócić `false` (wiersze 3. i 4.). Jeżeli węzeł zawierający element `e` jest węzłem wewnętrznym, trzeba znaleźć pierwszy od prawej węzeł w lewym poddrzewie i zastąpić `e` elementem z węzła pierwszego od prawej (wiersze 6. – 9.). Teraz do usunięcia przeznaczony jest węzeł zewnętrzny. Należy pobrać ścieżkę z korzenia do tego węzła (wiersz 12.), wywołać metodę `deleteLastNodeInPath(path)` w celu usunięcia ostatniego elementu ścieżki i zadbać o to, by drzewo zachowało właściwości drzewa czerwono-czarnego (wiersz 15.).

Metoda `deleteLastNodeInPath` (wiersze 22. – 36.) pobiera ostatni węzeł `u` oraz węzły `parentOfu`, `grandparentOfu` i `childOfu` (wiersze 23. – 26.). Jeśli `childOfu` jest korzeniem lub `u` jest czerwony, drzewo jest poprawne (wiersze 29. i 30.). Jeżeli `childOfu` jest czerwony, należy zmienić kolor na czarny (wiersze 31. i 32.). Gotowe. W przeciwnym razie `u` jest czarny, a `childOfu` jest czarny lub równy `null`. Wywołaj metodę `fixDoubleBlack`, aby wyeliminować problem podwójnie czarnego węzła (wiersz 35.).

Metoda `fixDoubleBlack` (wiersze 39. – 97.) eliminuje problem podwójnie czarnego węzła. Należy pobrać węzły `y`, `y1` i `y2` (wiersz 42.). Węzeł `y` jest bratem węzła podwójnie czarnego. Węzły `y1` i `y2` to lewe i prawe dziecko `y`. Rozważ trzy sytuacje:

1. Jeśli `y` jest czarny, a jedno z dzieci to węzeł czerwony, problem podwójnie czarnego węzła można rozwiązać za pomocą pojedynczej zmiany struktury i kolorów tak jak w scenariuszu 1. (wiersze 44. – 63.).
2. Jeżeli `y` jest czarny, a jego dzieci są czarne lub równe `null`, zmień kolor `y` na czerwony. Gdy rodzic (`parent`) węzła `y` jest czarny, oznacz rodzica jako nowy węzeł podwójnie czarny i rekurencyjnie wywołaj metodę `fix↪DoubleBlack` (wiersz 77.).
3. Jeśli `y` jest czerwony, dostosuj węzły, aby `parent` był dzieckiem węzła `y` (wiersze 84. i 89.), zmień kolor węzła `parent` na czerwony i węzła `y` na czarny (wiersze 92. i 93.). Węzeł `y` ma być nowym rodzicem (wiersz 94.). Rekurencyjnie wywołaj metodę `fixDoubleBlack` dla tego samego węzła podwójnie czarnego, ale z innym kolorem węzła `parent` (wiersz 95.).

Na rysunku 36.28 zobrazowane jest usuwanie elementów. Aby usunąć 50 z drzewa z rysunku 36.28a, uwzględnij scenariusz 1.2 (rysunek 36.28b). Nowe drzewo po zmianie struktury i kolorów pokazane jest na rysunku 36.28c.

Przy usuwaniu 20 na rysunku 36.28c 20 jest węzłem wewnętrznym zastępowanym przez 16 (rysunek 36.28d). Teraz uwzględniany jest scenariusz 2. z usuwaniem węzła pierwszego od prawej (rysunek 36.28e). Zmiana koloru węzłów pozwala uzyskać nowe drzewo (rysunek 36.28f).

Przy usuwaniu 15 połącz węzeł 3 z 20 i pokoloruj 3 na czarno (rysunek 36.28g) — gotowe.

Rysunek 36.28j przedstawia nowe drzewo uzyskane po usunięciu 25. Teraz usuń 16. Uwzględnij scenariusz 2. (rysunek 36.28k). Nowe drzewo jest pokazane na rysunku 36.28l.

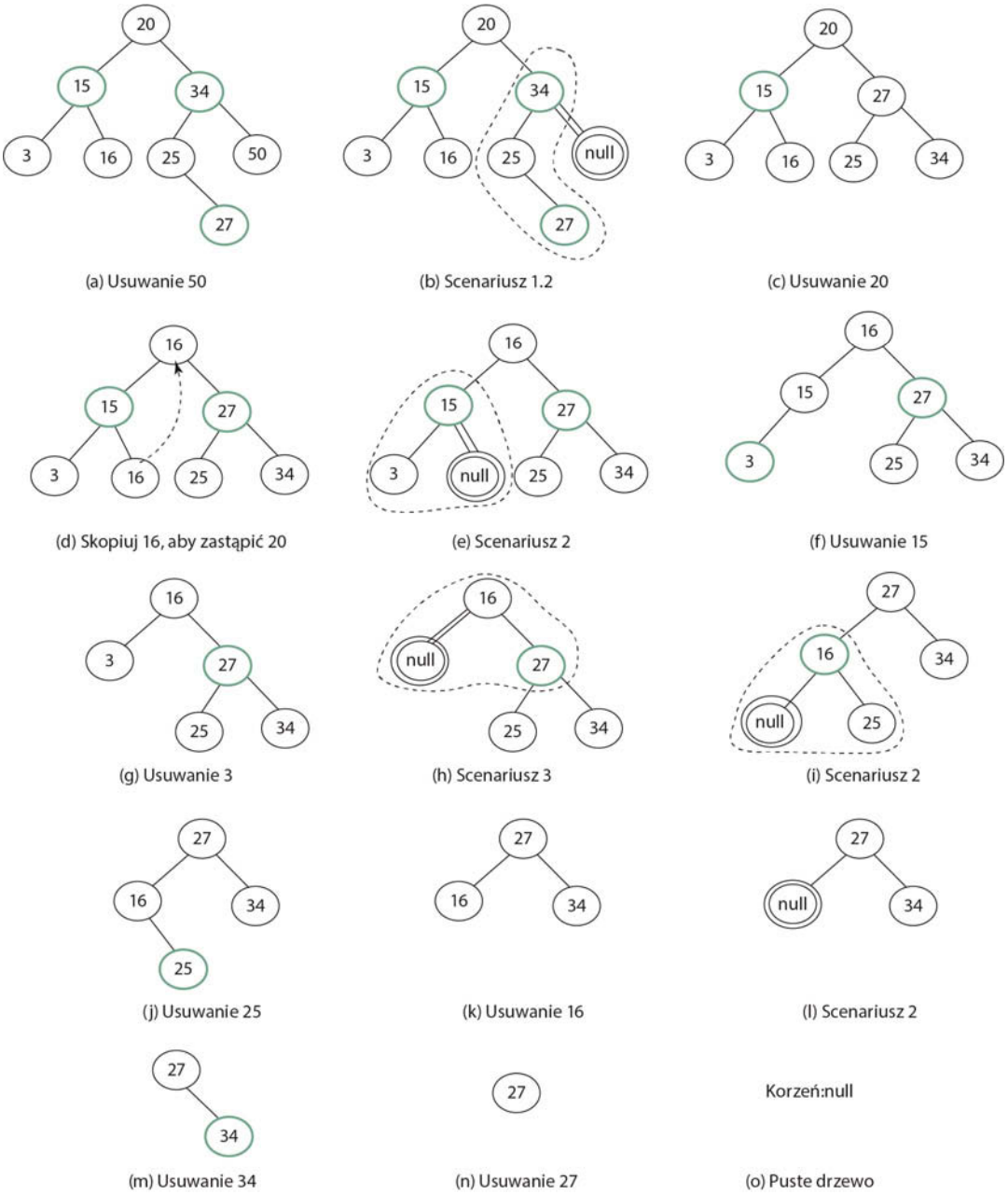
Rysunek 36.28m przedstawia nowe drzewo uzyskane po usunięciu 34.

Rysunek 36.28n ilustruje nowe drzewo otrzymane po usunięciu 27.

36.5.1. Jakie pola ma klasa `RBTreeNode`?

36.5.2. Jak wstawić element do drzewa czerwono-czarnego? Jak rozwiązać problem dwóch kolejnych czerwonych węzłów?

36.5.3. Jak usunąć element z drzewa czerwono-czarnego? Jak rozwiązać problem podwójnie czarnego węzła?



RYSUNEK 36.28. Usuwanie elementów z drzewo czerwono-czarnego

36.5.4. Opisz zmiany w drzewie w trakcie wstawiania elementów 1, 2, 3, 4, 10, 9, 7, 5, 8 i 6 (w tej kolejności).

36.5.5. Używane jest drzewo zbudowane w poprzednim ćwiczeniu. Opisz zmiany w drzewie w trakcie usuwania elementów 1, 2, 3, 4, 10, 9, 7, 5, 8 i 6 (w tej kolejności).

36.6. Implementowanie klasy RBTree

W tym podrozdziale zaimplementujesz klasę `RBTree`.

Listing 36.3 przedstawia kompletną implementację klasy `RBTree`.

LISTING 36.3. RBTree.java

```
 1 import java.util.ArrayList;
 2
 3 public class RBTree<E extends Comparable<E>> extends BST<E> {
 4   /** Tworzy domyślne drzewo czerwono-czarne */
 5   public RBTree() {
 6   }
 7
 8   /** Tworzy drzewo czerwono-czarne na podstawie tablicy elementów */
 9   public RBTree(E[] elements) {
10     super(elements);
11   }
12
13   @Override /** Przesłanianie metody createNewNode, aby tworzyła węzły RBTreeNode */
14   protected RBTreeNode<E> createNewNode(E e) {
15     return new RBTreeNode<E>(e);
16   }
17
18   @Override /** Przesłanianie metody insert, aby w razie potrzeby
19                 wyważała drzewo */
20   public boolean insert(E e) {
21     boolean successful = super.insert(e);
22     if (!successful)
23       return false; // e już znajduje się w drzewie
24     else {
25       ensureRBTree(e);
26     }
27
28     return true; // e został wstawiony
29   }
30
31   /** Gwarantuje, że drzewo jest drzewem czerwono-czarnym */
32   private void ensureRBTree(E e) {
33     // Pobieranie ścieżki z korzenia do elementu e
34     ArrayList<TreeNode<E>> path = path(e);
35
36     int i = path.size() - 1; // Indeks bieżącego węzła w ścieżce
37
38     // u jest ostatnim węzłem w ścieżce i zawiera element e
39     RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));
40
41     // v (jeśli istnieje) jest rodzicem węzła u
42     RBTreeNode<E> v = (u == root) ? null :
43       (RBTreeNode<E>)(path.get(i - 1));
44
45     u.setRed(); // Można pokolorować u na czerwono
46
47     if (u == root) // Jeśli e został wstawiony w korzeniu, pokoloruj korzeń na czarno
```

```
48      u.setBlack();
49    else if (v.isRed())
50      fixDoubleRed(u, v, path, i); // Usuwanie problemu dwóch kolejnych czerwonych węzłów w u
51  }
52
53  /** Usuwa problem dwóch kolejnych czerwonych węzłów w u */
54  private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
55      ArrayList<TreeNode<E>> path, int i) {
56    // w jest dziadkiem u
57    RBTreeNode<E> w = (RBTreeNode<E>)(path.get(i - 2));
58    RBTreeNode<E> parentOfw = (w == root) ? null :
59      (RBTreeNode<E>)path.get(i - 3);
60
61    // Pobieranie węzła x (brata węzła v)
62    RBTreeNode<E> x = (w.left == v) ?
63      (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);
64
65    if (x == null || x.isBlack()) {
66      // Scenariusz 1. x (brat węzła v) jest czarny
67      if (w.left == v && v.left == u) {
68        // Scenariusz 1.1. u < v < w, zmiana struktury i koloru węzłów
69        restructureRecolor(u, v, w, w, parentOfw);
70
71        w.left = v.right; // v.right to y3 z rysunku 36.6
72        v.right = w;
73      }
74      else if (w.left == v && v.right == u) {
75        // Scenariusz 1.2. v < u < w, zmiana struktury i koloru węzłów
76        restructureRecolor(v, u, w, w, parentOfw);
77        v.right = u.left;
78        w.left = u.right;
79        u.left = v;
80        u.right = w;
81      }
82      else if (w.right == v && v.right == u) {
83        // Scenariusz 1.3. w < v < u, zmiana struktury i koloru węzłów
84        restructureRecolor(w, v, u, w, parentOfw);
85        w.right = v.left;
86        v.left = w;
87      }
88      else {
89        // Scenariusz 1.4. w < u < v, zmiana struktury i koloru węzłów
90        restructureRecolor(w, u, v, w, parentOfw);
91        w.right = u.left;
92        v.left = u.right;
93        u.left = w;
94        u.right = v;
95      }
96    }
97    else { // Scenariusz 2. x (brat węzła v) jest czerwony
98      // Zmiana koloru węzłów
99      w.setRed();
100     u.setRed();
101     ((RBTreeNode<E>)(w.left)).setBlack();
102     ((RBTreeNode<E>)(w.right)).setBlack();
```

```
103
104       if (w == root) {
105         w.setBlack();
106       }
107       else if (((RBTreeNode<E>)parentOfw).isRed()) {
108         // Przejście w górę ścieżki, aby rozwiązać nowy problem dwóch kolejnych czerwonych węzłów
109         u = w;
110         v = (RBTreeNode<E>)parentOfw;
111         fixDoubleRed(u, v, path, i - 2); // i – 2 powoduje przejście w górę ścieżki
112       }
113     }
114   }
115
116   /** Łączenie b z parentOfw i zmiana koloru a, b, c na podstawie zależności a < b < c */
117   private void restructureRecolor(RBTreeNode<E> a, RBTreeNode<E> b,
118       RBTreeNode<E> c, RBTreeNode<E> w, RBTreeNode<E> parentOfw) {
119     if (parentOfw == null)
120       root = b;
121     else if (parentOfw.left == w)
122       parentOfw.left = b;
123     else
124       parentOfw.right = b;
125
126     b.setBlack(); // b staje się korzeniem w poddrzewie
127     a.setRed(); // a staje się lewym dzieckiem węzła b
128     c.setRed(); // c staje się prawym dzieckiem węzła b
129   }
130
131   @Override /** Usuwa element z drzewa RBTree.
132               * Zwraca true po udanym usunięciu elementu.
133               * Zwraca false, jeśli element nie znajduje się w drzewie */
134   public boolean delete(E e) {
135     // Znajdowanie usuwanego węzła
136     TreeNode<E> current = root;
137     while (current != null) {
138       if (e.compareTo(current.element) < 0) {
139         current = current.left;
140       }
141       else if (e.compareTo(current.element) > 0) {
142         current = current.right;
143       }
144       else
145         break; // Element is znajduje się w drzewie wskazywanym przez current
146     }
147
148     if (current == null)
149       return false; // Element nie występuje w drzewie
150
151     java.util.ArrayList<TreeNode<E>> path;
152
153     // Węzeł current jest węzłem wewnętrznym
154     if (current.left != null && current.right != null) {
155       // Znajdowanie pierwszego od prawej węzła w lewym poddrzewie węzła current
156       TreeNode<E> rightMost = current.left;
157       while (rightMost.right != null) {
```

```
158         rightMost = rightMost.right; // Należy przechodzić w prawo
159       }
160
161       path = path(rightMost.element); // Pobieranie ścieżki przed zastępowaniem elementów
162
163       // Zastępowanie elementu z current elementem z rightMost
164       current.element = rightMost.element;
165     }
166     else
167       path = path(e); // Pobieranie ścieżki do węzła current
168
169     // Usuwanie ostatniego węzła w ścieżce i w razie potrzeby przejście w górę ścieżki
170     deleteLastNodeInPath(path);
171
172     size--; // Po usunięciu 1 elementu
173     return true; // Element został usunięty
174   }
175
176   /** Usuwa ostatni węzeł w ścieżce */
177   public void deleteLastNodeInPath(ArrayList<TreeNode<E>> path) {
178     int i = path.size() - 1; // Indeks do węzła w ścieżce
179     // u jest ostatnim węzłem w ścieżce
180     RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));
181     RBTreeNode<E> parentOfu = (u == root) ? null :
182       (RBTreeNode<E>)(path.get(i - 1));
183     RBTreeNode<E> grandparentOfu = (parentOfu == null ||
184       parentOfu == root) ? null :
185       (RBTreeNode<E>)(path.get(i - 2));
186     RBTreeNode<E> childOfu = (u.left == null) ?
187       (RBTreeNode<E>)(u.right) : (RBTreeNode<E>)(u.left);
188
189     // Usuwanie węzła u; łączenie childOfu z parentOfu
190     connectNewParent(parentOfu, u, childOfu);
191
192     // Zmiana koloru węzłów i usunięcie problemu podwójnie czarnego węzła, jeśli jest to konieczne
193     if (childOfu == root || u.isRed())
194       return; // Jeśli childOfu jest korzeniem lub u jest czerwony, można zakończyć pracę
195     else if (childOfu != null && childOfu.isRed())
196       childOfu.setBlack(); // Kolorowanie childOfu na czarno — gotowe
197     else // u jest czarny, childOfu jest czarny lub równy null
198       // Usunięcie problemu podwójnie czarnego węzła w parentOfu
199       fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
200   }
201
202   /** Usunięcie problemu w rodzicu */
203   private void fixDoubleBlack(
204       RBTreeNode<E> grandparent, RBTreeNode<E> parent,
205       RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
206     // Pobieranie y, y1 i y2
207     RBTreeNode<E> y = (parent.right == db) ?
208       (RBTreeNode<E>)(parent.left) : (RBTreeNode<E>)(parent.right);
209     RBTreeNode<E> y1 = (RBTreeNode<E>)(y.left);
210     RBTreeNode<E> y2 = (RBTreeNode<E>)(y.right);
211
212     if (y.isBlack() && y1 != null && y1.isRed()) {
```

```
213       if (parent.right == db) {
214         // Scenariusz 1.1. y jest czarnym lewym bratem, a y1 jest czerwony
215         connectNewParent(grandparent, parent, y);
216         recolor(parent, y, y1); // Dostosowanie kolorów
217
218         // Dostosowanie wskaźników do dzieci
219         parent.left = y.right;
220         y.right = parent;
221       }
222       else {
223         // Scenariusz 1.3. y jest czarnym lewym bratem, a y1 jest czerwony
224         connectNewParent(grandparent, parent, y1);
225         recolor(parent, y1, y); // Dostosowanie kolorów
226
227         // Dostosowanie wskaźników do dzieci
228         parent.right = y1.left;
229         y.left = y1.right;
230         y1.left = parent;
231         y1.right = y;
232       }
233     }
234     else if (y.isBlack() && y2 != null && y2.isRed()) {
235       if (parent.right == db) {
236         // Scenariusz 1.2. y jest czarnym lewym bratem, a y2 jest czerwony
237         connectNewParent(grandparent, parent, y2);
238         recolor(parent, y2, y); // Dostosowanie kolorów
239
240         // Dostosowanie wskaźników do dzieci
241         y.right = y2.left;
242         parent.left = y2.right;
243         y2.left = y;
244         y2.right = parent;
245       }
246       else {
247         // Scenariusz 1.4. y jest czarnym prawym bratem, a y2 jest czerwony
248         connectNewParent(grandparent, parent, y);
249         recolor(parent, y, y2); // Dostosowanie kolorów
250
251         // Adjust child links
252         y.left = parent;
253         parent.right = y1;
254       }
255     }
256     else if (y.isBlack()) {
257       // Scenariusz 2. y jest czarny, a jego dzieci są czarne lub równe null
258       y.setRed(); // Kolorowanie y na czerwono
259       if (parent.isRed())
260         parent.setBlack(); // Gotowe
261       else if (parent != root) {
262         // Przenoszenie problemu podwójnie czarnego węzła do rodzica;
263         // rekurencyjne eliminowanie nowych wystąpień problemu podwójnie czarnego węzła
264         db = parent;
265         parent = grandparent;
266         grandparent =
267           (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
```

```
268       fixDoubleBlack(grandparent, parent, db, path, i - 1);
269     }
270   }
271   else { // y.isRed()
272     if (parent.right == db) {
273       // Scenariusz 3.1. y jest czerwonym lewym dzieckiem węzła parent
274       parent.left = y2;
275       y.right = parent;
276     }
277     else {
278       // Scenariusz 3.2. y jest czerwonym prawym dzieckiem węzła parent
279       parent.right = y.left;
280       y.left = parent;
281     }
282
283     parent.setRed(); // Kolorowanie węzła parent na czerwono
284     y.setBlack(); // Kolorowanie węzła y na czarno
285     connectNewParent(grandparent, parent, y); // y jest nowym węzłem parent
286     fixDoubleBlack(y, parent, db, path, i - 1);
287   }
288 }
289
290 /** Zmiana koloru węzłów parent, newParent i c. Scenariusz 1. usuwania */
291 private void recolor(RBTreeNode<E> parent,
292     RBTreeNode<E> newParent, RBTreeNode<E> c) {
293   // Zachowanie koloru węzła parent w węźle newParent
294   if (parent.isRed())
295     newParent.setRed();
296   else
297     newParent.setBlack();
298
299   // c i parent stają się czarnymi dziećmi węzła newParent
300   parent.setBlack();
301   c.setBlack();
302 }
303
304 /** Łączenie newParent z grandParent */
305 private void connectNewParent(RBTreeNode<E> grandparent,
306     RBTreeNode<E> parent, RBTreeNode<E> newParent) {
307   if (parent == root) {
308     root = newParent;
309     if (root != null)
310       newParent.setBlack();
311   }
312   else if (grandparent.left == parent)
313     grandparent.left = newParent;
314   else
315     grandparent.right = newParent;
316 }
317
318 @Override /** Odwiedzanie węzłów w porządku preorder od poddrzewa */
319   protected void preorder(TreeNode<E> root) {
320     if (root == null) return;
321     System.out.print(root.element +
322       (((RBTreeNode<E>)root).isRed() ? " (czerwony) " : " (czarny) "));
```

```
323       preorder(root.left);
324       preorder(root.right);
325     }
326
327     /** RBTreeNode to klasa TreeNode plus pole reprezentujące kolor */
328     protected static class RBTreeNode<E extends Comparable<E>> extends
329         BST.TreeNode<E> {
330       private boolean red = true; // Określa kolor węzła
331
332       public RBTreeNode(E e) {
333         super(e);
334       }
335
336       public boolean isRed() {
337         return red;
338       }
339
340       public boolean isBlack() {
341         return !red;
342       }
343
344       public void setBlack() {
345         red = false;
346       }
347
348       public void setRed() {
349         red = true;
350       }
351
352       int blackHeight;
353     }
354   }
```

Klasa `RBTree` rozszerza klasę `BST`. `RBTree`, podobnie jak `BST`, ma konstruktor bezargumentowy; tworzy on puste drzewo `RBTree` (wiersze 5. i 6.). Dostępny jest też konstruktor tworzący początkowe drzewo `RBTree` na podstawie tablicy elementów (wiersze 9. – 11.).

Metoda `createNewNode()` z klasy `BST` tworzy węzły `TreeNode`. W klasie `RBTree` ta metoda jest przesłonięta i zwraca węzły `RBTreeNode` (wiersze 14. – 16.). Ta metoda jest wywoływana w metodzie `insert` z klasy `BST` w celu utworzenia węzła.

W klasie `RBTree` przesłaniana jest też metoda `insert` (wiersze 20. – 29.). Ta metoda najpierw wywołuje metodę `insert` z klasy `BST`, a następnie wywołuje metodę `ensureRBTree(e)` (wiersz 25.), aby zagwarantować, że po wstawieniu nowego elementu drzewo nadal będzie drzewem czerwono-czarnym.

Metoda `ensureRBTree(e)` najpierw pobiera ścieżkę węzłów prowadzącą z korzenia do elementu `e` (wiersz 34.). Z tej ścieżki pobierane są węzły `u` i `v` (jest to rodzic węzła `u`). Jeśli `u` jest korzeniem, należy pokolorować go na czarno (wiersze 47. i 48.). Jeżeli `v` jest czerwony, należy wywołać metodę `fixDoubleRed`, aby usunąć problem dwóch kolejnych węzłów czerwonych w `u` i `v` (wiersze 49. i 50.).

Metoda `fixDoubleRed(u, v, path, i)` usuwa problem dwóch kolejnych czerwonych węzłów w węźle `u`. Najpierw pobiera węzły `w` (jest to rodzic węzła `u` ze ścieżki; wiersz 57.), `parentOfw` (jeśli istnieje; wiersze 58. i 59.) i `x` (jest to brat węzła `v`; wiersze 62. i 63.). Jeśli `x` jest czarny lub równy `null`, należy rozważyć cztery scenariusze, aby rozwiązać problem dwóch kolejnych czerwonych węzłów (wiersze 67. – 96.). Gdy `x` jest czerwony, należy pokolorować `w` i `u` na czerwono, a dzieci węzła `w` na czarno (wiersze 101. – 104.). Jeżeli `w` jest korzeniem, pokoloruj `w` na

czarno (wiersze 104. – 106.). W przeciwnym razie należy przejść w górę ścieżki i rozwiązać nowy problem dwóch kolejnych czerwonych węzłów (wiersze 109. – 111.).

Metoda `delete(E e)` z klasy `RBTree` jest przesłonięta w wierszach 134. – 174. Ta metoda znajduje węzeł zawierający element `e` (wiersze 136. – 146.). Jeśli otrzymany węzeł to `null`, dany element nie występuje w drzewie (wiersze 148. i 149.). Metoda uwzględnia dwa scenariusze:

- Dla węzłów wewnętrznych należy znaleźć pierwszy od prawej węzeł w lewym poddrzewie (wiersze 156. – 159.), pobrać ścieżkę z korzenia do tego pierwszego od prawej węzła (wiersz 161.) i zastąpić element w bieżącym węźle elementem z węzła pierwszego od prawej (wiersz 164.).
- Dla węzłów zewnętrznych należy pobrać ścieżkę z korzenia do danego węzła (wiersz 167.).

Usuwany jest ostatni węzeł w ścieżce. Wywołaj metodę `deleteLastNodeInPath(path)`, aby usunąć ten węzeł i zapewnić, że drzewo po usunięciu węzła pozostanie drzewem czerwono-czarnym (wiersz 170.).

Metoda `deleteLastNodeInPath(path)` najpierw pobiera węzły `parentOfu`, `grandparentOfu` i `childOfu` (wiersze 180. – 187.). Węzeł `u` jest ostatnim węzłem w ścieżce. Zapisz `childOfu` jako dziecko `parentOfu` (wiersz 190.). Skutkuje to usunięciem `u` z drzewa. Należy rozważyć trzy scenariusze:

- Jeśli `childOfu` jest korzeniem lub jest czerwony, nie trzeba nic więcej robić (wiersze 193. i 194.).
- W przeciwnym razie jeżeli `childOfu` jest czerwony, należy pokolorować go na czarno (wiersze 195. i 196.).
- W przeciwnym razie należy wywołać metodę `fixDoubleBlack`, aby rozwiązać problem podwójnie czarnego węzła w `childOfu` (wiersz 199.).

Metoda `fixDoubleBlack` najpierw pobiera węzły `y`, `y1` i `y2` (wiersze 207. – 210.). Węzeł `y` jest bratem pierwszego podwójnie czarnego węzła, a `y1` i `y2` są lewym i prawym dzieckiem węzła `y`. Rozważ trzy sytuacje:

- Jeśli `y` jest czarny, a `y1` lub `y2` jest czerwony, rozwiąż problem podwójnie czarnego węzła ze scenariusza 1. (wiersze 213. – 255.).
- W przeciwnym razie jeśli `y` jest czarny, rozwiąż problem podwójnie czarnego węzła ze scenariusza 2., zmieniając kolor węzłów. Jeżeli rodzic jest czarny i nie jest korzeniem, przenieś problem podwójnie czarnego węzła do rodzica i rekurencyjnie wywołaj metodę `fixDoubleBlack` (wiersze 264. – 268.).
- W przeciwnym razie `y` jest czerwony. Wtedy należy dostosować węzły i ustawić rodzica jako dziecko węzła `y` (wiersze 272. – 281.). Wywołaj `fixDoubleBlack` dla tak dostosowanych węzłów (wiersz 286.), aby rozwiązać problem podwójnie czarnego węzła.

Metoda `preorder(TreeNode<E> root)` jest przesłonięta, aby wyświetlać kolory węzłów (wiersze 319. – 325.).

36.7. Testowanie klasy RBTree

W tym podrozdziale opisany jest program testowy używający klasy `RBTree`.

Na listingu 36.4 pokazany jest program testowy. Tworzy on drzewo `RBTree` zainicjowane tablicą liczb całkowitych 34, 3 i 50 (wiersze 4. i 5.), wstawia elementy w wierszach 10. – 22. i usuwa elementy w wierszach 25. – 46.

LISTING 36.4. TestRBTree.java

```
1 public class TestRBTree {
2   public static void main(String[] args) {
3     // Tworzenie drzewa czerwono-czarnego
4     RBTree<Integer> tree =
5       new RBTree<Integer>(new Integer[]{34, 3, 50});
6     printTree(tree);
```

```
 7
 8     tree.insert(20);
 9     printTree(tree);
10
11     tree.insert(15);
12     printTree(tree);
13
14     tree.insert(16);
15     printTree(tree);
16
17     tree.insert(25);
18     printTree(tree);
19
20     tree.insert(27);
21     printTree(tree);
22
23     tree.delete(50);
24     printTree(tree);
25
26     tree.delete(20);
27     printTree(tree);
28
29     tree.delete(15);
30     printTree(tree);
31
32     tree.delete(3);
33     printTree(tree);
34
35     tree.delete(25);
36     printTree(tree);
37
38     tree.delete(16);
39     printTree(tree);
40
41     tree.delete(34);
42     printTree(tree);
43
44     tree.delete(27);
45     printTree(tree);
46   }
47
48   public static <E extends Comparable<E>>
49     void printTree(BST <E> tree) {
50     // Przechodzenie drzewa
51     System.out.print("\nInorder (posortowane): ");
52     tree.inorder();
53     System.out.print("\nPostorder: ");
54     tree.postorder();
55     System.out.print("\nPreorder: ");
56     tree.preorder();
57     System.out.print("\nLiczba węzłów: " + tree.getSize());
58     System.out.println();
59   }
60 }
```

```
Inorder (posortowane): 3 34 50
Postorder: 3 50 34
Preorder: 34 (czarny) 3 (czerwony) 50 (czerwony)
Liczba węzłów: 3

Inorder (posortowane): 3 20 34 50
Postorder: 20 3 50 34
Preorder: 34 (czarny) 3 (czarny) 20 (czerwony) 50 (czarny)
Liczba węzłów: 4

Inorder (posortowane): 3 15 20 34 50
Postorder: 3 20 15 50 34
Preorder: 34 (czarny) 15 (czarny) 3 (czerwony) 20 (czerwony) 50 (czarny)
Liczba węzłów: 5

Inorder (posortowane): 3 15 16 20 34 50
Postorder: 3 16 20 15 50 34
Preorder: 34 (czarny) 15 (czerwony) 3 (czarny) 20 (czarny) 16 (czerwony) 50 (czarny)
Liczba węzłów: 6

Inorder (posortowane): 3 15 16 20 25 34 50
Postorder: 3 16 25 20 15 50 34
Preorder: 34 (czarny) 15 (czerwony) 3 (czarny) 20 (czarny) 16 (czerwony) 25
(czerwony)
  50 (czarny)
Liczba węzłów: 7

Inorder (posortowane): 3 15 16 20 25 27 34 50
Postorder: 3 16 15 27 25 50 34 20
Preorder: 20 (czarny) 15 (czerwony) 3 (czarny) 16 (czarny) 34 (czerwony) 25 (czarny)
  27 (czerwony) 50 (czarny)
Liczba węzłów: 8

Inorder (posortowane): 3 15 16 20 25 27 34
Postorder: 3 16 15 25 34 27 20
Preorder: 20 (czarny) 15 (czerwony) 3 (czarny) 16 (czarny) 27 (czerwony)
  25 (czarny) 34 (czarny)
Liczba węzłów: 7

Inorder (posortowane): 3 15 16 25 27 34
Postorder: 3 15 25 34 27 16
Preorder: 16 (czarny) 15 (czarny) 3 (czerwony) 27 (czerwony) 25 (czarny) 34 (czarny)
Liczba węzłów: 6

Inorder (posortowane): 3 16 25 27 34
Postorder: 3 25 34 27 16
Preorder: 16 (czarny) 3 (czarny) 27 (czerwony) 25 (czarny) 34 (czarny)
Liczba węzłów: 5

Inorder (posortowane): 16 25 27 34
Postorder: 25 16 34 27
Preorder: 27 (czarny) 16 (czarny) 25 (czerwony) 34 (czarny)
Liczba węzłów: 4
```

```
Inorder (posortowane): 16 27 34
Postorder: 16 34 27
Preorder: 27 (czarny) 16 (czarny) 34 (czarny)
Liczba węzłów: 3

Inorder (posortowane): 27 34
Postorder: 34 27
Preorder: 27 (czarny) 34 (czerwony)
Liczba węzłów: 2

Inorder (posortowane): 27
Postorder: 27
Preorder: 27 (czarny)
Liczba węzłów: 1

Inorder (posortowane):
Postorder:
Preorder:
Liczba węzłów: 0
```

Rysunek 36.14 przedstawia zmiany w drzewie w trakcie dodawania elementów. Na rysunku 36.28 pokazane są zmiany w trakcie usuwania elementów.

36.8. Wydajność klasy RBTree

W drzewach czerwono-czarnych wyszukiwanie, wstawianie i usuwanie elementów ma wydajność O(log*n*).

Czas wyszukiwania, wstawiania i usuwania w drzewach czerwono-czarnych zależy od wysokości drzewa. Drzewo czerwono-czarne odpowiada drzewu 2-3-4. Gdy przekształcisz węzeł z drzewa 2-3-4 w węzły drzewa czerwono-czarnego, otrzymasz jeden czarny węzeł i zero, jeden lub dwa węzły czerwone będące dziećmi węzła czarnego. Liczba dzieci zależy od tego, czy pierwotny węzeł jest 2-, 3- czy 4-węzłem. Tak więc wysokość drzewa czerwono-czarnego jest maksymalnie dwa razy większa niż analogicznego drzewa 2-3-4. Ponieważ wysokość drzewa 2-3-4 wynosi log*n*, wysokość drzewa czerwono-czarnego to 2log*n*.

Drzewo czerwono-czarne ma tę samą złożoność co drzewo AVL, co ilustruje tabela 36.1. Zwykle drzewo czerwono-czarne jest wydajniejsze od drzewa AVL, ponieważ wymaga tylko jednorazowej zmiany struktury węzłów w trakcie wstawiania i usuwania elementów.

Tabela 36.1. Złożoność czasowa metod z klas RBTree, AVLTree i Tree24

Metoda	*Drzewo czerwono-czarne*	*Drzewo AVL*	*Drzewo 2-3-4*
`search (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`insert (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`delete (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`getSize()`	*O*(1)	*O*(1)	*O*(1)
`isEmpty()`	*O*(1)	*O*(1)	*O*(1)

Drzewo czerwono-czarne ma też tę samą złożoność co drzewo 2-3-4 (zobacz tabelę 36.1). Zwykle jest jednak wydajniejsze od drzewa 2-3-4. Wynika to z dwóch powodów:

1. Drzewo czerwono-czarne wymaga tylko jednorazowej zmiany struktury węzłów w trakcie wstawiania i usuwania elementów. Jednak drzewo 2-3-4 może wymagać wielu podziałów w czasie wstawiania i wielu operacji złączania w trakcie usuwania elementu.
2. Drzewo czerwono-czarne jest binarnym drzewem poszukiwań. Drzewo binarne można zaimplementować w sposób wymagający mniej pamięci niż drzewo 2-3-4, ponieważ węzeł w drzewie 2-3-4 ma najwyżej trzy elementy i czworo dzieci. Dlatego w drzewach 2-3-4 2- i 3-węzły powodują marnowanie pamięci.

Listing 36.5 pozwala w empiryczny sposób sprawdzić wydajność drzew AVL, drzew 2-3-4 i drzew czerwono-czarnych.

LISTING 36.5. TreePerformanceTest.java

```
1 public class TreePerformanceTest {
2   public static void main(String[] args) {
3     final int TEST_SIZE = 500000; // Wielkość drzewa używana w testach
4
5     // Tworzenie drzewa AVL
6     Tree<Integer> tree1 = new AVLTree<Integer>();
7     System.out.println("Czas dla drzewa AVL: " +
8       getTime(tree1, TEST_SIZE) + " ms");
9
10    // Tworzenie drzewa 2-3-4
11    Tree<Integer> tree2 = new Tree24<Integer>();
12    System.out.println("Czas dla drzewa 2-3-4: "
13      + getTime(tree2, TEST_SIZE) + " ms");
14
15    // Tworzenie drzewa czerwono-czarnego
16    Tree<Integer> tree3 = new RBTree<Integer>();
17    System.out.println("Czas dla drzewa czerwono-czarnego: "
18      + getTime(tree3, TEST_SIZE) + " ms");
19  }
20
21  public static long getTime(Tree<Integer> tree, int testSize) {
22    long startTime = System.currentTimeMillis(); // Czas początkowy
23
24    // Tworzenie listy do przechowywania różnych liczb całkowitych
25    java.util.List<Integer> list = new java.util.ArrayList<Integer>();
26    for (int i = 0; i < testSize; i++)
27      list.add(i);
28
29    java.util.Collections.shuffle(list); // Tasowanie listy
30
31    // Wstawianie elementów z listy do drzewa
32    for (int i = 0; i < testSize; i++)
33      tree.insert(list.get(i));
34
35    java.util.Collections.shuffle(list); // Tasowanie listy
36
37    // Usuwanie elementów z listy z drzewa
38    for (int i = 0; i < testSize; i++)
39      tree.delete(list.get(i));
```

```
40
41      // Zwracanie czasu wykonywania operacji
42      return System.currentTimeMillis() - startTime;
43    }
44 }
```

```
Czas dla drzewa AVL: 7609 ms
Czas dla drzewa 2–3-4: 8594 ms
Czas dla drzewa czerwono-czarnego: 5515 ms
```

Metoda `getTestTime` tworzy listę różnych liczb całkowitych od 0 do `testSize – 1` (wiersze 25. – 27.), tasuje tę listę (wiersz 29.), dodaje elementy z listy do drzewa (wiersze 32. i 33.), ponownie tasuje listę (wiersz 35.), usuwa elementy z drzewa (wiersze 38. i 39.) i zwraca czas wykonywania operacji (wiersz 42.).

Program tworzy drzewo AVL (wiersz 6.), drzewo 2-3-4 (wiersz 11.) i drzewo czerwono-czarne (wiersz 16.). Następnie pobiera czas dodawania i usuwania 500 000 elementów w każdym z tych drzew.

Wyniki pokazują, że drzewo czerwono-czarne zapewnia najwyższą wydajność, a następne jest drzewo AVL.

Uwaga

Klasa `java.util.TreeSet` z API Javy jest zaimplementowana za pomocą drzew czerwono-czarnych. Każdy wpis ze zbioru jest przechowywany w drzewie. Ponieważ metody `search`, `insert` i `delete` w drzewach czerwono-czarnych działają w czasie $O(\log n)$, metody `get`, `add`, `remove` i `contains` z klasy `java.util.TreeSet` też mają złożoność $O(\log n)$.

Uwaga

Klasa `java.util.TreeMap` z API Javy też jest zaimplementowana za pomocą drzew czerwono-czarnych. Każdy wpis z odwzorowania jest przechowywany w drzewie. Uporządkowanie elementów zależy od ich kluczy. Ponieważ metody `search`, `insert` i `delete` w drzewach czerwono-czarnych działają w czasie $O(\log n)$, metody `get`, `put`, `remove` i `containsKey` z klasy `java.util.TreeMap` też mają złożoność $O(\log n)$.

NAJWAŻNIEJSZE POJĘCIA

długość ścieżki czarnych węzłów
problem podwójnie czarnego węzła
problem dwóch kolejnych czerwonych węzłów
węzeł zewnętrzny
drzewo czerwono-czarne

PODSUMOWANIE ROZDZIAŁU

1. Drzewo czerwono-czarne jest binarnym drzewem poszukiwań opartym na *drzewie 2-3-4*. Drzewo czerwono-czarne odpowiada drzewu 2-3-4. Możesz przekształcić drzewo czerwono-czarne w drzewo 2-3-4 i na odwrót.
2. W drzewie czerwono-czarnym każdy węzeł ma kolor czarny lub czerwony. Korzeń zawsze jest czarny. Dwa kolejne węzły nie mogą być czerwone. Wszystkie węzły zewnętrzne mają tę samą długość ścieżki czarnych węzłów.
3. Ponieważ drzewo czerwono-czarne jest binarnym drzewem poszukiwań, klasa `RBTree` rozszerza klasę `BST`.
4. Wyszukiwanie elementu w drzewie czerwono-czarnym przebiega tak samo jak w binarnym drzewie poszukiwań (ponieważ drzewo czerwono-czarne jest odmianą binarnego drzewa poszukiwań).

5. Nowy element zawsze jest wstawiany w liściu. Jeśli nowym węzłem jest korzeń, należy pokolorować go na czarno. W przeciwnym razie używany jest kolor czerwony. Jeżeli rodzic nowego węzła jest czerwony, trzeba rozwiązać problem *dwóch kolejnych czerwonych węzłów*, zmieniając kolor węzłów i/lub strukturę drzewa.
6. Gdy usuwany jest węzeł wewnętrzny, trzeba znaleźć pierwszy od prawej węzeł w lewym poddrzewie usuwanego węzła. Zastąp element z usuwanego węzła elementem ze znalezionego węzła pierwszego od prawej, po czym usuń ten ostatni węzeł.
7. Jeśli usuwany węzeł zewnętrzny jest czerwony, wystarczy powiązać jego rodzica z dzieckiem usuwanego zewnętrznego węzła.
8. Jeżeli usuwany węzeł zewnętrzny jest czarny, należy rozważyć kilka przypadków, aby się upewnić, że zachowana zostanie odpowiednia długość ścieżek czarnych węzłów w węzłach zewnętrznych.
9. Wysokość drzewa czerwono-czarnego wynosi $O(\log n)$, dlatego złożoność czasowa metod `search`, `insert` i `delete` to $O(\log n)$.

Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

Ćwiczenia programistyczne

***36.1.** *Konwersja drzewa czerwono-czarnego na drzewo 2-3-4.* Napisz program, który przekształca drzewo czerwono-czarne w drzewo 2-3-4.

***36.2.** *Konwersja drzewa 2-3-4 na drzewo czerwono-czarne.* Napisz program, który przekształca drzewo 2-3-4 w drzewo czerwono-czarne.

*****36.3.** *Animacja działania metod drzew czerwono-czarnych.* Napisz program z GUI, który wyświetla animację działania metod `insert`, `delete` i `search` w drzewach czerwono-czarnych (rysunek 36.6).

****36.4.** *Wskaźnik do rodzica w klasie `RBTree`.* Przyjmij, że klasa `TreeNode` zdefiniowana w klasie `BST` zawiera wskaźnik do rodzica węzła (zobacz ćwiczenie 26.17). Zaimplementuj klasę `RBTree` zgodną z tą zmianą. Napisz program testowy, który dodaje do drzewa liczby 1, 2, …, 100 i wyświetla ścieżki do wszystkich liści.

ROZDZIAŁ

37

Testy z użyciem JUnit

Cele

- Wyjaśnienie, czym jest JUnit i jak działa (podrozdział 37.2).
- Utworzenie i uruchomienie klasy testów JUnit w wierszu poleceń (podrozdział 37.2).
- Utworzenie i uruchomienie klasy testów JUnit w środowisku NetBeans (podrozdział 37.3).
- Utworzenie i uruchomienie klasy testów JUnit w środowisku Eclipse (podrozdział 37.4).

37.1. Wprowadzenie

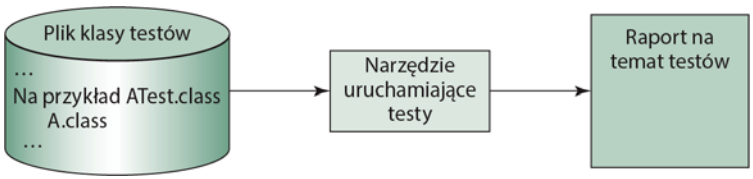
JUnit jest narzędziem do testowania programów w Javie[1].

Na samym początku książki, w podrozdziale 2.16, opisany został proces tworzenia oprogramowania obejmujący tworzenie specyfikacji wymagań oraz analizowanie, projektowanie, implementowanie, testowanie, wdrażanie i konserwację kodu. Testy są ważną częścią tego procesu. W tym rozdziale dowiesz się, jak testować klasy Javy za pomocą JUnit.

37.2. Podstawy JUnit

Aby przetestować klasę, musisz napisać klasę testów i uruchomić ją w JUnit, by wygenerować raport na temat badanej klasy.

JUnit jest główną platformą do testowania programów w Javie. Ma postać niezależnej otwartej biblioteki dostępnej jako plik *jar*. Ten plik zawiera *narzędzie uruchamiające testy* (ang. *test runner*), które służy do wykonywania programów testowych. Przyjmij, że używasz klasy `A`. Aby ją przetestować, musisz napisać klasę testów `ATest`. W *klasie testów* należy umieścić metody do testowania klasy `A`. Narzędzie uruchamiające testy wykona klasę `ATest`, aby wygenerować raport (rysunek 37.1).



Rysunek 37.1. Narzędzie uruchamiające testy wykonuje klasę testów w celu wygenerowania raportu

W omawianym przykładzie zobaczysz, jak działa JUnit. Aby przygotować przykład, najpierw pobierz JUnit ze strony *http://sourceforge.net/projects/junit/files/*. Obecnie najnowszą wersją jest *junit-4.10.jar*. Umieść ten plik w katalogu *c:\kod\lib* i dodaj do zmiennej środowiskowej `classpath`:

```
set classpath=.;%classpath%;c:\kod\lib\junit-4.10.jar
```

Aby sprawdzić, czy zmienna środowiskowa jest poprawnie skonfigurowana, otwórz nowe okno wiersza poleceń i wpisz następującą instrukcję:

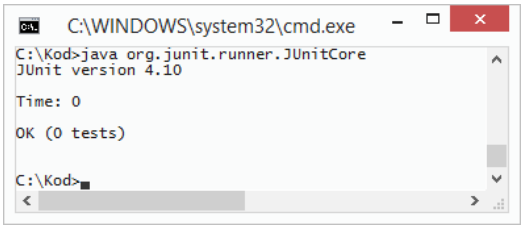
```
java org.junit.runner.JUnitCore
```

Powinieneś zobaczyć komunikat widoczny na rysunku 37.2.

Aby użyć JUnit, utwórz klasę testów. Jeśli testowana klasa to `A`, klasę testów zwyczajowo należy nazwać `ATest`. Oto prosty szablon klasy testów:

```
1 package mytest;
2
3 import org.junit.*;
4 import static org.junit.Assert.*;
5
```

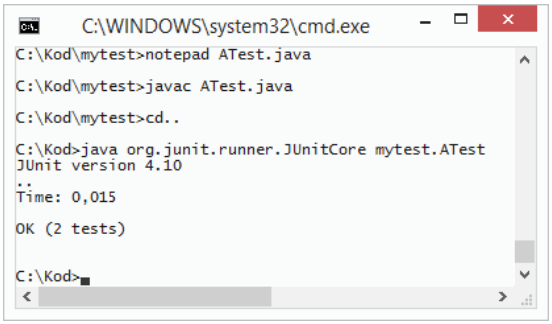
[1] W witrynie poświęconej oryginalnemu wydaniu książki jest to rozdział 44. — *przyp. tłum.*



RYSUNEK 37.2. Narzędzie uruchamiające testy wyświetla wersję platformy JUnit

```
 6 public class ATest {
 7   @Test
 8   public void m1() {
 9     // Pisanie metody testowej
10   }
11
12   @Test
13   public void m2() {
14     // Następna metoda testowa
15   }
16
17   @Before
18   public void setUp() throws Exception {
19     // Tu można skonfigurować obiekty używane w różnych metodach testowych
20   }
21 }
```

Tę klasę umieść w katalogu *mytest*. Przyjmij, że znajduje się ona w katalogu *c:\kod\mytest*. Należy skompilować klasę w katalogu *mytest* i uruchomić z poziomu katalogu *c:\kod* w pokazany sposób:



Zauważ, że polecenie uruchamiające test w konsoli wygląda tak:

```
java org.junit.runner.JUnitCore mytest.ATest
```

Gdy uruchomisz to polecenie, klasa `JUnitCore` będzie kontrolować wykonywanie klasy `ATest`. Najpierw wykonana zostanie metoda `setUp()`, aby skonfigurować wspólne obiekty używane w testach. Następnie uruchomione zostaną metody testowe `m1` i `m2` (w tej kolejności). Jeśli to potrzebne, możesz zdefiniować wiele metod testowych.

Aby zaimplementować metodę testową, możesz użyć następujących metod:

`assertTrue(wyrażenieLogiczne)`
Informuje o powodzeniu, jeśli `wyrażenieLogiczne` ma wartość `true`.

`assertEquals(Object, Object)`
Informuje o powodzeniu, jeśli według metody `equals` oba obiekty są sobie równe.

`assertNull(Object)`
Informuje o powodzeniu, jeśli przekazana referencja do obiektu to `null`.

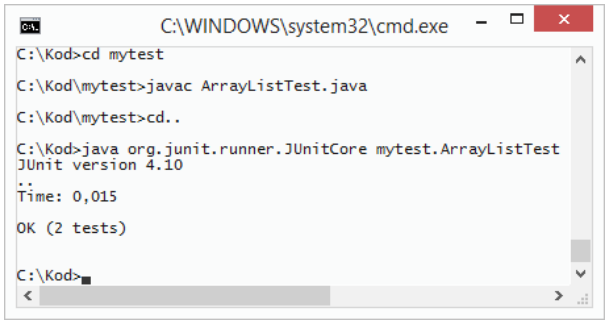
`fail(String)`
Powoduje zakończenie testu niepowodzeniem i wyświetla podany łańcuch znaków.

Na listingu 37.1 pokazana jest przykładowa klasa testów sprawdzająca klasę `java.util.ArrayList`.

LISTING 37.1. ArrayListTest.java

```
 1 package mytest;
 2
 3 import org.junit.*;
 4 import static org.junit.Assert.*;
 5 import java.util.*;
 6
 7 public class ArrayListTest {
 8   private ArrayList<String> list = new ArrayList<String>();
 9
10   @Before
11   public void setUp() throws Exception {
12   }
13
14   @Test
15   public void testInsertion() {
16     list.add("Pekin");
17     assertEquals("Pekin", list.get(0));
18     list.add("Szanghaj");
19     list.add("Hongkong");
20     assertEquals("Hongkong", list.get(list.size() - 1));
21   }
22
23   @Test
24   public void testDeletion() {
25     list.clear();
26     assertTrue(list.isEmpty());
27
28     list.add("A");
29     list.add("B");
30     list.add("C");
31     list.remove("B");
32     assertEquals(2, list.size());
33   }
34 }
```

Uruchamianie testów jest pokazane na rysunku 37.3. Zauważ, że najpierw trzeba skompilować plik *ArrayListTest.java*. Klasa `ArrayListTest` jest umieszczana w pakiecie `mytest`. Należy więc umieścić plik *ArrayListTest.java* w katalogu *mytest*.



Rysunek 37.3. Uruchomienie klasy ArrayListTest skutkuje wyświetleniem raportu na temat testów

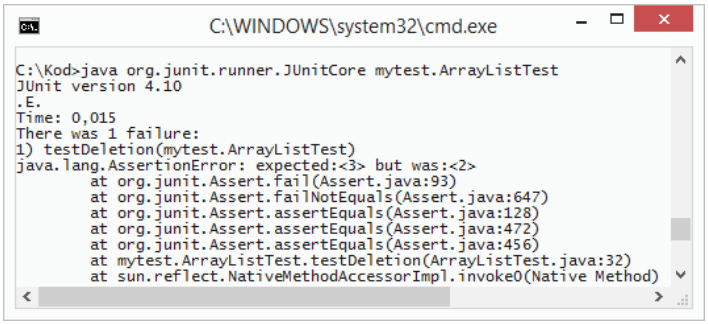
Te testy JUnit nie informują o żadnych błędach. Zmień teraz kod w wierszu 32.:

```
assertEquals(2, list.size());
```

na

```
assertEquals(3, list.size());
```

Ponownie uruchom klasę `ArrayListTest`. Pojawią się informacje o błędzie widoczne na rysunku 37.4.



Rysunek 37.4. W raporcie znajdują się informacje na temat błędu

Możesz zdefiniować dowolną liczbę metod testowych. W tym przykładzie zdefiniowane są dwie metody testowe: `testInsertion` i `testDeletion`. JUnit najpierw wykonuje metodę `testInsertion`, a następnie `testDeletion`.

Uwaga

W tym przykładzie klasa testów musi znaleźć się w pakiecie z nazwą taką jak `mytest`. JUnit nie zadziała, jeśli klasa testów będzie umieszczona w pakiecie domyślnym.

Na listingu 37.2 przedstawiona jest klasa testów sprawdzająca klasę `Loan` z listingu 10.2. Dla wygody plik *Loan.java* umieść w tym samym katalogu, w którym znajdzie się plik *LoanTest.java*. Klasa `Loan` pokazana jest na listingu 37.3.

LISTING 37.2. LoanTest.java

```
 1 package mytest;
 2
 3 import org.junit.*;
 4 import static org.junit.Assert.*;
 5
 6 public class LoanTest {
 7   @Before
 8   public void setUp() throws Exception {
 9   }
10
11   @Test
12   public void testPaymentMethods() {
13     double annualInterestRate = 2.5;
14     int numberOfYears = 5;
15     double loanAmount = 1000;
16     Loan loan = new Loan(annualInterestRate, numberOfYears,
17       loanAmount);
18
19     assertTrue(loan.getMonthlyPayment() ==
20       getMonthlyPayment(annualInterestRate, numberOfYears,
21       loanAmount));
22     assertTrue(loan.getTotalPayment() ==
23       getTotalPayment(annualInterestRate, numberOfYears,
24       loanAmount));
25   }
26
27   /** Obliczanie miesięcznych rat */
28   private double getMonthlyPayment(double annualInterestRate,
29       int numberOfYears, double loanAmount) {
30     double monthlyInterestRate = annualInterestRate / 1200;
31     double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
32       (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
33     return monthlyPayment;
34   }
35
36   /** Obliczanie łącznej kwoty do spłaty */
37   public double getTotalPayment(double annualInterestRate,
38       int numberOfYears, double loanAmount) {
39     return getMonthlyPayment(annualInterestRate, numberOfYears,
40       loanAmount) * numberOfYears * 12;
41   }
42 }
```

LISTING 37.3. Loan.java

```
 1 package mytest;
 2
 3 public class Loan {
 4   private double annualInterestRate;
 5   private int numberOfYears;
 6   private double loanAmount;
 7   private java.util.Date loanDate;
 8
 9   /** Konstruktor domyślny */
```

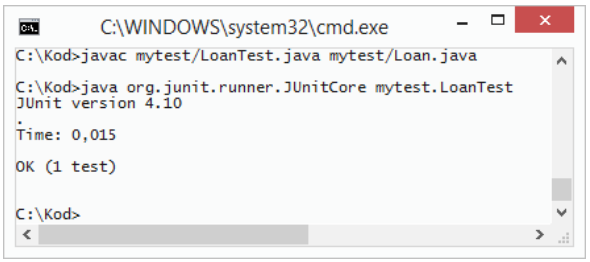
```
10  public Loan() {
11    this(2.5, 1, 1000);
12  }
13
14  /** Tworzy obiekt typu Loan z określonym rocznym oprocentowaniem,
15      liczbą lat spłaty i kwotą kredytu
16    */
17  public Loan(double annualInterestRate, int numberOfYears,
18      double loanAmount) {
19    this.annualInterestRate = annualInterestRate;
20    this.numberOfYears = numberOfYears;
21    this.loanAmount = loanAmount;
22    loanDate = new java.util.Date();
23  }
24
25  /** Zwraca annualInterestRate (roczną stopę oprocentowania) */
26  public double getAnnualInterestRate() {
27    return annualInterestRate;
28  }
29
30  /** Ustawia nową wartość annualInterestRate (roczną stopę oprocentowania) */
31  public void setAnnualInterestRate(double annualInterestRate) {
32    this.annualInterestRate = annualInterestRate;
33  }
34
35  /** Zwraca numberOfYears (liczbę lat spłaty) */
36  public int getNumberOfYears() {
37    return numberOfYears;
38  }
39
40  /** Ustawia nową wartość numberOfYears (liczbę lat spłaty) */
41  public void setNumberOfYears(int numberOfYears) {
42    this.numberOfYears = numberOfYears;
43  }
44
45  /** Zwraca loanAmount (kwotę kredytu) */
46  public double getLoanAmount() {
47    return loanAmount;
48  }
49
50  /** Ustawia nową wartość newloanAmount (kwotę kredytu) */
51  public void setLoanAmount(double loanAmount) {
52    this.loanAmount = loanAmount;
53  }
54
55  /** Oblicza miesięczną ratę */
56  public double getMonthlyPayment() {
57    double monthlyInterestRate = annualInterestRate / 1200;
58    double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
59      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
60    return monthlyPayment;
61  }
62
63  /** Oblicza łączną kwotę do spłaty */
64  public double getTotalPayment() {
```

```
65      double totalPayment = getMonthlyPayment() * numberOfYears * 12;
66      return totalPayment;
67    }
68
69    /** Zwraca datę uzyskania kredytu */
70    public java.util.Date getLoanDate() {
71      return loanDate;
72    }
73 }
```

Metoda `testPaymentMethods()` w klasie `LoanTest` tworzy obiekt typu `Loan` (wiersze 16. i 17.) i sprawdza, czy wywołanie `loan.getMonthlyPayment()` zwraca tę samą wartość co metoda `getMonthlyPayment(annualInterestRate, numberOfYears, loanAmount)`. Ta ostatnia metoda jest zdefiniowana w klasie `LoanTest` (wiersze 28. – 34.).

Metoda `testPaymentMethods()` sprawdza także, czy wywołanie `loan.getTotalPayment()` zwraca tę samą wartość co `getTotalPayment(annualInterestRate, numberOfYears, loanAmount)`. Ta ostatnia metoda jest zdefiniowana w klasie `LoanTest` (wiersze 37. – 41.).

Przykładowy przebieg tego programu ilustruje rysunek 37.5.



RYSUNEK 37.5. Narzędzie uruchamiające testy z JUnit wykonuje klasę LoanTest i informuje o braku błędów

37.2.1. Czym jest JUnit?
37.2.2. Czym jest narzędzie uruchamiające testy w JUnit?
37.2.3. Czym jest klasa testów? Jak ją utworzyć?
37.2.4. Jak używać metody `assertTrue`?
37.2.5. Jak używać metody `assertEquals`?

37.3. Używanie JUnit w NetBeans

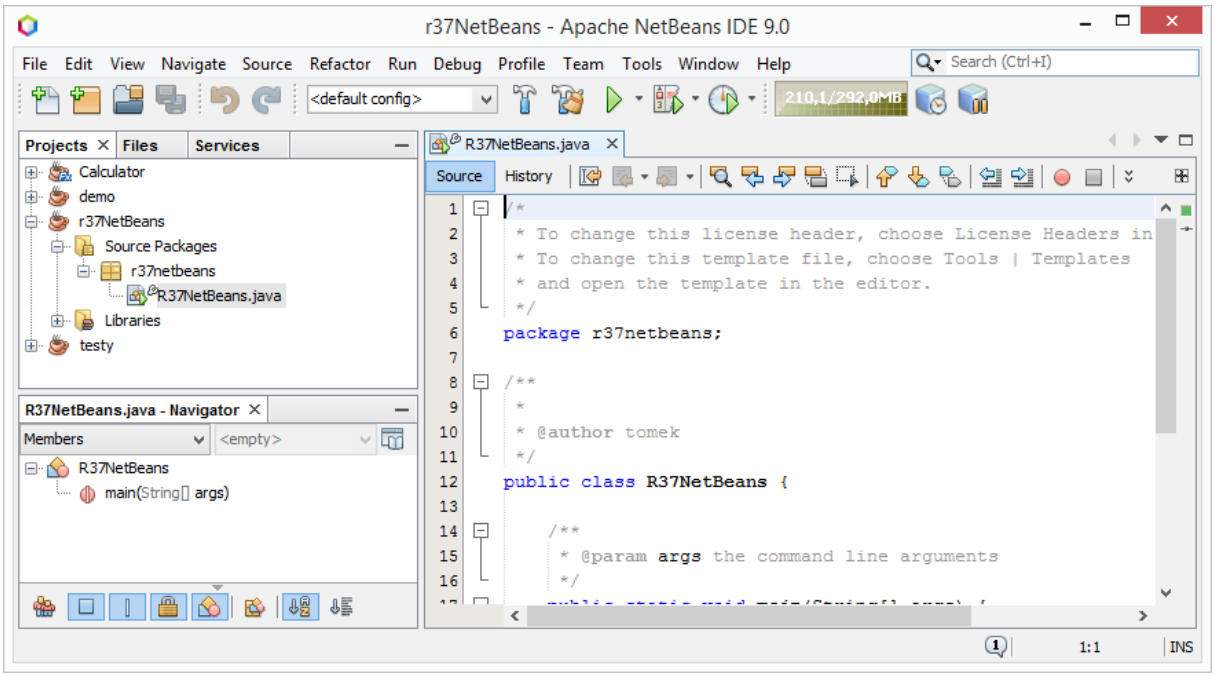
Platforma JUnit jest zintegrowana ze środowiskiem NetBeans. W NetBeans można automatycznie wygenerować program testowy i zautomatyzować proces testowania.

Środowiska IDE takie jak NetBeans i Eclipse pozwalają znacznie uprościć proces tworzenia i uruchamiania klas testowych. W tym podrozdziale znajdziesz wprowadzenie do używania JUnit w NetBeans. Następny podrozdział opisuje, jak używać JUnit w Eclipse.

Jeśli nie znasz NetBeans, zajrzyj do suplementu II.B w witrynie powiązanej z oryginalnym wydaniem książki. Tu zakładam, że zainstalowałeś NetBeans 8 lub nowszą wersję. Utwórz projekt `r37NetBeans`:

Krok 1. Wybierz opcję *File/New Project*, aby wyświetlić okno dialogowe *New Project*.
Krok 2. Wybierz opcję *Java* w polu *Categories* i *Java Application* w polu *Projects*. Kliknij przycisk *Next*, aby wyświetlić okno dialogowe *New Java Application*.

Krok 3. Wpisz *r37NetBeans* jako nazwę projektu i *c:\kod* jako jego lokalizację. Kliknij przycisk *Finish*, aby utworzyć projekt (rysunek 37.6).



RYSUNEK 37.6. Utworzony nowy projekt o nazwie r37NetBeans

Aby zobaczyć, jak tworzyć klasy testów, najpierw należy utworzyć klasę do sprawdzenia. Niech będzie to klasa `Loan` z listingu 10.2. Oto kroki tworzenia klasy `Loan` w katalogu *r37NetBeans*.

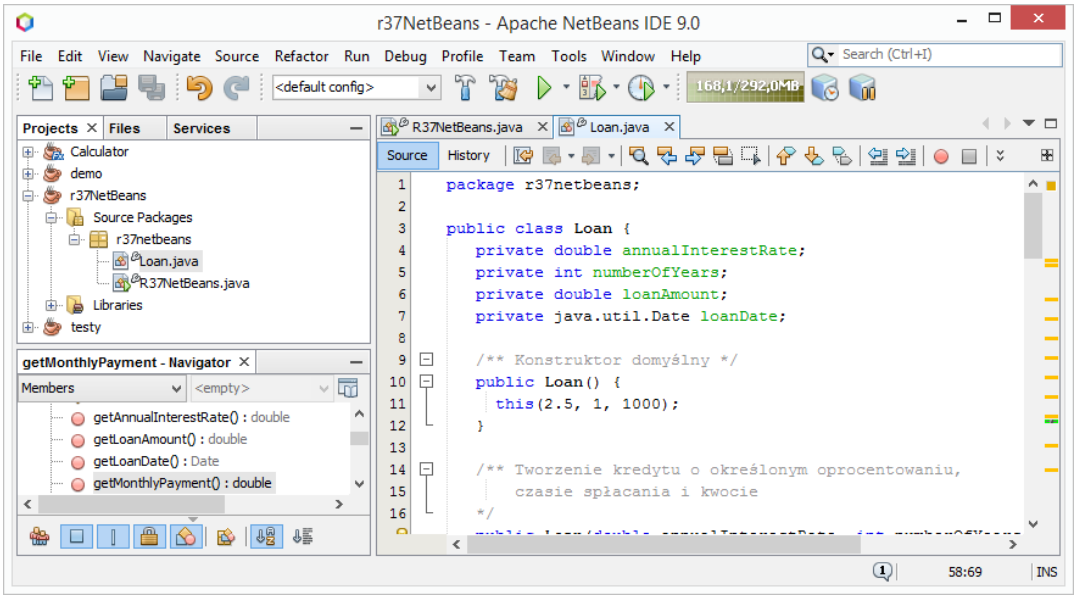
Krok 1. Kliknij prawym przyciskiem myszy projekt *r37NetBeans* i wybierz opcję *New/Java Class*, aby wyświetlić okno dialogowe *New Java Class*.
Krok 2. Wpisz *Loan* jako nazwę klasy i *r37NetBeans* jako nazwę pakietu. Kliknij przycisk *Finish*, aby utworzyć klasę.
Krok 3. Skopiuj kod z listingu 10.2 do klasy `Loan`. Jako pierwszy wiersz nowej klasy wpisz *package r37NetBeans* (rysunek 37.7).

Teraz możesz utworzyć klasę testów, aby przetestować klasę `Loan`:

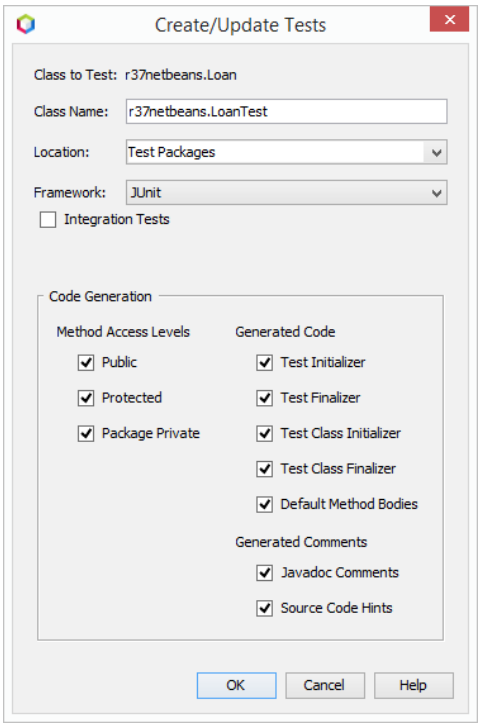
Krok 1. Kliknij prawym przyciskiem myszy plik *Loan.java* w projekcie, aby wyświetlić menu kontekstowe. Wybierz opcję *Tools/Create/Update Tests*, by wyświetlić okno dialogowe *Create Tests* (rysunek 37.8).

Krok 2. Kliknij przycisk *OK*. Zobaczysz okno dialogowe *Select JUnit Version* (rysunek 37.9). Wybierz opcję *JUnit 4.x* i kliknij przycisk *OK*, aby wygenerować klasę testów `LoanTest` (rysunek 37.10). Zauważ, że klasa *LoanTest.java* jest umieszczana w projekcie w węźle *Test Packages*.

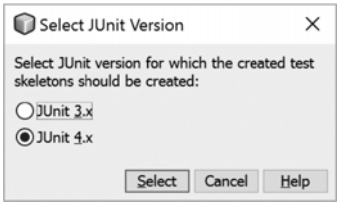
Teraz możesz zmodyfikować klasę `LoanTest`. W tym celu skopiuj kod z listingu 37.2. Uruchom kod z pliku *LoanTest.java*. Zobaczysz raport na temat testów widoczny na rysunku 37.11.



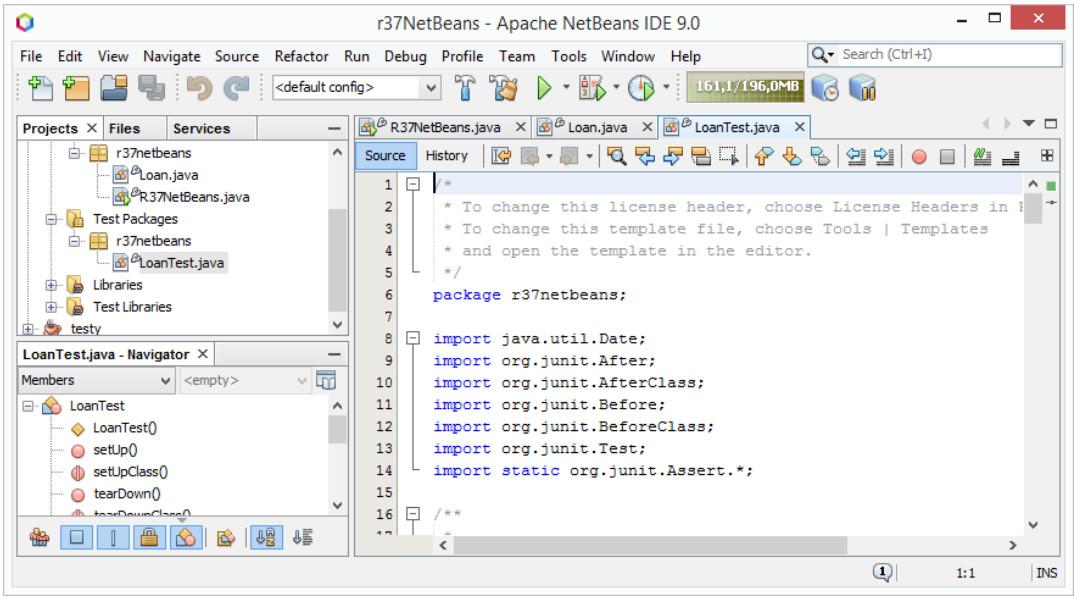
RYSUNEK 37.7. Utworzona klasa Loan



RYSUNEK 37.8. Okno dialogowe Create Tests pozwala utworzyć klasę testów



RYSUNEK 37.9. Wybierz platformę JUnit 4.x, aby utworzyć klasę testów



RYSUNEK 37.10. Klasa LoanTest jest generowana automatycznie

37.4. Używanie JUnit w Eclipse

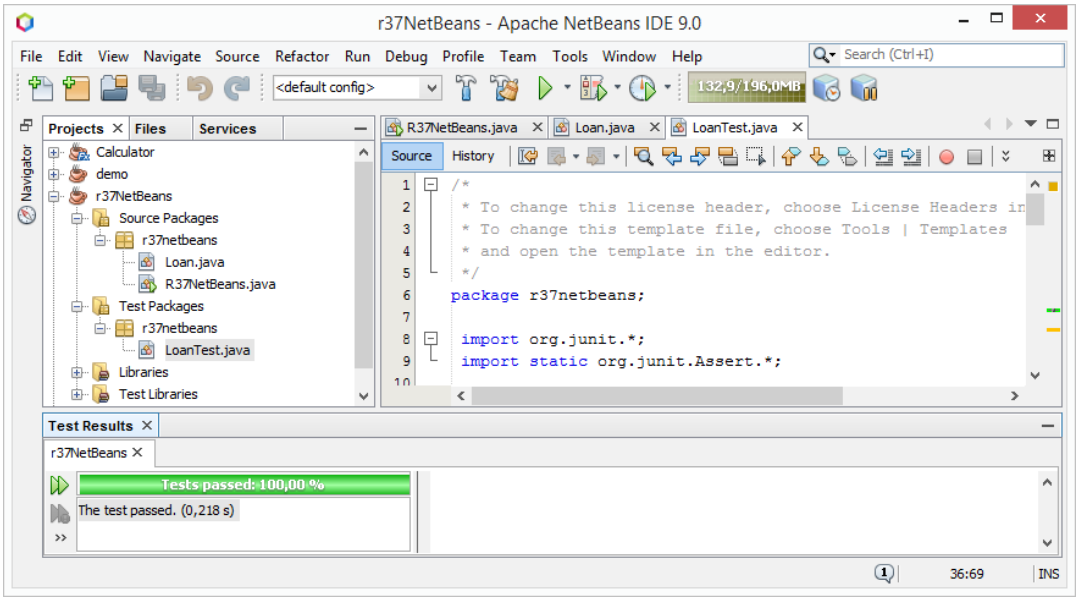
Platforma JUnit jest zintegrowana ze środowiskiem Eclipse. W Eclipse można automatycznie wygenerować program testowy i zautomatyzować proces testowania.

W tym podrozdziale opisane jest używanie JUnit w Eclipse. Jeśli nie znasz środowiska Eclipse, zapoznaj się z suplementem II.D w witrynie poświęconej oryginalnemu wydaniu książki. Zakładam, że zainstalowałeś Eclipse 4.5 lub nowszą wersję. Utwórz projekt *r37Eclipse*:

Krok 1. Wybierz opcję *Plik/Nowy/Projekt Java*, aby wyświetlić okno dialogowe *Nowy projekt Java* (rysunek 37.12).
Krok 2. Wpisz *r37Eclipse* jako nazwę projektu i kliknij przycisk *Zakończ*, aby utworzyć projekt.

Aby zobaczyć, jak tworzyć klasy testów, najpierw należy utworzyć testowaną klasę. Niech będzie to klasa `Loan` z listingu 10.2. Oto kroki tworzenia klasy `Loan` w katalogu *r37Eclipse*:

Krok 1. Kliknij prawym przyciskiem myszy węzeł projektu *r37Eclipse* i wybierz opcję *Nowy/Klasa*, aby wyświetlić okno dialogowe *Nowa klasa Java* (rysunek 37.13).



RYSUNEK 37.11. Po wykonaniu klasy LoanTest wyświetlany jest raport na temat testów

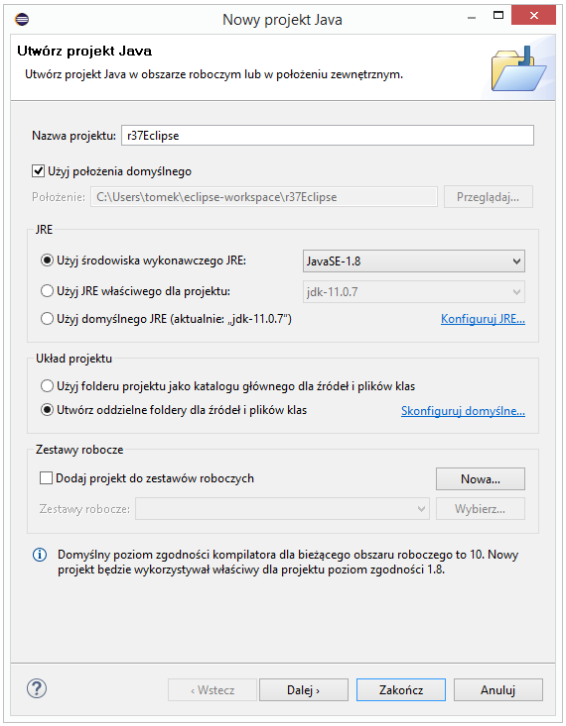
Krok 2. Wpisz *mytest* w polu *Pakiet* i kliknij przycisk *Zakończ*, aby utworzyć klasę.
Krok 3. Skopiuj kod z listingu 10.2 do klasy `Loan` i upewnij się, że pierwszy wiersz to `package mytest` (rysunek 37.14).

Teraz możesz utworzyć klasę testową dla klasy `Loan`:

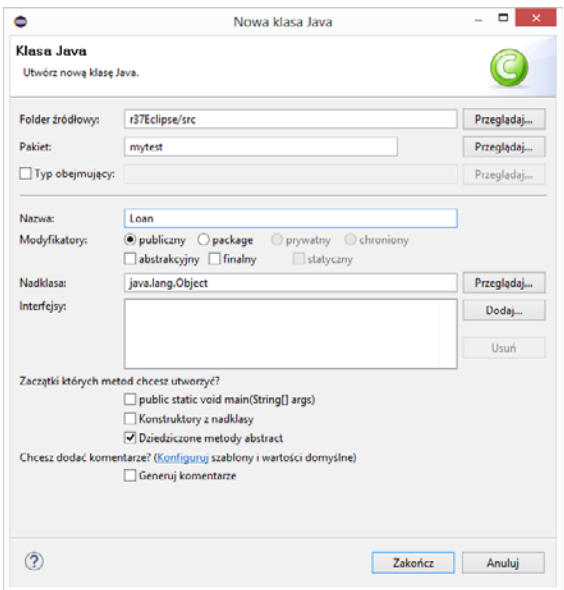
Krok 1. Kliknij prawym przyciskiem myszy plik *Loan.java* w projekcie, aby wyświetlić menu kontekstowe. Wybierz opcję *Nowy/Przypadek testowy JUnit*, aby wyświetlić okno dialogowe *Nowy przypadek testowy JUnit* (rysunek 37.15).

Krok 2. Kliknij przycisk *Zakończ*. Zobaczysz okno dialogowe z prośbą o dodanie lokalizacji platformy JUnit 4 do ścieżki budowania projektu. Kliknij przycisk *OK*, aby dodać tę lokalizację. Utworzona zostanie klasa testowa `LoanTest` (rysunek 37.16).

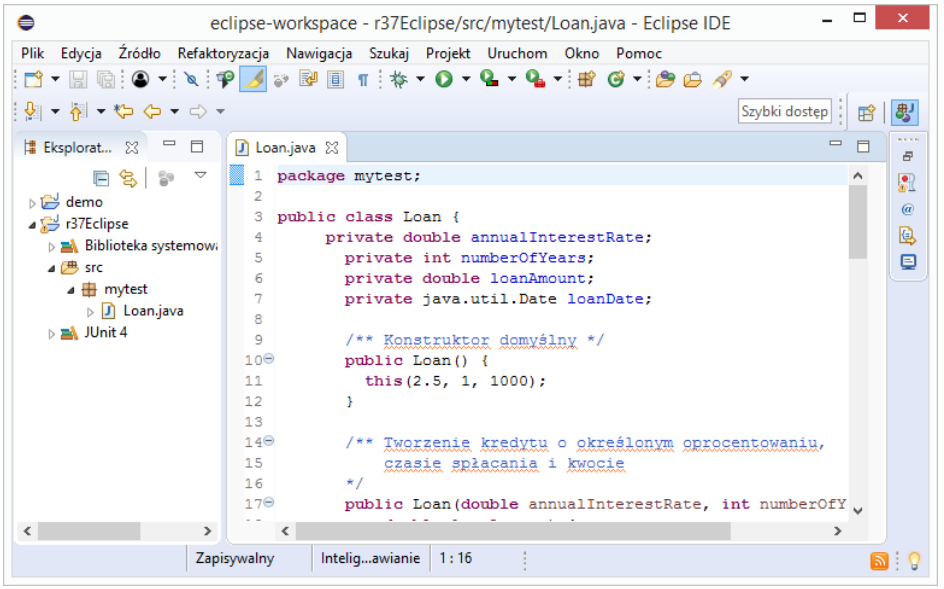
Teraz możesz zmodyfikować klasę `LoanTest`. W tym celu skopiuj kod z listingu 37.2. Uruchom plik *LoanTest.java*. Zobaczysz raport na temat testu pokazany na rysunku 37.17.



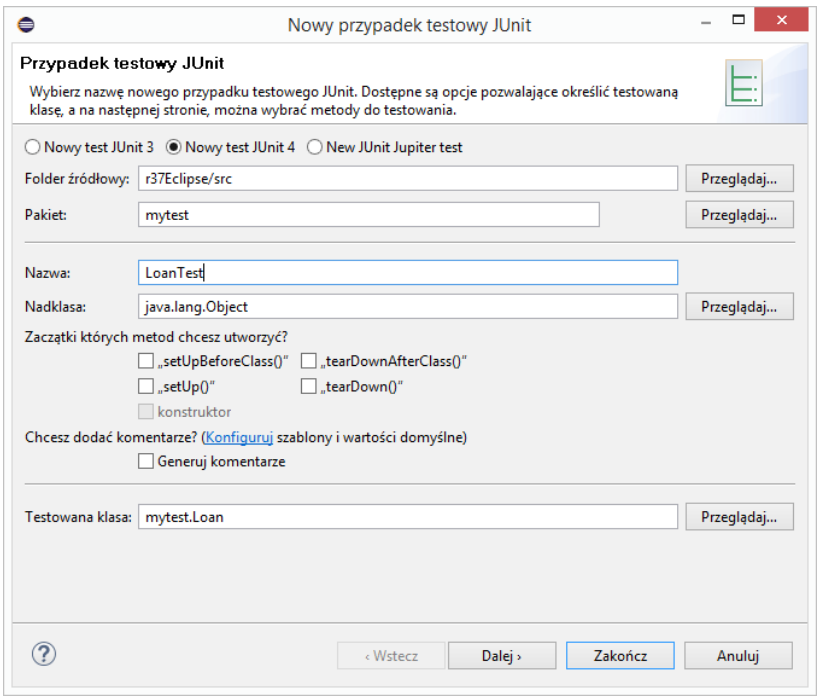
RYSUNEK 37.12. W oknie dialogowym Nowy projekt Java można utworzyć nowy projekt



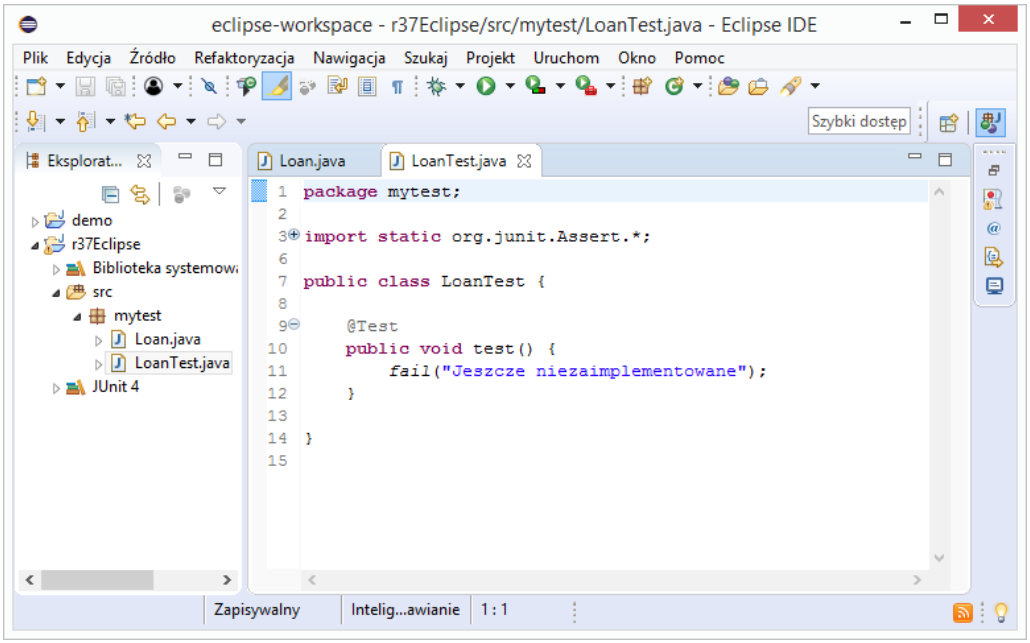
RYSUNEK 37.13. W oknie dialogowym Nowa klasa Java można utworzyć nową klasę Javy



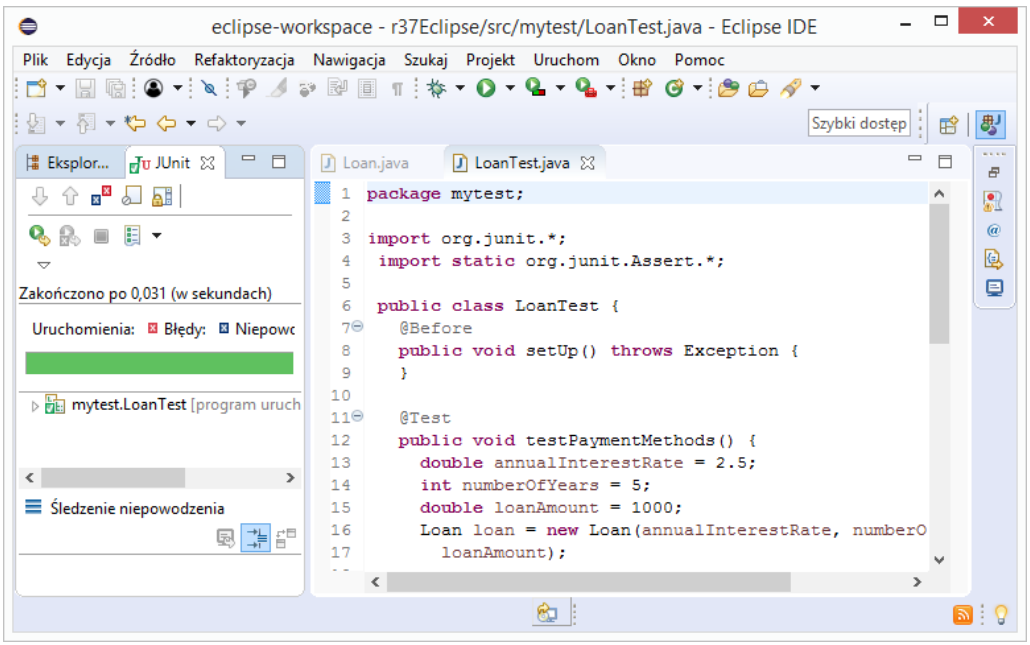
RYSUNEK 37.14. Środowisko tworzy klasę Loan



RYSUNEK 37.15. W oknie dialogowym Nowy przypadek testowy JUnit można utworzyć klasę testów



RYSUNEK 37.16. Klasa LoanTest jest generowana automatycznie



RYSUNEK 37.17. Po wykonaniu klasy LoanTest wyświetlany jest raport na temat testów

NAJWAŻNIEJSZE POJĘCIA

JUnit	klasa testów
JUnitCore	narzędzie uruchamiające testy

PODSUMOWANIE ROZDZIAŁU

1. JUnit jest otwartą platformą do testowania programów w Javie.
2. Aby przetestować klasę Javy, utwórz dla niej klasę testów i użyj narzędzia uruchamiającego testy z JUnit do wykonania klasy testów w celu wygenerowania raportu.
3. Klasy testów można tworzyć i uruchamiać w wierszu poleceń, a także w narzędziach takich jak środowiska NetBeans i Eclipse.

Quiz

Rozwiąż dotyczący tego rozdziału quiz w witrynie powiązanej z oryginalnym wydaniem książki.

ĆWICZENIA PROGRAMISTYCZNE

37.1. Napisz klasę testów do przetestowania metod `length`, `charAt`, `substring` i `indexOf` z klasy `java.lang.String`.

37.2. Napisz klasę testów do przetestowania metod `add`, `remove`, `addAll`, `removeAll`, `size`, `isEmpty` i `contains` z klasy `java.util.HashSet`.

37.3. Napisz klasę testów do przetestowania metody `isPrime` z listingu 6.7, *PrimeNumberMethod.java*.

37.4. Napisz klasę testów do przetestowania metod `getBMI` i `getStatus` z klasy `BMI` z listingu 10.4.