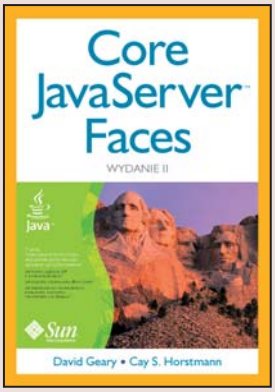
# JavaServer Faces. Wydanie II




Autor: David Geary, Cay S. Horstmann
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-1354-0
Tytuł oryginału: Core JavaServer Faces (Core Series)
Format: 172x245, stron: 648

**Poznaj nowoczesne technologie, aby perfekcyjnie tworzyć sprawne oprogramowanie!**

- Jak tworzyć aplikacje JSF w środowisku Eclipse?
- Jak korzystać z frameworku Boss Seam?
- Jak implementować niestandardowe komponenty, konwertery i mechanizmy weryfikujące?

JavaServer Faces (JSF) jest obecnie najpopularniejszą technologią, która pozwala na projektowanie interfejsu użytkownika poprzez umieszczenie na formularzu komponentów i powiązanie ich z obiektami Javy bez konieczności mieszania kodu źródłowego ze znacznikami. Mocną stroną JSF jest rozszerzalny model komponentowy, a także możliwość współpracy z środowiskami do budowy graficznych interfejsów użytkownika metodą przeciągnij-i-upuść. To nowoczesne narzędzie oferuje także mechanizmy rozwiązujące najtrudniejsze problemy w zakresie nawigacji, zarządzania połączeniami z usługami zewnętrznymi i konfiguracjami oraz izolowania prezentacji od logiki biznesowej.

Książka „JavaServer Faces. Wydanie II" prezentuje gruntownie zaktualizowaną wiedzę dotyczącą JSF oraz wyczerpujące omówienia najnowszych udoskonaleń mechanizmów wiążących tę technologię z platformą Java EE 5, a także analizę rozszerzeń interfejsów API. Zawiera praktyczne porady i wskazówki, dzięki którym szybko nauczysz się technik sterowania przechodzeniem pomiędzy stronami z wykorzystaniem frameworku Shale; poznasz sposoby zastępowania znaczników JSP znacznikami XHTML za pomocą technologii Facelets; do perfekcji opanujesz sztukę rozbudowy tej technologii o własne biblioteki. „Core JavaServer Faces" to doskonały, usystematyzowany zbiór najlepszych praktyk budowy oprogramowania, minimalizowania trwale kodowanych elementów i maksymalizacji produktywności.

- Komponenty zarządzane
- Zaawansowane techniki nawigacji
- Znaczniki i formularze
- Konwersja i weryfikacja poprawności danych
- Implementacja klas niestandardowych mechanizmów weryfikacji
- Powidoki i pakiet Apache Tiles
- Niestandardowe komponenty, konwertery i mechanizmy weryfikujące
- Eliminowanie wycieków połączeń
- Uzyskiwanie dostępu do informacji składowych w katalogach LDAP
- Implementacja mechanizmu weryfikacji w czasie rzeczywistym z wykorzystaniem frameworku Ajax4jsf

**Zastosuj nowoczesne technologie JSP, aby w prosty sposób budować zaawansowane i sprawne aplikacje**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

# Spis treści

# 2

# Komponenty zarządzane

Centralnym elementem projektów aplikacji internetowych jest rozdział warstw prezentacji i logiki biznesowej. W technologii JavaServer Faces za taki rozdział odpowiadają **komponenty** (ang. *beans*). Strony JSF odwołują się do właściwości komponentów. Kod implementacji tych komponentów definiuje właściwą logikę programu. Ponieważ właśnie komponenty są kluczem do programowania aplikacji JSF, w niniejszym rozdziale skoncentrujemy się na ich szczegółowym omówieniu.

W pierwszej części tego rozdziału przeanalizujemy podstawowe elementy komponentów, o których powinien wiedzieć każdy programista aplikacji JSF. W dalszej części omówimy program przykładowy, który dobrze ilustruje łączne funkcjonowanie tych elementów. W pozostałych podrozdziałach skupimy się na technicznych aspektach konfiguracji komponentów i wyrażeń reprezentujących wartości. Czytelnicy, którzy mają tę książkę w rękach po raz pierwszy, mogą te podrozdziały pominąć i wrócić do nich w przyszłości.

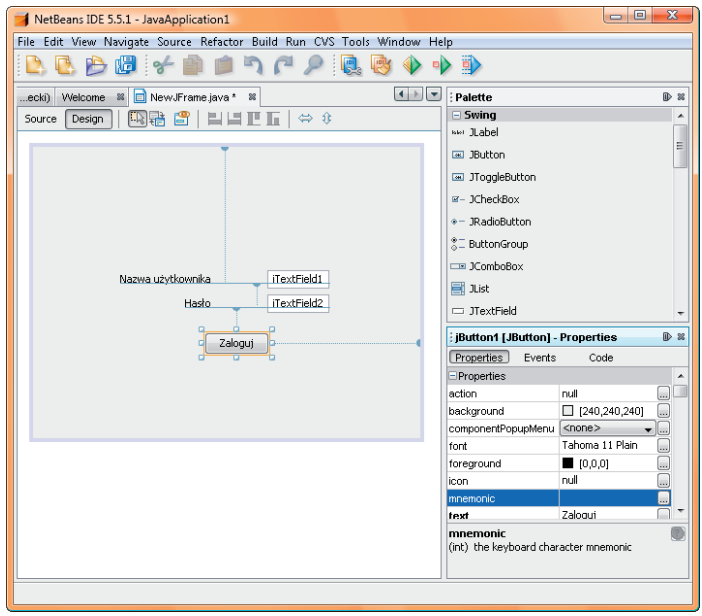
## Definicja komponentu

Zgodnie ze specyfikacją JavaBeans (dostępną na stronie internetowej *http://java.sun.com ↪/products/javabeans/*) komponent Javy ma postać „wielokrotnego komponentu oprogramowania, który można modyfikować za pomocą narzędzia do projektowania". Przytoczona definicja jest bardzo szeroka, co jest o tyle zrozumiałe, że — jak się niedługo okaże — komponenty można wykorzystywać do rozmaitych celów.

Na pierwszy rzut oka komponent wydaje się bardzo podobny do zwykłego obiektu Javy. Okazuje się jednak, że komponent Javy odpowiada za realizację innych zadań. Obiekty są tworzone i wykorzystywane w ramach programów Javy (odpowiednio przez wywołania konstruktorów i wywołania metod). Nieco inaczej jest w przypadku komponentów, które mogą być konfigurowane i wykorzystywane *bez konieczności programowania*.

Część Czytelników zapewne zastanawia się, skąd się wziął angielski termin *bean*. W Stanach Zjednoczonych słowo *Java* jest synonimem kawy, a smak kawy kryje się w jej ziarnach (ang. *beans*). Opisywana analogia części programistów wydaje się wyjątkowo zmyślna, innych po prostu denerwuje — nam nie pozostaje nic innego, jak pogodzić się z przyjętą terminologią.

„Klasycznym" zastosowaniem komponentów JavaBeans jest konstruowanie interfejsów użytkownika. Okno palety oferowane przez narzędzia do projektowania takich interfejsów obejmuje takie komponenty jak pola tekstowe, suwaki, pola wyboru itp. Zamiast samodzielnie pisać kod Swinga, korzystamy z narzędzia do projektowania interfejsu, który umożliwia nam przeciąganie komponentów dostępnych na palecie i upuszczanie ich na tworzonym formularzu. Możemy następnie dostosowywać właściwości tych komponentów przez ustawianie odpowiednich wartości w *odpowiednim oknie* dialogowym (patrz rysunek 2.1).




**Rysunek 2.1.** *Dostosowywanie komponentu do potrzeb aplikacji za pośrednictwem narzędzia do projektowania graficznego interfejsu użytkownika*

W technologii JavaServer Faces zadania komponentów nie kończą się na obsłudze elementów interfejsu użytkownika. Stosujemy je za każdym razem, gdy musimy wiązać klasy Javy ze stronami WWW lub plikami konfiguracyjnymi.

Wróćmy na chwilę do aplikacji login przedstawionej w podrozdziale „Prosty przykład" w rozdziale 1. Egzemplarz komponentu `UserBean` skonfigurowano w pliku *faces-config.xml*:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Znaczenie tych zapisów można by wyrazić następującymi słowami: skonstruuj obiekt klasy `com.corejsf.UserBean`, nadaj mu nazwę `user` i utrzymuj go przy życiu w czasie trwania *sesji* (czyli dla wszystkich żądań wygenerowanych przez tego samego klienta).

Do tak zdefiniowanego komponentu JavaBean można uzyskiwać dostęp z poziomu komponentów JSF. Na przykład poniższe pole tekstowe odczytuje i aktualizuje właściwość `password` komponentu `user`:

```
<h:inputSecret value="#{user.password}"/>
```

Jak widać, programista aplikacji JavaServer Faces nie musi pisać żadnego kodu konstruującego i operującego na komponencie `user`. Za konstruowanie komponentów zgodnie z elementami zdefiniowanymi w znaczniku `managed-bean` (w ramach pliku konfiguracyjnego) odpowiada implementacja JSF.

W aplikacjach JSF komponenty są zwykle wykorzystywane do następujących celów:

- w roli komponentów interfejsu użytkownika (w formie tradycyjnych komponentów interfejsu);
- w roli elementów łączących w jedną całość zachowanie formularza internetowego (są to tzw. komponenty wspomagające; ang. *backing beans*);
- w roli obiektów biznesowych, których właściwości są wyświetlane na stronach WWW;
- w roli takich usług jak zewnętrzne źródła danych wymagające skonfigurowania w momencie wdrażania aplikacji.

Z uwagi na wszechobecność i uniwersalność komponentów skoncentrujemy się wyłącznie na tych elementach specyfikacji JavaBeans, które znajdują zastosowanie z perspektywy programistów aplikacji JavaServer Faces.

## Właściwości komponentu

Klasy komponentów muszą być tworzone w zgodzie z pewnymi konwencjami programowania, aby odpowiednie narzędzia mogły swobodnie operować na ich składowych. W niniejszym punkcie zajmiemy się właśnie tymi konwencjami.

Do najważniejszych elementów klasy komponentu należą udostępniane właściwości. **Właściwością** jest każdy atrybut komponentu, który obejmuje:

- nazwę,
- typ,
- metody zwracające i (lub) ustawiające wartość tej właściwości.

Na przykład klasa `UserBean` zdefiniowana i wykorzystywana w poprzednim przykładzie zawiera właściwość typu `String` nazwaną `password`. Za zapewnianie dostępu do wartości tej właściwości odpowiadają metody `getPassword` i `setPassword`.

Niektóre języki programowania, w szczególności Visual Basic i C#, oferują bezpośrednią obsługę właściwości. Z drugiej strony, w języku Java komponent ma postać zwykłej klasy zaimplementowanej zgodnie z pewnymi konwencjami kodowania.

Specyfikacja JavaBeans definiuje pojedyncze wymaganie stawiane klasom komponentów — każda taka klasa musi definiować publiczny konstruktor domyślny, czyli konstruktor bezparametrowy. Z drugiej strony, aby definiowanie właściwości było możliwe, twórca komponentu musi albo stosować *wzorzec nazewniczy* dla metod zwracających i ustawiających, albo definiować deskryptory właściwości. Drugi model jest dość kłopotliwy i nie zyskał popularności, więc nie będziemy się nim zajmować. Szczegółowe omówienie tego zagadnienia można znaleźć w rozdziale 8. książki *Core Java™ 2, vol. 2 — Advanced Features (7th ed.)*[1] autorstwa Caya Horstmanna i Gary'ego Cornella.

Definiowanie właściwości w zgodzie ze wzorcami nazewniczymi jest bardzo proste. Przeanalizujmy teraz następującą parę metod:

```
public T getFoo()
public void setFoo(T newValue)
```

Przedstawiona para definiuje właściwość dostępną do odczytu i zapisu, typu `T`, nazwaną `foo`. Gdybyśmy użyli tylko pierwszej z powyższych metod, nasza właściwość byłaby dostępna tylko do odczytu. Pozostawienie samej drugiej metody oznaczałoby, że jest to właściwość dostępna tylko do zapisu.

Nazwy i sygnatury metod muszą gwarantować pełną zgodność z tym wzorcem. Nazwa metody musi się rozpoczynać od przedrostka `get` lub `set`. Metoda `get` nie może otrzymywać żadnych parametrów. Metoda `set` musi otrzymywać jeden parametr, ale nie może zwracać żadnych wartości. Klasa komponentu może zawierać inne metody (odbiegające od opisanej konwencji), które jednak nie definiują dodatkowych właściwości.

Warto pamiętać, że nazwa samej właściwości jest identyczna jak nazwa metod uzyskujących dostęp do jej wartości po usunięciu przedrostka `get` lub `set` oraz zamianie pierwszej litery na małą. Tworząc na przykład metodę `getFoo`, definiujemy właściwość nazwaną `foo` (z pierwszą literą `F` zamienioną na `f`). Jeśli jednak za przedrostkiem znajdują się co najmniej *dwie* wielkie litery, pierwsza litera nazwy właściwości pozostaje niezmieniona. Na przykład metoda nazwana `getURL` definiuje właściwość `URL`, nie `uRL`.

W przypadku właściwości typu `boolean` mamy do wyboru dwa różne prefiksy dla metod zwracających ich wartości. Obie wersje:

```
public boolean isConnected()
```

i

```
public boolean getConnected()
```

są prawidłowymi nazwami metod zwracających wartość właściwości `connected`.

Specyfikacja JavaBeans milczy na temat *zachowań* metod zwracających i ustawiających. W wielu przypadkach działanie tych metod sprowadza się do prostego operowania na polach egzemplarza. Z drugiej strony, te same metody mogą równie dobrze obejmować bardziej wyszukane operacje, jak dostęp do bazy danych, konwersję danych, weryfikację danych itd.

[1] Polskie wydanie: *Java 2. Techniki zaawansowane. Wydanie II*, Helion, 2005 — *przyp. tłum.*

Specyfikacja JavaBeans przewiduje też możliwość stosowania właściwości indeksowanych definiowanych w formie zbiorów metod podobnych do tego, który przedstawiono poniżej:

```
public T[] getFoo()
public T getFoo(int index)
public void setFoo(T[] newArray)
public void setFoo(int index, T newValue)
```

Okazuje się jednak, że technologia JSF nie zapewnia obsługi operacji dostępu do indeksowanych wartości.

Klasa komponentu może też zawierać metody inne niż te odpowiedzialne za zwracanie i ustawianie wartości właściwości. Takie metody oczywiście nie definiują kolejnych właściwości komponentu.

## Wyrażenia reprezentujące wartości

Wiele komponentów interfejsu użytkownika aplikacji JSF definiuje atrybut `value`, który umożliwia określanie wartości bądź odwołania do wartości uzyskiwanej z właściwości komponentu JavaBean. Wartość stosowaną bezpośrednio można zdefiniować na przykład w następujący sposób:

```
<h:outputText value="Witaj świecie!"/>
```

Można też użyć wyrażenia reprezentującego wartość:

```
<h:outputText value="#{user.name}"/>
```

W większości przypadków wyrażenia podobne do `#{user.name}` odwołują się do właściwości. Warto pamiętać, że wyrażenie w tej formie może być stosowane nie tylko do odczytywania wartości, ale też do ich zapisywania, jeśli zostanie użyte dla komponentu wejściowego:

```
<h:inputText value="#{user.name}"/>
```

W momencie renderowania danego komponentu zostanie wywołana metoda zwracająca właściwość będącą przedmiotem odwołania. Metoda ustawiająca zostanie wywołana w czasie przetwarzania odpowiedzi użytkownika.

Wyrażenia reprezentujące wartości stosowane w aplikacjach JSF mają ścisły związek z językiem wyrażeń znanym z technologii JSP. Dla tego rodzaju wyrażeń stosuje się jednak separator `${...}` (zamiast separatora `#{...}`). W standardach JSF 1.2 i JSP 1.2 zunifikowano składnię obu języków wyrażeń. (Kompletną analizę tej składni można znaleźć w podrozdziale „Składnia wyrażeń reprezentujących wartości").

Separator `${...}` oznacza, że dane wyrażenie ma zostać przetworzone *natychmiast*, czyli w czasie przetwarzania odpowiedniej strony przez serwer aplikacji. Separator `#{...}` stosuje się dla wyrażeń, które mają być przetwarzane *możliwie późno*. W takim przypadku serwer aplikacji zachowuje wyrażenie w niezmienionej formie i przystępuje do jego przetwarzania dopiero wtedy, gdy odpowiednia wartość jest naprawdę potrzebna.

Warto pamiętać, że wyrażenia odroczone stosuje się dla wszystkich właściwości komponentów JSF, natomiast wyrażenia przetwarzane natychmiast są wykorzystywane dla tradycyjnych konstrukcji JSP lub JSTL (od ang. *JavaServer Pages Standard Template Library*), które rzadko są niezbędne w procesie tworzenia stron JSF.

Szczegółowe omówienie składni wyrażeń reprezentujących wartości można znaleźć w podrozdziale „Składnia wyrażeń reprezentujących wartości”.

# Pakiety komunikatów

Implementując aplikację internetową, warto rozważyć zgromadzenie wszystkich łańcuchów komunikatów w jednym, centralnym miejscu. Taki proces ułatwi zachowanie spójności komunikatów i — co bardzo ważne — upraszcza lokalizowanie (internacjonalizację) aplikacji z myślą o różnych ustawieniach regionalnych. W tym podrozdziale przeanalizujemy mechanizmy technologii JSF umożliwiające organizację komunikatów. W podrozdziale „Przykładowa aplikacja” przeanalizujemy przykład komponentów zarządzanych pracujących łącznie z pakietami komunikatów.

Łańcuchy komunikatów należy zebrać w pliku, którego format odpowiada przyjętemu porządkowi chronologicznemu:

```
guessNext=Odgadnij następną liczbę w sekwencji!
answer=Twoja odpowiedź:
```

Precyzyjne omówienie formatu tego pliku można znaleźć w dokumentacji API dla metody `load` klasy `java.util.Properties`.

Należy ten plik zapisać w katalogu, w którym składujemy klasy — np. jako *insrc/java/com/*
*➥corejsf/messages.properties*. Można oczywiście wybrać dowolną ścieżkę do katalogu i nazwę pliku, jednak musimy użyć rozszerzenia *.properties*.

Pakiety komunikatów możemy deklarować na dwa sposoby. Najprostszym rozwiązaniem jest umieszczenie następujących elementów w pliku konfiguracyjnym *faces-config.xml*:

```
<application>
   <resource-bundle>
      <base-name>com.corejsf.messages</base-name>
      <var>msgs</var>
   </resource-bundle>
</application>
```

Alternatywnym rozwiązaniem jest dodanie elementu `f:loadBundle` do każdej strony JavaServer Faces wymagającej dostępu do danego pakietu komunikatów:

```
<f:loadBundle basename="com.corejsf.messages" var="msgs"/>
```

W obu przypadkach dostęp do komunikatów wchodzących w skład pakietu odbywa się za pośrednictwem zmiennej odwzorowania (mapy) nazwanej `msgs`. (Nazwa bazowa, czyli `com.corejsf.messages`, przypomina nazwę klasy i rzeczywiście okazuje się, że plik właściwości jest wczytywany przez mechanizm ładowania klas).

Możemy teraz uzyskiwać dostęp do łańcuchów komunikatów w ramach wyrażeń reprezentujących wartości:

```
<h:outputText value="#{msgs.guessNext}"/>
```

To wszystko, czego nam trzeba! Kiedy zdecydujemy się na lokalizację naszej aplikacji z myślą o innych ustawieniach regionalnych, będziemy musieli tylko opracować odpowiednie pliki z komunikatami.

Element `resource-bundle` zapewnia większą efektywność niż element `f:loadBundle`, ponieważ zapewnia, że pakiet komunikatów będzie odczytywany jednorazowo dla całej aplikacji. Warto jednak pamiętać, że element `resource-bundle` jest obsługiwany, począwszy od wersji JSF 1.2. Oznacza to, że jeśli chcemy zapewnić zgodność naszej aplikacji ze specyfikacją JSF 1.1, musimy się posługiwać elementem `f:loadBundle`.

Lokalizując pakiety komunikatów, musimy nadawać odpowiednim plikom nazwy z przyrostkiem właściwym dla ustawień regionalnych, czyli dwuliterowego kodu języka (zgodnego ze standardem ISO-639) poprzedzonego znakiem podkreślenia. Na przykład łańcuchy w języku niemieckim należałoby zdefiniować w pliku *com/corejsf/messages_de.properties*.

Listę wszystkich dwu- i trzyliterowych kodów języków standardu ISO-639 można znaleźć na stronie internetowej *http://www.loc.gov/standards/iso639-2/*.

Mechanizmy obsługi umiędzynarodawiania aplikacji Javy automatycznie ładują pakiety komunikatów właściwe dla bieżących ustawień regionalnych. Pakiet domyślny (bez przyrostka języka ISO-639) jest swoistym zabezpieczeniem na wypadek, gdyby zlokalizowany pakiet był niedostępny. Szczegółowe omówienie problemu umiędzynarodawiania aplikacji Javy można znaleźć w rozdziale 10. książki *Core Java™ 2, vol. 2 — Advanced Features (7th ed.)*[2] autorstwa Caya Horstmanna i Gary'ego Cornella.

Przygotowując tłumaczenia aplikacji, należy mieć na uwadze jeden dość specyficzny aspekt — pliki pakietów komunikatów nie są kodowane z wykorzystaniem schematu UTF-8. Znaki Unicode spoza zbioru pierwszych 127 znaków są kodowane za pomocą sekwencji specjalnej `\uxxxx`. Tego rodzaju pliki można tworzyć za pomocą narzędzia `native2ascii` pakietu Java SDK.

Dla tych samych ustawień regionalnych może istnieć wiele pakietów komunikatów. Możemy na przykład opracować odrębne pakiety dla najczęściej wykorzystywanych komunikatów o błędach.

## Komunikaty obejmujące zmienne

Komunikaty często obejmują elementy zmienne, które — co oczywiste — wymagają wypełnienia przed wyświetleniem. Przypuśćmy na przykład, że chcemy wyświetlić na ekranie zdanie *Masz n punktów*, gdzie *n* jest wartością uzyskiwaną z komponentu. W tym celu musimy utworzyć łańcuch źródłowy z symbolem zastępczym:

```
currentScore=Masz {0} punktów.
```

[2] Polskie wydanie: *Java 2. Techniki zaawansowane. Wydanie II*, Helion, 2005 — *przyp. tłum.*

Symbole zastępcze (ang. *placehorders*) mają postać {0}, {1}, {2} itd. W kodzie naszej strony JSF powinniśmy użyć znacznika h:outputFormat i zdefiniować wartości dla tych symboli zastępczych w formie elementów potomnych f:param:

```
<h:outputFormat value="#{msgs.currentScore}">
   <f:param value="#{quiz.score}"/>
</h:outputFormat>
```

Znacznik h:outputFormat wykorzystuje do formatowania łańcuchów komunikatów klasę MessageFormat biblioteki standardowej. Wspomniana klasa oferuje szereg mechanizmów opracowanych z myślą o formatowaniu łańcuchów zależnych od ustawień regionalnych.

Za pomocą przyrostka number,currency dołączanego do symbolu zastępczego możemy formatować liczby jako kwoty w lokalnej walucie:

```
currentTotal=Dysponujesz kwotą {0,number,currency}.
```

W Stanach Zjednoczonych wartość 1023.95 zostanie sformatowana jako $1,023.95. Ta sama wartość w Niemczech zostałaby wyświetlona jako €1.023,95 (w Polsce 1 023,95 zł) — z uwzględnieniem symbolu lokalnej waluty i konwencji separatora części dziesiętnej.

Format choice umożliwia nam formatowanie liczb na różne sposoby (w zależności od wartości poprzedzającej jednostkę), czyli: *zero punktów*, *jeden punkt*, *dwa punkty*, *3 punkty*, *4 punkty*, *5 punktów* itd. Poniżej przedstawiono łańcuch formatu zapewniający oczekiwany efekt:

```
currentScore=Masz {0,choice,0#zero punktów|1#jeden punkt|2#dwa punkty|3#trzy punkty
    |4#cztery punkty|5#{0} punktów}.
```

Mamy do czynienia z sześcioma możliwymi przypadkami (0, 1, 2, 3, 4 oraz ≥5), z których każdy definiuje odrębny łańcuch komunikatu.

Łatwo zauważyć, że symbol zastępczy 0 występuje dwukrotnie: raz na etapie wyboru formatu i drugi raz w ostatniej, szóstej opcji, gdzie generuje wynik w postaci komunikatu *5 punktów*.

Przykłady użycia tej konstrukcji można znaleźć na listingach 2.5 i 2.6 w podrozdziale „Przykładowa aplikacja". O ile angielskie ustawienia regionalne (z komunikatem *Your score is ...*) w ogóle nie wymagają stosowania formatu choice w naszej przykładowej aplikacji, o tyle niemieckie wyrażenie *Sie haben ... punkte* (podobnie jak polskie *Masz ... punktów*) nie jest uniwersalne, ponieważ nie uwzględnia liczby pojedynczej *einen punkt*.

Dodatkowych informacji na temat klasy MessageFormat należy szukać w dokumentacji API lub w rozdziale 10. książki *Core Java™ 2, vol. 2 — Advanced Features (7th ed.)*[3] autorstwa Caya Horstmanna i Gary'ego Cornella.

## Konfigurowanie ustawień regionalnych aplikacji

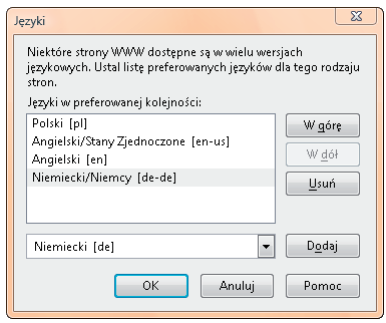
Po opracowaniu pakietów komunikatów musimy zdecydować, jak zdefiniować ustawienia regionalne naszej aplikacji. Mamy do dyspozycji trzy rozwiązania:

[3] Polskie wydanie: *Java 2. Techniki zaawansowane. Wydanie II*, Helion, 2005 — *przyp. tłum.*

1. Możemy pozostawić wybór ustawień regionalnych przeglądarce. Ustawienia domyślne i wszystkie ustawienia obsługiwane należy zdefiniować w pliku konfiguracyjnym *WEB-INF/faces-config.xml* (lub innym zasobie konfiguracyjnym aplikacji):

```
<faces-config>
   <application>
      <locale-config>
         <default-locale>pl</default-locale>
         <supported-locale>de</supported-locale>
      </locale-config>
  </application>
</faces-config>
```

   Kiedy przeglądarka nawiązuje połączenie z naszą aplikacją, zwykle dołącza do nagłówka protokołu HTTP wartość `Accept-Language` (patrz *http://www.w3.org/↪International/questions/qa-accept-lang-locales.html*). Implementacja technologii JavaServer Faces odczytuje ten nagłówek i odnajduje najlepsze dopasowanie wśród obsługiwanych ustawień regionalnych. Możemy sprawdzić funkcjonowanie tego mechanizmu, ustawiając język preferowany w swojej przeglądarce (patrz rysunek 2.2).




**Rysunek 2.2.** *Wybór preferowanego języka*

2. Możemy dodać atrybut `locale` do elementu `f:view` — oto przykład:

```
<f:view locale="de">
```

   Ustawienia regionalne można też określać dynamicznie:

```
<f:view locale="#{user.locale}"/>
```

   Kod reprezentujący ustawienia regionalne ma teraz postać łańcucha zwracanego przez metodę `getLocale`. Takie rozwiązanie jest wygodne w przypadku aplikacji, które umożliwiają użytkownikowi wybór preferowanych ustawień.

3. Ustawienia regionalne można też definiować programowo. W tym celu wystarczy wywołać metodę `setLocale` obiektu klasy `UIViewRoot`:

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

   Przykład praktycznego użycia tego mechanizmu można znaleźć w punkcie „Stosowanie łączy poleceń” w rozdziale 4.

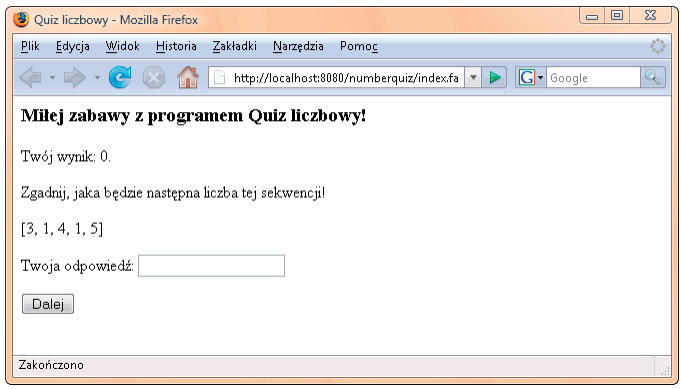
# Przykładowa aplikacja

Po omówieniu tych dość abstrakcyjnych reguł i obostrzeń warto przystąpić do analizy konkretnego przykładu. Działanie naszej aplikacji będzie się sprowadzało do prezentowania szeregu pytań swoistego quizu. Każde z nich będzie obejmowało sekwencję cyfr i wymagało od użytkownika wskazania kolejnej cyfry danej sekwencji.

Na rysunku 2.3 przedstawiono przykładowy ekran aplikacji z prośbą o podanie kolejnej liczby dla następującej sekwencji:

```
3 1 4 1 5
```

**Rysunek 2.3.**
*Quiz liczbowy*




Podobne łamigłówki często są stosowane w testach na inteligencję. Rozwiązanie tej zagadki wymaga znalezienia pewnego wzorca (w tym przypadku mamy do czynienia z pierwszymi cyframi liczby $\pi$).

Kiedy prawidłowo określimy kolejną cyfrę tej sekwencji (w tym przypadku `9`), nasz wynik zostanie powiększony o jeden punkt.

Istnieje angielski zwrot (wprost idealnie pasujący do środowisk opartych na Javie), którego zapamiętanie ułatwia identyfikację pierwszych ośmiu cyfr liczby $\pi$: *Can I have a small container of coffee?* Wystarczy policzyć litery w kolejnych wyrazach, aby otrzymać sekwencję `3 1 4 1 5 9 2 6`. Więcej informacji o podobnych konstrukcjach można znaleźć na stronie internetowej *http://dir.yahoo.com/Science/Mathematics/Numerical ↪_Analysis/Numbers/Specific_Numbers/Pi/Mnemonics/*.

W naszym przykładzie kolejne pytania zaimplementujemy w klasie `QuizBean`. W rzeczywistej aplikacji najprawdopodobniej wykorzystalibyśmy bazę danych do składowania tego rodzaju informacji. Celem tego przykładu jest jednak demonstracja sposobu wykorzystywania komponentów JavaBeans o złożonej strukturze.

W pierwszej kolejności przeanalizujemy kod klasy `ProblemBean`, która definiuje dwie właściwości: `solution` typu `int` oraz `sequence` typu `ArrayList` (patrz listing 2.1).

**Listing 2.1.** *Kod zdefiniowany w pliku numberquiz/src/java/com/corejsf/ProblemBean.java*

```
1. package com.corejsf;
2. import java.util.ArrayList;
3.
4. public class ProblemBean {
5.    private ArrayList<Integer> sequence;
6.    private int solution;
7.
8.    public ProblemBean() {}
9.
10.   public ProblemBean(int[] values, int solution) {
11.      sequence = new ArrayList<Integer>();
12.      for (int i = 0; i < values.length; i++)
13.         sequence.add(values[i]);
14.      this.solution = solution;
15.   }
16.
17.   // właściwość sequence:
18.   public ArrayList<Integer> getSequence() { return sequence; }
19.   public void setSequence(ArrayList<Integer> newValue) { sequence = newValue; }
20.
21.   // właściwość solution:
22.   public int getSolution() { return solution; }
23.   public void setSolution(int newValue) { solution = newValue; }
24. }
```

Musimy teraz zdefiniować właściwy komponent quizu z następującymi właściwościami:

- `problems`: właściwość dostępna tylko do zapisu i reprezentująca pytania quizu;
- `score`: właściwość dostępna tylko do odczytu i reprezentująca bieżącą punktację;
- `current`: właściwość dostępna tylko do odczytu i reprezentująca bieżące pytanie;
- `answer`: właściwość umożliwiająca odczyt i zapis bieżącej odpowiedzi podanej przez użytkownika.

Właściwość `problems` w ogóle nie jest wykorzystywana przez nasz przykładowy program — ograniczamy się do inicjalizacji zbioru problemów w konstruktorze klasy `QuizBean`. Z drugiej strony, w punkcie „Wiązanie definicji komponentów” w dalszej części tego rozdziału omówimy sposób definiowania zbioru problemów wewnątrz pliku *faces-config.xml*, a więc bez konieczności pisania jakiegokolwiek kodu.

Właściwość `current` jest wykorzystywana do wyświetlania bieżącego problemu. Warto jednak pamiętać, że wspomniana właściwość reprezentuje obiekt klasy `ProblemBean`, którego nie możemy bezpośrednio wyświetlać w polu tekstowym. W związku z tym sekwencję liczb uzyskujemy za pośrednictwem jeszcze jednej właściwości:

```
<h:outputText value="#{quiz.current.sequence}"/>
```

Właściwość `sequence` reprezentuje wartość typu `ArrayList`. W procesie wyświetlania można ją przekonwertować na łańcuch, wywołując metodę `toString`. W wyniku tego wywołania otrzymamy następujący łańcuch wynikowy:

```
[3, 1, 4, 1, 5]
```

I wreszcie musimy się zmierzyć z problemem obsługi właściwości `answer`. W pierwszej kolejności wiążemy ją z polem tekstowym formularza:

```
<h:inputText value="#{quiz.answer}"/>
```

W momencie wyświetlania tego pola tekstowego następuje wywołanie metody zwracającej `getAnswer`, którą zaimplementowano w taki sposób, aby zwracała pusty łańcuch.

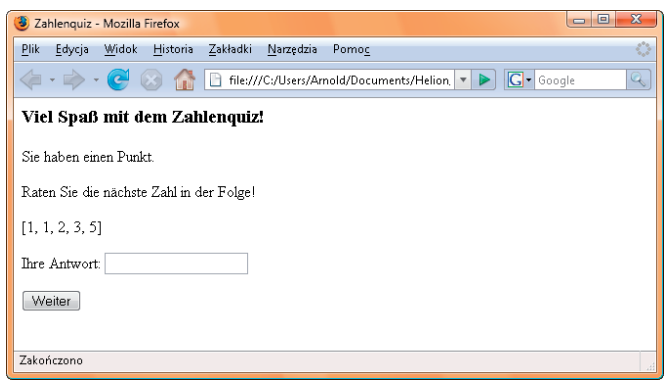
Po wysłaniu formularza implementacja JSF wywołuje metodę ustawiającą, przekazując na jej wejściu wartość wpisaną przez użytkownika w tym polu tekstowym. Metoda `setAnswer` sprawdza odpowiedź, aktualizuje wynik punktowy (w przypadku prawidłowej decyzji użytkownika) i przechodzi do kolejnego problemu.

```
public void setAnswer(String newValue) {
   try {
      int answer = Integer.parseInt(newValue.trim());
      if (getCurrent().getSolution() == answer) score++;
      currentIndex = (currentIndex + 1) % problems.size();
   }
   catch (NumberFormatException ex) {
   }
}
```

Umieszczanie w metodzie ustawiającej wartość właściwości kodu niezwiązanego z jej oryginalnym przeznaczeniem nie jest najlepszym rozwiązaniem. Operacje aktualizacji wyniku i przejścia do kolejnego problemu powinny być realizowane przez metodę obsługującą akcję kliknięcia przycisku. Ponieważ jednak nie analizowaliśmy mechanizmów reagujących na tego rodzaju zdarzenia, na razie będziemy korzystać z elastyczności oferowanej przez metody ustawiające.

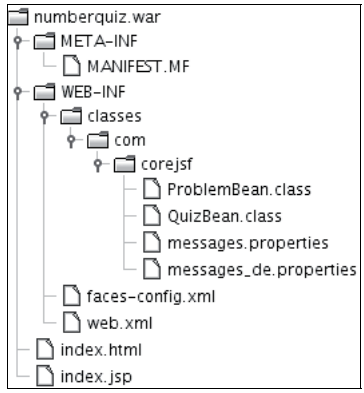
Inną wadą naszej przykładowej aplikacji jest brak mechanizmu przerywania działania po ostatnim pytaniu quizu. Ograniczyliśmy się do rozwiązania polegającego na powrocie do pierwszej strony, aby umożliwić użytkownikowi uzyskiwanie coraz lepszych rezultatów. W następnym rozdziale omówimy sposób implementacji lepszego modelu. Warto raz jeszcze przypomnieć, że celem tej aplikacji jest pokazanie, jak w praktyce konfigurować i wykorzystywać komponenty.

Na koniec warto zwrócić uwagę na mechanizm umiędzynarodowienia naszej aplikacji przez opracowanie dodatkowych pakietów komunikatów. Zachęcamy do przełączenia przeglądarki na język niemiecki, aby naszym oczom ukazał się ekran podobny do tego z rysunku 2.4.




**Rysunek 2.4.** *Viel Spaß mit dem Zahlenquiz!*

Na tym możemy zakończyć analizę naszej przykładowej aplikacji. Na rysunku 2.5 przedstawiono strukturę katalogów. Pozostały kod źródłowy przedstawiono na listingach od 2.2 do 2.6.




**Rysunek 2.5.** *Struktura katalogów przykładowej aplikacji łamigłówki liczbowej*

**Listing 2.2.** *Zawartość pliku numberquiz/web/index.jsp*

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

   <f:view>
      <head>
         <title><h:outputText value="#{msgs.title}"/></title>
      </head>
      <body>
         <h:form>
            <h3>
               <h:outputText value="#{msgs.heading}"/>
            </h3>
            <p>
               <h:outputFormat value="#{msgs.currentScore}">
                  <f:param value="#{quiz.score}"/>
               </h:outputFormat>
            </p>
            <p>
               <h:outputText value="#{msgs.guessNext}"/>
            </p>
            <p>
               <h:outputText value="#{quiz.current.sequence}"/>
            </p>
            <p>
               <h:outputText value="#{msgs.answer}"/>
               <h:inputText value="#{quiz.answer}"/></p>
            <p>
               <h:commandButton value="#{msgs.next}" action="next"/>
            </p>
         </h:form>
      </body>
   </f:view>
</html>
```

**Listing 2.3.** *Zawartość pliku numberquiz/src/java/com/corejsf/QuizBean.java*

```
 1. package com.corejsf;
 2. import java.util.ArrayList;
 3.
 4. public class QuizBean {
 5.    private ArrayList<ProblemBean> problems = new ArrayList<ProblemBean>();
 6.    private int currentIndex;
 7.    private int score;
 8.
 9.    public QuizBean() {
10.       problems.add(
11.          new ProblemBean(new int[] { 3, 1, 4, 1, 5 }, 9)); // liczba pi
12.       problems.add(
13.          new ProblemBean(new int[] { 1, 1, 2, 3, 5 }, 8)); // ciąg Fibonacciego
14.       problems.add(
15.          new ProblemBean(new int[] { 1, 4, 9, 16, 25 }, 36)); // kwadraty
16.       problems.add(
17.          new ProblemBean(new int[] { 2, 3, 5, 7, 11 }, 13)); // liczby pierwsze
18.       problems.add(
19.          new ProblemBean(new int[] { 1, 2, 4, 8, 16 }, 32)); // potęgi dwójki
20.    }
21.
22.    // Właściwość problems:
23.    public void setProblems(ArrayList<ProblemBean> newValue) {
24.       problems = newValue;
25.       currentIndex = 0;
26.       score = 0;
27.    }
28.
29.    // Właściwość score:
30.    public int getScore() { return score; }
31.
32.    // Właściwość current:
33.    public ProblemBean getCurrent() {
34.       return problems.get(currentIndex);
35.    }
36.
37.    // Właściwość answer:
38.    public String getAnswer() { return ""; }
39.    public void setAnswer(String newValue) {
40.       try {
41.          int answer = Integer.parseInt(newValue.trim());
42.          if (getCurrent().getSolution() == answer) score++;
43.          currentIndex = (currentIndex + 1) % problems.size();
44.       }
45.       catch (NumberFormatException ex) {
46.       }
47.    }
48. }
```

**Listing 2.4.** *Zawartość pliku numberquiz/web/WEB-INF/faces-config.xml*

```
1. <?xml version="1.0"?>
2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
    <application>
       <locale-config>
          <default-locale>pl</default-locale>
          <supported-locale>de</supported-locale>
       </locale-config>
    </application>

    <navigation-rule>
       <from-view-id>/index.jsp</from-view-id>
       <navigation-case>
          <from-outcome>next</from-outcome>
          <to-view-id>/index.jsp</to-view-id>
       </navigation-case>
    </navigation-rule>

     <managed-bean>
        <managed-bean-name>quiz</managed-bean-name>
        <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
     </managed-bean>

     <application>
        <resource-bundle>
           <base-name>com.corejsf.messages</base-name>
           <var>msgs</var>
        </resource-bundle>
     </application>
</faces-config>
```

**Listing 2.5.** *Zawartość pliku numberquiz/src/java/com/corejsf/messages.properties*

```
title=Quiz liczbowy
heading=Miłej zabawy z programem Quiz liczbowy!
currentScore=Twój wynik: {0}.
guessNext=Zgadnij, jaka będzie następna liczba tej sekwencji!
answer=Twoja odpowiedź:
next=Dalej
```

**Listing 2.6.** *Zawartość pliku numberquiz/src/java/com/corejsf/messsages_de.properties*

```
title=Zahlenquiz
heading=Viel Spa\u00df mit dem Zahlenquiz!
currentScore=Sie haben {0,choice,0#0 Punkte|1#einen Punkt|2#{0} Punkte}.
guessNext=Raten Sie die n\u00e4chste Zahl in der Folge!
answer=Ihre Antwort:
next=Weiter
```

# Komponenty wspierające

W niektórych sytuacjach najwygodniejszym rozwiązaniem jest zaprojektowanie komponentu obejmującego wybraną część lub wszystkie obiekty komponentów właściwe dla danego formularza WWW. Taki komponent określa się mianem **komponentu wspierającego, wspomagającego** (ang. *backing bean*).

Możemy zdefiniować komponent wspierający dla formularza quizu przez dodanie odpowiednich właściwości do komponentu tego formularza:

```
public class QuizFormBean {
   private UIOutput scoreComponent;
   private UIInput answerComponent;
   // Właściwość scoreComponent:
   public UIOutput getScoreComponent() { return scoreComponent; }
   public void setScoreComponent(UIOutput newValue) { scoreComponent = newValue; }

   // Właściwość answerComponent:
   public UIInput getAnswerComponent() { return answerComponent; }
   public void setAnswerComponent(UIInput newValue) { answerComponent = newValue; }
   ...
}
```

Komponenty wyjściowe interfejsu użytkownika należą do klasy `UIOutput`, natomiast komponenty wejściowe należą do klasy `UIInput`. Obie te klasy szczegółowo omówimy w punkcie „Zestaw narzędzi programisty komponentów niestandardowych" w rozdziale 9.

Po co właściwie mielibyśmy tworzyć tego rodzaju komponenty? Podczas lektury punktu „Weryfikacja relacji łączących wiele komponentów" w rozdziale 6. Czytelnicy dowiedzą się, że w niektórych przypadkach mechanizmy weryfikujące i obsługujące zdarzenia muszą mieć dostęp do komponentów interfejsu użytkownika na formularzu. Co więcej, komponenty wspierające (w środowisku Java Studio Creator nazywane **komponentami stron**, ang. *page beans*, z uwagi na konieczność ich dodawania do wszystkich stron) z reguły są wykorzystywane przez wizualne środowiska wytwarzania aplikacji JSF. Wspomniane środowiska automatycznie generują odpowiednie metody zwracające i ustawiające wartości właściwości dla wszystkich komponentów interfejsu graficznego przeciąganych na formularz.

Jeśli zdecydujemy się na użycie komponentu wspierającego, będziemy musieli związać komponenty interfejsu użytkownika dodane do formularza z komponentami wchodzącymi w skład komponentu wspierającego. Możemy zrealizować ten cel, stosując atrybut `binding`:

```
<h:outputText binding="#{quizForm.scoreComponent}"/>
```

Po skonstruowaniu drzewa komponentów dla danego formularza następuje wywołanie metody `getScoreComponent` komponentu wspomagającego, która jednak zwraca wartość `null`. Oznacza to, że komponent wyjściowy jest konstruowany i instalowany w ramach tego komponentu wspierającego za pomocą metody `setScoreComponent`.

Stosowanie komponentów wspierających w niektórych przypadkach jest uzasadnione, co nie zmienia faktu, że w pewnych sytuacjach mogą być nadużywane. Nigdy nie powinniśmy mieszać komponentów formularza z danymi biznesowymi w ramach jednego komponentu.

Jeśli komponenty wspierające wykorzystujemy dla danych warstwy prezentacji, dla obiektów biznesowych powinniśmy stosować odrębny zbiór komponentów.

# Zasięg komponentów

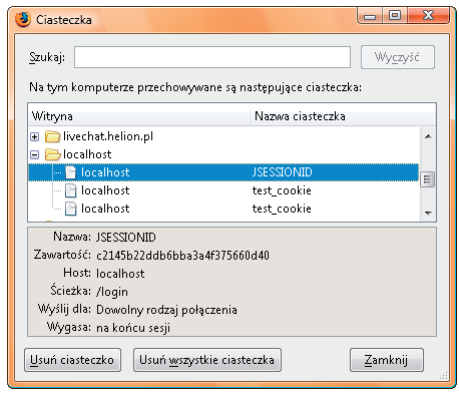
Z myślą o zapewnieniu wygody programistom aplikacji internetowych kontenery serwletów oferują odrębne zasięgi, z których każdy zarządza tabelą związków nazwa-wartość.

Każdy z tych zasięgów zwykle obejmuje komponenty i inne obiekty, które muszą być dostępne z poziomu innych komponentów aplikacji internetowej.

## Komponenty obejmujące zasięgiem sesję

Musimy pamiętać, że protokół HTTP jest *bezstanowy*. Przeglądarka wysyła żądanie na serwer, serwer zwraca odpowiedź, po czym żadna ze stron (ani przeglądarka, ani serwer) nie ma obowiązku utrzymywać w pamięci jakichkolwiek informacji o tej transakcji. Taki model zdaje egzamin w sytuacji, gdy przeglądarka żąda od serwera prostych informacji, ale okazuje się dalece niedoskonały w przypadku aplikacji pracujących po stronie serwera. Na przykład w aplikacji sklepu internetowego oczekujemy od serwera zachowywania zawartości koszyka z zakupami.

Właśnie dlatego kontenery serwletów rozszerzają protokół HTTP o mechanizm utrzymywania i śledzenia *sesji*, czyli następujących po sobie połączeń nawiązywanych przez tego samego klienta. Istnieje wiele różnych metod śledzenia sesji. Najprostszą z nich jest stosowanie znaczników kontekstu klienta (tzw. ciasteczek; ang. *cookies*) — par nazwa-wartość wysyłanych klientowi przez serwer w założeniu, że będą zwracane w ramach kolejnych żądań (patrz rysunek 2.6).




**Rysunek 2.6.** *Znacznik kontekstu klienta wysłany przez aplikację JSF*

Dopóki klient nie dezaktywuje znaczników kontekstu, serwer będzie otrzymywał identyfikator sesji wraz z kolejnymi żądaniami tego klienta.

Serwery aplikacji wykorzystują też strategie alternatywne (np. polegające na przepisywaniu adresów URL) do obsługi sesji nawiązywanych przez aplikacje klienckie, które nie zwracają znaczników kontekstu klienta. Strategia przepisywania adresów URL sprowadza się do dodawania do każdego adresu URL identyfikatora sesji podobnego do poniższego:

```
http://corejsf.com/login/index.jsp;jsessionid=b55cd6...d8e
```

Aby zweryfikować funkcjonowanie tego modelu, warto wymusić na przeglądarce odrzucanie znaczników kontekstu klienta z serwera *localhost*, po czym ponownie uruchomić naszą aplikację internetową i wysłać formularz. Kolejna strona powinna zawierać atrybut `jsessionid`.

Śledzenie sesji za pomocą znaczników kontekstu klienta nie wymaga żadnych dodatkowych działań ze strony programisty aplikacji, a standardowe znaczniki JavaServer Faces automatycznie zastosują strategię przepisywania adresów URL w przypadku aplikacji klienckich, które nie obsługują znaczników.

*Zasięg sesyjny* oznacza, że wszelkie dane są utrwalane od momentu ustanowienia sesji aż do chwili jej zakończenia. Sesja jest kończona wskutek wywołania przez aplikację internetową metody `invalidate` obiektu klasy `HttpSession` lub po upływie przyjętego limitu czasowego.

Aplikacje internetowe zwykle umieszczają większość swoich komponentów w zasięgu sesyjnym.

Na przykład komponent `UserBean` może przechowywać informacje o użytkownikach, które będą dostępne przez cały okres trwania sesji. Komponent `ShoppingCartBean` może być wypełniany stopniowo wraz z otrzymywaniem kolejnych żądań składających się na sesję.

## Komponenty obejmujące zasięgiem aplikację

Komponenty obejmujące swoim zasięgiem aplikację są utrzymywane przez cały czas wykonywania aplikacji. Odpowiedni zasięg jest współdzielony przez wszystkie żądania i wszystkie sesje.

Komponenty zarządzane umieszczamy w zasięgu aplikacji, jeśli chcemy, aby były dostępne z poziomu wszystkich egzemplarzy naszej aplikacji internetowej. Taki komponent jest konstruowany w odpowiedzi na pierwsze żądanie dowolnego egzemplarza aplikacji i jest utrzymywany przy życiu do momentu usunięcia tej aplikacji internetowej z serwera aplikacji.

## Komponenty obejmujące zasięgiem żądanie

Zasięg na poziomie żądań charakteryzuje się wyjątkowo krótkim czasem życia. Jest konstruowany w momencie wysłania żądania protokołu HTTP i utrzymywany do momentu odesłania odpowiedzi klientowi.

Jeśli umieścimy komponent zarządzany w zasięgu żądania, nowy egzemplarz tego komponentu będzie tworzony dla każdego żądania. Takie rozwiązanie jest nie tylko kosztowne, ale

też nie zdaje egzaminu w sytuacji, gdy chcemy zachowywać niezbędne dane pomiędzy żądaniami. W zasięgu żądania powinniśmy umieszczać obiekty tylko wtedy, gdy chcemy je przekazywać do innej fazy przetwarzania w ramach bieżącego żądania.

Na przykład znacznik `f:loadBundle` umieszcza w zasięgu żądania zmienną pakietu komunikatów, która jest potrzebna tylko w czasie trwania fazy *wizualizacji odpowiedzi* dla tego samego żądania.

Tylko komponenty obejmujące zasięgiem żądania mają charakter jednowątkowy i jako takie gwarantują bezpieczeństwo wątków. Co ciekawe, jednowątkowe nie są komponenty sesyjne. Użytkownik może na przykład jednocześnie wysyłać odpowiedzi z wielu okien przeglądarki internetowej. Każda odpowiedź jest przetwarzana przez odrębny wątek żądania. Gdybyśmy chcieli zapewnić bezpieczeństwo przetwarzania wielowątkowego w naszych komponentach sesyjnych, powinniśmy opracować odpowiednie mechanizmy blokowania.

## Adnotacje cyklu życia

Począwszy od wersji JSF 1.2, istnieje możliwość wskazywania metod komponentów zarządzanych, które mają być wywoływane automatycznie bezpośrednio po skonstruowaniu tych komponentów i bezpośrednio przed ich wyjściem poza odpowiedni zasięg. Takie rozwiązanie jest szczególnie wygodne w przypadku komponentów ustanawiających połączenia z zasobami zewnętrznymi, np. bazami danych.

Metody komponentu zarządzanego można oznaczać adnotacjami `@PostConstruct` lub `@PreDestroy`:

```
public class MyBean {
   @PostConstruct
   public void initialize() {
      // kod inicjalizujący
   }
   @PreDestroy
   public void shutdown() {
      // kod zamykający
   }

   // pozostałe metody komponentu
}
```

Tak oznaczone metody będą wywoływane automatycznie, pod warunkiem że dana aplikacja internetowa zostanie wdrożona w kontenerze zapewniającym obsługę standardu Java Specification Request (JSR) 250 (patrz *http://www.jcp.org/en/jsr/detail?id=250*). Adnotacje w tej formie są obsługiwane przez serwery aplikacji zgodne ze standardem Java EE 5, w tym serwer GlassFish. Oczekuje się, że w niedalekiej przyszłości adnotacje będą obsługiwane także przez kontenery autonomiczne (np. Tomcata).

# Konfigurowanie komponentów

W niniejszym podrozdziale omówimy techniki konfigurowania komponentów w plikach konfiguracyjnych. Ponieważ szczegółowe rozwiązania w tym zakresie mają charakter ściśle techniczny, część Czytelników zapewne zdecyduje się tylko przejrzeć ten podrozdział i wrócić do niego dopiero wtedy, gdy stanie przed koniecznością skonfigurowania komponentów ze złożonymi właściwościami.

Najczęściej wykorzystywanym plikiem konfiguracyjnym jest *WEB-INF/faces-config.xml*. Okazuje się jednak, że możemy ustawienia konfiguracyjne umieścić także w następujących miejscach:

- W plikach nazwanych *META-INF/faces-config.xml* w ramach dowolnych plików JAR wczytywanych przez mechanizmy ładowania klas kontekstu zewnętrznego. (Opisywany mechanizm jest wykorzystywany w sytuacji, gdy dostarczamy komponenty wielokrotnego użytku w formie plików JAR).
- W plikach zdefiniowanych za pośrednictwem parametru inicjalizacji *javax.faces.*
  ➥*CONFIG_FILES* w ramach pliku *WEB-INF/web.xml*. Przykład definicji tego parametru przedstawiono poniżej:

```
<web-app>
   <context-param>
      <param-name>javax.faces.CONFIG_FILES</param-name>
      <param-value>WEB-INF/navigation.xml,WEB-INF/beans.xml</param-value>
   </context-param>
   ...
</web-app>
```

  (Opisany mechanizm jest szczególnie korzystny, jeśli korzystamy z narzędzi do projektowania oprogramowania, ponieważ skutecznie izoluje nawigację, komponenty itd.).

Dla uproszczenia będziemy w tym rozdziale stosowali tylko plik konfiguracyjny *WEB-INF/*
➥*faces-config.xml*.

Komponent jest konfigurowany przez element `managed-bean` wewnątrz elementu `faces-config` najwyższego poziomu. Dla każdego tak konfigurowanego komponentu musimy określić przynajmniej nazwę, klasę i zasięg:

```
<faces-config>
   <managed-bean>
      <managed-bean-name>user</managed-bean-name>
      <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>
</faces-config>
```

Zasięg może być reprezentowany przez słowa `request`, `session`, `application` lub `none`. Zasięg `none` oznacza, że dany obiekt nie jest utrzymywany w żadnej z trzech map zasięgów. Obiekty z tym zasięgiem wykorzystuje się w roli elementów składowych podczas wiązania skomplikowanych komponentów. Przykład takiego rozwiązania przeanalizujemy w punkcie „Wiązanie definicji komponentów” w dalszej części tego podrozdziału.

## Ustawianie wartości właściwości

Nasze rozważania rozpoczniemy od analizy prostego przykładu. Poniżej modyfikujemy egzemplarz komponentu `UserBean`:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>name</property-name>
      <value>ja</value>
   </managed-property>
   <managed-property>
      <property-name>password</property-name>
      <value>sekret</value>
   </managed-property>
</managed-bean>
```

Kiedy komponent `user` jest po raz pierwszy odnajdywany, serwer aplikacji tworzy nowy egzemplarz za pomocą konstruktora domyślnego `UserBean()`. Następnie są wykonywane metody `setName` i `setPassword`.

Aby zainicjalizować właściwość wartością `null`, należy użyć elementu `null-value`. Przykład takiego rozwiązania przedstawiono poniżej:

```
<managed-property>
   <property-name>password</property-name>
   <null-value/>
</managed-property>
```

## Inicjalizacja list i map

Do inicjalizacji wartości typu `List` lub `Map` służy specjalna konstrukcja składniowa. Poniżej przedstawiono przykład inicjalizacji listy:

```
<list-entries>
   <value-class>java.lang.Integer</value-class>
   <value>3</value>
   <value>1</value>
   <value>4</value>
   <value>1</value>
   <value>5</value>
</list-entries>
```

W powyższym fragmencie kodu użyto opakowania w formie typu danych `java.lang.Integer`, ponieważ struktura `List` nie może zawierać wartości typów prostych.

Lista może zawierać wymieszane elementy `value` i `null-value`. Element `value-class` jest opcjonalny — jeśli z niego zrezygnujemy, zostanie wygenerowana lista obiektów klasy `java.lang.String`.

Z nieco bardziej skomplikowaną sytuacją mamy do czynienia w przypadku map. W odpowiednich konstrukcjach możemy stosować opcjonalne elementy `key-class` i `value-class` (gdzie typem domyślnym ponownie jest klasa `java.lang.String`). Musimy następnie zdefiniować sekwencję elementów `map-entry`, z których każdy obejmuje element `key` oraz element `value` lub `null-value`.

Poniżej przedstawiono przykładową konfigurację struktury typu `Map`:

```
<map-entries>
   <key-class>java.lang.Integer</key-class>
   <map-entry>
      <key>1</key>
      <value>George Washington</value>
   </map-entry>
   <map-entry>
      <key>3</key>
      <value>Thomas Jefferson</value>
   </map-entry>
   <map-entry>
      <key>16</key>
      <value>Abraham Lincoln</value>
   </map-entry>
   <map-entry>
      <key>26</key>
      <value>Theodore Roosevelt</value>
   </map-entry>
</map-entries>
```

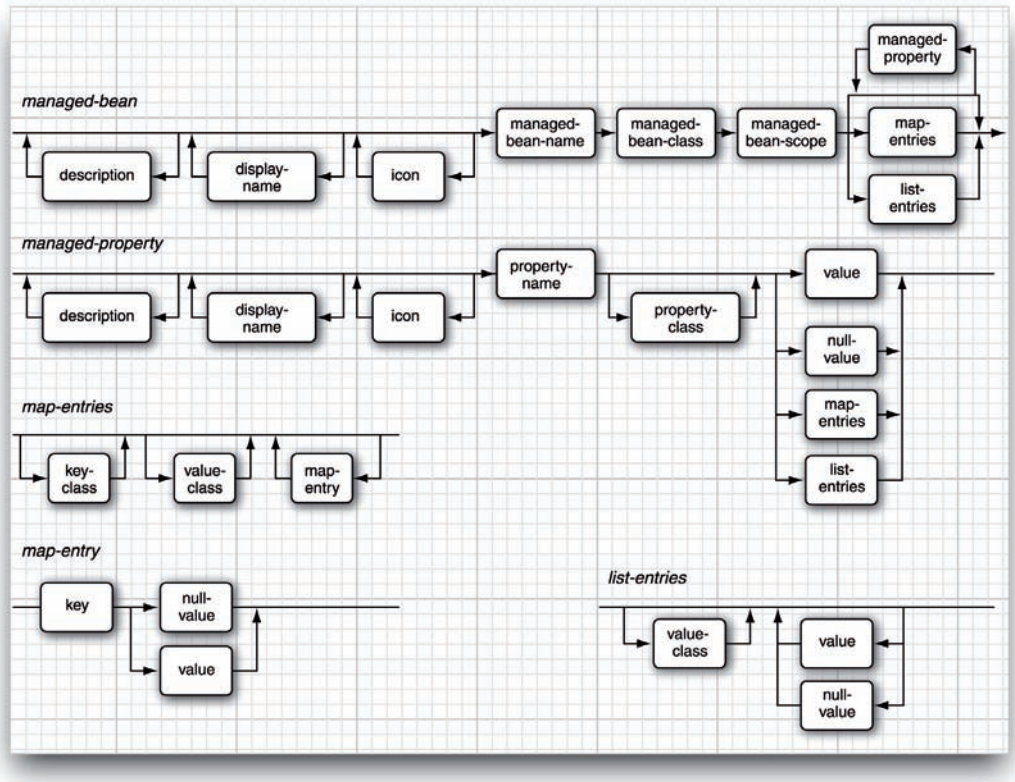
Za pomocą elementów `list-entries` i `map-entries` można inicjalizować struktury `managed-bean` lub `managed-property`, pod warunkiem że odpowiedni komponent lub właściwość jest egzemplarzem typu `List` lub `Map`.

Na rysunku 2.7 przedstawiono *diagram składni definicji* elementu `managed-bean` i wszystkich jego elementów potomnych. Strzałki na tym diagramie dobrze ilustrują, które konstrukcje można stosować w ramach elementu `managed-bean`. Na przykład z drugiego z przedstawionych grafów wynika, że element `managed-property` rozpoczyna się od zera, jednego lub wielu elementów `description`, po których następuje zero, jeden lub wiele elementów `display-name`, zero, jeden lub wiele elementów `icon`, wymagany element `property-name`, opcjonalny element `property-class` oraz dokładnie jeden z elementów `value`, `null-value`, `map-entries` lub `list-entries`.

## Wiązanie definicji komponentów

Bardziej skomplikowane konstrukcje możemy definiować, stosując w elementach `value` wyrażenia reprezentujące wartości, które związą ze sobą więcej komponentów. Wróćmy na chwilę do komponentu quizu użytego w aplikacji `numberquiz`.

Quiz obejmuje zbiór problemów reprezentowanych przez właściwość `problems` dostępną tylko do zapisu. Możemy ten komponent skonfigurować za pomocą następujących elementów:




**Rysunek 2.7.** *Diagram składni definicji elementów komponentu zarządzanego*

```
<managed-bean>
   <managed-bean-name>quiz</managed-bean-name>
   <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>problems</property-name>
      <list-entries>
         <value-class>com.corejsf.ProblemBean</value-class>
         <value>#{problem1}</value>
         <value>#{problem2}</value>
         <value>#{problem3}</value>
         <value>#{problem4}</value>
         <value>#{problem5}</value>
      </list-entries>
   </managed-property>
</managed-bean>
```

Oczywiście musimy teraz zdefiniować komponenty nazwane od `problem1` do `problem5`:

```
<managed-bean>
   <managed-bean-name>problem1</managed-bean-name>
   <managed-bean-class>
      com.corejsf.ProblemBean
   </managed-bean-class>
```

```
    <managed-bean-scope>none</managed-bean-scope>
       <managed-property>
          <property-name>sequence</property-name>
          <list-entries>
             <value-class>java.lang.Integer</value-class>
             <value>3</value>
             <value>1</value>
             <value>4</value>
             <value>1</value>
             <value>5</value>
          </list-entries>
       </managed-property>
       <managed-property>
          <property-name>solution</property-name>
          <value>9</value>
       </managed-property>
</managed-bean>
```

W odpowiedzi na żądanie kierowane do komponentu `quiz` nastąpi automatyczne utworzenie komponentów od `problem1` do `problem5`. Kolejność, w jakiej wskazujemy komponenty zarządzane, nie ma większego znaczenia.

Warto zwrócić uwagę na zasięg `none` komponentów problemów wynikający z tego, że żądania pod ich adresem nigdy nie pochodzą ze strony JSF, a ich egzemplarze są tworzone wskutek żądań wskazujących na komponent `quiz`.

Wiążąc ze sobą różne komponenty, musimy być pewni co do zgodności ich zasięgów. W tabeli 2.1 wymieniono wszystkie dopuszczalne kombinacje.

**Tabela 2.1.** *Zgodne zasięgi komponentów*

| Definiując komponent o takim zasięgu... | ... możemy korzystać z komponentów o takich zasięgach |
|---|---|
| `none` | `none` |
| `application` | `none, application` |
| `session` | `none, application, session` |
| `request` | `none, application, session, request` |

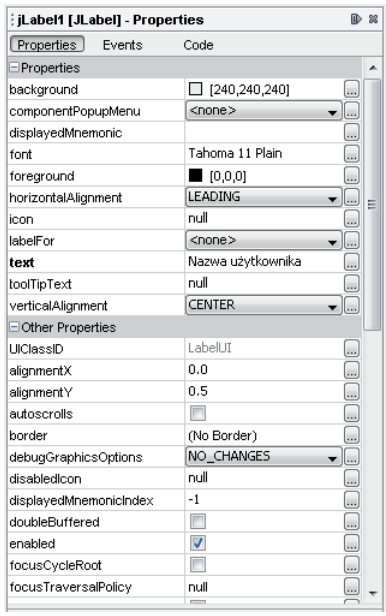
# Konwersje łańcuchów

Wartości właściwości i elementy list lub map konfigurujemy za pomocą elementu `value` obejmującego łańcuch. Łańcuch zawarty w tym elemencie musi zostać przekonwertowany na odpowiedni typ właściwości lub elementu. W przypadku typów prostych taka konwersja jest bardzo prosta. Możemy na przykład zdefiniować wartość typu `boolean`, stosując łańcuch `true` lub `false`.

Począwszy od wersji JSF 1.2, możemy wykorzystywać elementy `value` także do definiowania wartości typów wyliczeniowych. Niezbędna konwersja jest realizowana za pomocą metody `Enum.valueOf(`*`klasa_właściwości, wartość_tekstowa`*`)`.

Dla pozostałych typów właściwości implementacja JSF próbuje zlokalizować pasującą implementację interfejsu `PropertyEditor`. Jeśli taki edytor właściwości istnieje, następuje wywołanie metody `setAsText` tego edytora celem konwersji łańcuchów na wartości właściwości. Edytory właściwości są bardzo często wykorzystywane przez komponenty pracujące po stronie klienta i mają na celu konwersję wartości właściwości na reprezentację tekstową lub graficzną umożliwiającą ich wyświetlanie w oknie właściwości (patrz rysunek 2.8).

**Rysunek 2.8.** *Okienko właściwości narzędzia do projektowania graficznego interfejsu użytkownika*




Definiowanie edytora właściwości jest zadaniem dość złożonym, zatem zainteresowanych Czytelników zachęcamy do lektury rozdziału 8. książki pt. *Core Java™ 2, vol. 2 — Advanced Features (7th ed.)*[4] autorstwa Caya Horstmanna i Gary'ego Cornella.

Musimy pamiętać, że reguły konwersji są dość restrykcyjne. Jeśli na przykład dysponujemy właściwością typu `URL`, nie możemy po prostu zapisać adresu URL w formie łańcucha, mimo że istnieje konstruktor `URL(String)`. Musimy albo opracować edytor właściwości dla typu `URL`, albo ponownie zdefiniować typ tej właściwości jako `String`.

Reguły konwersji podsumowano w tabeli 2.2. Identyczne reguły są stosowane dla akcji `jsp:setProperty` specyfikacji JSP.

# Składnia wyrażeń reprezentujących wartości

W tym podrozdziale szczegółowo omówimy składnię wyrażeń reprezentujących wartości. Można niniejszy materiał traktować jak swoisty leksykon. Czytelnicy, którzy mają tę książkę w rękach po raz pierwszy, mogą ten podrozdział pominąć i wrócić do niego w przyszłości.

[4] Polskie wydanie: *Java 2. Techniki zaawansowane. Wydanie II*, Helion, 2005 — *przyp. tłum.*

**Tabela 2.2.** *Konwersje łańcuchów*

| Typ docelowy | Konwersja |
|---|---|
| `int`, `byte`, `short`, `long`, `float`, `double` lub odpowiedni typ opakowania | Metoda `valueOf` typu opakowania lub `0`, jeśli dany łańcuch jest pusty. |
| `boolean` lub `Boolean` | Wynik metody `Boolean.valueOf` lub `false`, jeśli dany łańcuch jest pusty. |
| `char` lub `Character` | Pierwszy znak danego łańcucha lub `(char)0`, jeśli ten łańcuch jest pusty. |
| `String` lub `Object` | Kopia danego łańcucha lub `new String("")`, jeśli ten łańcuch jest pusty. |
| właściwość komponentu | Typ, który wywołał metodę `setAsText` edytora właściwości (jeśli taki edytor istnieje). Jeśli edytor właściwości nie istnieje lub generuje wyjątek, właściwość ma przypisywaną wartość `null` (w przypadku łańcucha pustego) lub jest generowany błąd. |

W pierwszej kolejności skoncentrujemy się na wyrażeniach w postaci `a.b`. Przyjmijmy, że wiemy, jaki obiekt jest reprezentowany przez zmienną `a`. Jeśli `a` jest tablicą, listą lub mapą, musimy stosować pewne reguły specjalne (patrz punkt „Stosowanie nawiasów kwadratowych" w dalszej części tego podrozdziału). Jeśli `a` jest dowolnym innym obiektem, `b` musi być nazwą właściwości tego obiektu. Dokładne znaczenie wyrażenia `a.b` zależy od tego, czy dane wyrażenie jest wykorzystywane w trybie **r-wartości** (ang. *r-value*) czy **l-wartości** (ang. *l-value*).

Przytoczona terminologia jest stosowana w teorii języków programowania w kontekście wyrażeń **po prawej stronie** operatora przypisania, traktowanych inaczej niż wyrażenia **po lewej stronie** tego operatora.

Przeanalizujmy następującą operację przypisania:

```
left = right;
```

Kompilator generuje inny kod dla wyrażenia `left` i inny dla wyrażenia `right`. Wyrażenie `right` zostanie przetworzone w trybie r-wartości, a wynikiem tego przetworzenia będzie pojedyncza wartość. Wyrażenie `left` zostanie przetworzone w trybie l-wartości, co doprowadzi do zapisania jakiejś wartości w pewnym miejscu.

Ten sam model jest stosowany w przypadku wyrażeń reprezentujących wartości stosowanych w komponentach interfejsu użytkownika:

```
<h:inputText value="#{user.name}"/>
```

W czasie wizualizowania tego pola tekstowego wyrażenie `user.name` jest przetwarzane w trybie r-wartości, co wiąże się z koniecznością wywołania metody `getName`. Na etapie dekodowania to samo wyrażenie jest przetwarzane w trybie l-wartości, zatem jest wywoływana metoda `setName`.

Ogólnie wyrażenie `a.b` jest przetwarzane w trybie r-wartości przez wywołanie metody zwracającej wartość właściwości oraz w trybie l-wartości przez wywołanie metody ustawiającej wartość właściwości.

## Stosowanie nawiasów kwadratowych

Podobnie jak w języku JavaScript zamiast notacji kropki możemy stosować nawiasy kwadratowe. Oznacza to, że wszystkie trzy przedstawione poniżej wyrażenia reprezentują dokładnie to samo:

```
a.b
a["b"]
a['b']
```

Na przykład wyrażenia `user.password`, `user["password"]` oraz `user['password']` są sobie równoważne.

Po co ktokolwiek miałby stosować wyrażenie `user["password"]`, skoro dużo łatwiejsze do zapisania jest wyrażenie `user.password`? Z kilku powodów:

- Kiedy uzyskujemy dostęp do tablicy lub mapy, notacja `[]` jest bardziej intuicyjna.
- Notację `[]` można stosować łącznie z łańcuchami zawierającymi kropki i myślniki, np. `msgs["error.password"]`.
- Notacja `[]` umożliwia nam dynamiczne wyznaczanie nazwy interesującej nas właściwości, np. `a[b.propname]`.

W wyrażeniach reprezentujących wartości należy stosować apostrofy, jeśli w roli separatora atrybutów wykorzystujemy cudzysłowy: `value="#{user['password']}"`. Apostrofy i cudzysłowy można stosować zamiennie: `value='#{user["password"]}'`.

## Wyrażenia odwołujące się do map i list

Język wyrażeń reprezentujących wartość nie ogranicza się do odwołań do właściwości komponentów. Przykładowo: niech `m` będzie obiektem dowolnej klasy implementującej interfejs `Map`. Wyrażenie `m["key"]` (lub wyrażenie równoważne w postaci `m.key`) jest wówczas łączem do odpowiedniej wartości. W trybie r-wartości nastąpi wywołanie metody:

```
m.get("key")
```

W trybie l-wartości zostanie wykonane następujące wyrażenie:

```
m.put("key", right);
```

W tym przypadku `right` reprezentuje wartość prawej strony, która ma zostać przypisana elementowi `m.key`.

Możemy też uzyskiwać dostęp do wartości dowolnego obiektu klasy implementującej interfejs `List` (np. klasy `ArrayList`). Pozycja na liście jest reprezentowana przez indeks całkowitoliczbowy. Na przykład wyrażenie `a[i]` (lub `a.i`) wiąże `i`-ty element listy `a`. Zmienna `i` musi mieć albo postać liczby całkowitej, albo łańcucha, który można na taką liczbę przekonwertować. Ta sama reguła znajduje zastosowanie w przypadku typów tablicowych. Jak zawsze wartości indeksów rozpoczynają się od zera.

Opisane powyżej reguły przetwarzania podsumowano w tabeli 2.3.

**Tabela 2.3.** *Przetwarzanie wyrażenia reprezentującego wartość w postaci a.b*

| Typ zmiennej a | Typ zmiennej b | Tryb l-wartości | Tryb r-wartości |
|---|---|---|---|
| `null` | dowolny | błąd | `null` |
| dowolny | `null` | błąd | `null` |
| `Map` | dowolny | `a.put(b, right)` | `a.get(b)` |
| `List` | możliwy do przekonwertowania na typ `int` | `a.set(b, right)` | `a.get(b)` |
| tablica | możliwy do przekonwertowania na typ `int` | `a[b] = right` | `a[b]` |
| komponent | dowolny | Wywołanie metody ustawiającej wartość właściwości nazwanej `b.toString()`. | Wywołanie metody zwracającej wartość właściwości nazwanej `b.toString()`. |

Okazuje się niestety, że wyrażenie reprezentujące wartość nie zdaje egzaminu w przypadku właściwości indeksowanych. Jeśli `p` jest właściwością indeksowaną komponentu `b`, a `i` jest liczbą całkowitą, wówczas wyrażenie `b.p[i]` nie uzyskuje dostępu do `i`-tej wartości tej właściwości. Wyrażenie w tej formie zostanie zakwalifikowane jako błędne składniowo. Brak obsługi tego rodzaju wyrażeń technologia JSF odziedziczyła po technologii JSP.

## Rozwiązywanie wyrazu początkowego

Wiemy już, jak wygląda proces przetwarzania wyrażeń w postaci `a.b`. Okazuje się, że te same reguły mogą być wielokrotnie stosowane dla wyrażeń w formie `a.b.c.d` (lub, co oczywiste, `a['b'].c["d"]`). Musimy jednak przeanalizować faktyczne znaczenie wyrazu początkowego `a`.

W prezentowanych do tej pory przykładach wyraz początkowy odwoływał się albo do komponentu skonfigurowanego w pliku *faces-config.xml*, albo do mapy pakietu komunikatów. Chociaż te dwa rozwiązania są stosowane zdecydowanie najczęściej, wyraz początkowy może reprezentować także inne konstrukcje.

Istnieje wiele predefiniowanych obiektów. Kompletną listę przedstawiono w tabeli 2.4; poniżej przedstawiono wybrany przykład wyrażenia:

```
header['User-Agent']
```

które reprezentuje wartość parametru `User-Agent` żądania HTTP, wartość tego parametru identyfikuje przeglądarkę internetową użytkownika.

**Tabela 2.4.** *Obiekty predefiniowane języka wyrażeń reprezentujących wartości*

| Nazwa zmiennej | Znaczenie |
|---|---|
| `header` | Struktura typu `Map` reprezentująca parametry nagłówka protokołu HTTP (obejmująca tylko po pierwszej wartości dla każdej nazwy). |
| `headerValues` | Struktura typu `Map` reprezentująca parametry nagłówka protokołu HTTP w formie tablicy typu `String[]` obejmującej wszystkie wartości dla danej nazwy. |
| `Param` | Struktura typu `Map` reprezentująca parametry żądania protokołu HTTP (obejmująca tylko po pierwszej wartości dla każdej nazwy). |
| `paramValues` | Struktura typu `Map` reprezentująca parametry żądania protokołu HTTP w formie tablicy typu `String[]` obejmującej wszystkie wartości dla danej nazwy. |
| `cookie` | Struktura typu `Map` reprezentująca nazwy i wartości znaczników kontekstu klienta dla bieżącego żądania. |
| `initParam` | Struktura typu `Map` reprezentująca parametry inicjalizacji danej aplikacji internetowej. Parametry inicjalizacji zostaną omówione w rozdziale 10. |
| `requestScope` | Struktura typu `Map` reprezentująca wszystkie atrybuty obejmujące zasięgiem żądanie. |
| `sessionScope` | Struktura typu `Map` reprezentująca wszystkie atrybuty obejmujące zasięgiem sesję. |
| `applicationScope` | Struktura typu `Map` reprezentująca wszystkie atrybuty obejmujące zasięgiem aplikację. |
| `facesContext` | Egzemplarz klasy `FacesContext` właściwy dla danego żądania. Klasa `FacesContext` zostanie omówiona w rozdziale 6. |
| `view` | Egzemplarz klasy `UIViewRoot` właściwy dla danego żądania. Klasa `UIViewRoot` zostanie omówiona w rozdziale 7. |

Jeśli wyraz początkowy nie jest żadnym z wymienionych obiektów predefiniowanych, implementacja JSF poszukuje tego wyrazu kolejno w **zasięgu żądania**, **zasięgu sesji** i **zasięgu aplikacji**. Wymienione zasięgi mają postać obiektów map zarządzanych przez kontener serwletów. Kiedy więc definiujemy na przykład komponent zarządzany, jego nazwa i wartość są dodawane do mapy odpowiedniego zasięgu.

Dopiero kiedy okaże się, że użytej nazwy nie można odnaleźć w tych zasięgach, następuje jej przekazanie do implementacji `VariableResolver` właściwej dla danej aplikacji JSF. Działanie domyślnego mechanizmu interpretacji zmiennych sprowadza się do przeszukania elementów `managed-bean` w zasobach konfiguracyjnych (czyli z reguły w pliku *faces-config.xml*).

Przeanalizujmy następujące wyrażenie przykładowe:

```
#{user.password}
```

Wyraz `user` nie jest żadnym z opisanych powyżej obiektów predefiniowanych. Kiedy po raz pierwszy zostanie odkryty w kodzie źródłowym, okaże się, że nie jest nazwą atrybutu w zasięgu żądania, sesji ani aplikacji.

W tej sytuacji mechanizm interpretacji zmiennych przetwarza następujący wpis znaleziony w pliku konfiguracyjnym *faces-config.xml*:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Mechanizm odpowiedzialny za interpretację zmiennych wywołuje konstruktor domyślny klasy `com.corejsf.UserBean`, po czym umieszcza nowo utworzony związek w mapie `sessionScope`. Ostatecznie mechanizm interpretujący zwraca ten obiekt jako wynik operacji przeszukiwania.

Jeśli wyraz `user` będzie wymagał ponownej interpretacji w ramach tej samej sesji, gotowe rozwiązanie zostanie zlokalizowane w zasięgu sesji (reprezentowanym przez mapę `sessionScope`).

## Wyrażenia złożone

W ramach wyrażeń reprezentujących wartości możemy stosować ograniczony zbiór operatorów:

- Operatory arytmetyczne: `+`, `-`, `*`, `/` oraz `%`. Dla ostatnich dwóch operatorów istnieją wersje alfabetyczne, odpowiednio `div` i `mod`.
- Operatory relacyjne: `<`, `<=`, `>`, `>=`, `==` i `!=`, a także ich wersje alfabetyczne: `lt`, `le`, `gt`, `ge`, `eq` oraz `ne`. Cztery pierwsze odpowiedniki alfabetyczne są niezbędne w sytuacji, gdy chcemy zagwarantować bezpieczne przetwarzanie danych w formacie XML.
- Operatory logiczne: `&&`, `||` i `!` oraz ich wersje alfabetyczne: `and`, `or` i `not`. Stosowanie pierwszego odpowiednika alfabetycznego jest warunkiem prawidłowego przetwarzania danych w formacie XML.
- Operator `empty`. Wyrażenie `empty a` jest prawdziwe (ma wartość `true`), jeśli `a` ma wartość `null`, jest tablicą lub łańcuchem zerowej długości bądź pustą kolekcją lub mapą (odpowiednio typu `Collection` lub `Map`).
- Trójargumentowy operator selekcji `?:`.

Dla wymienionych operatorów stosuje się te same reguły poprzedzania, które znamy z Javy. Operator `empty` charakteryzuje się identycznym priorytetem jak operatory jednoargumentowe `-` oraz `!`.

Ogólnie chcemy, aby możliwie niewielka część obliczeń związanych z przetwarzaniem wyrażeń była realizowana na naszych stronach internetowych, ponieważ takie rozwiązanie naruszałoby zasadę rozdziału warstwy prezentacji od logiki biznesowej. Okazuje się jednak, że w pewnych rzadkich sytuacjach możliwość stosowania operatorów w warstwie prezentacji jest warta rozważenia. Przypuśćmy na przykład, że chcemy ukrywać jakiś komponent interfejsu graficznego, gdy właściwość `hide` pewnego komponentu aplikacji ma wartość `true`. Aby ukryć komponent interfejsu, należy przypisać wartość `false` jego atrybutowi `rendered`. Odwrócenie tej wartości wymaga użycia operatora `!` (lub `not`):

```
<h:inputText rendered="#{!bean.hide}" ... />
```

I wreszcie możemy konkatenować zwykłe łańcuchy i wyrażenia reprezentujące wartość przez umieszczanie ich w bezpośrednim sąsiedztwie. Przeanalizujmy poniższy przykład:

```
<h:outputText value="#{messages.greeting}, #{user.name}!"/>
```

Powyższe wyrażenie konkatenuje cztery łańcuchy: łańcuch zwrócony przez wyrażenie `#{messages. greeting}`, łańcuch złożony z przecinka i spacji, łańcuch zwrócony przez wyrażenie `#{user.name}` oraz łańcuch złożony z samego wykrzyknika.

Zaprezentowaliśmy wszystkie reguły stosowane w procesie przetwarzania wyrażeń reprezentujących wartości. Oczywiście w praktyce zdecydowana większość wyrażeń ma postać `#{komponent.właściwość}`. Czytelnicy, którzy staną przed koniecznością obsługi bardziej złożonych wyrażeń, będą mogli w każdej chwili wrócić do tego materiału.

## Wyrażenia odwołujące się do metod

**Wyrażenie odwołujące się do metody** obejmuje obiekt wraz z metodą, którą można dla tego obiektu zastosować.

Typowy przykład użycia takiego wyrażenia przedstawiono poniżej:

```
<h:commandButton action="#{user.checkPassword}"/>
```

Zakładamy, że `user` jest wartością typu (obiektem klasy) `UserBean`, natomiast `checkPassword` jest metodą tej klasy. Wyrażenie odwołujące się do metody jest wygodnym sposobem opisania wywołania, które wymaga realizacji w niedalekiej przyszłości.

W czasie przetwarzania takiego wyrażenia odpowiednia metoda jest wykonywana na wskazanym obiekcie.

W naszym przypadku kliknięcie komponentu przycisku spowoduje wywołanie metody `user.checkPassword()` i przekazanie zwróconego łańcucha do mechanizmu odpowiedzialnego za obsługę nawigacji.

Reguły składniowe dla tego rodzaju wyrażeń bardzo przypominają te, które stosuje się dla wyrażeń reprezentujących wartości. Wszystkie składowe poza ostatnią służą do identyfikacji obiektu będącego przedmiotem wywołania metody. Ostatnią składową musi być nazwa metody, którą można zastosować dla obiektu wskazywanego przez poprzednie składowe.

Wyrażenie odwołujące się do metody można przypisać czterem atrybutom komponentów:

- `action` (patrz podrozdział „Nawigacja dynamiczna” w rozdziale 3.);
- `validator` (patrz punkt „Weryfikacja za pomocą metod komponentów” w rozdziale 6.);
- `valueChangeListener` (patrz podrozdział „Zdarzenia zmiany wartości” w rozdziale 7.);
- `actionListener` (patrz podrozdział „Zdarzenia akcji” w rozdziale 7.).

Typy parametrów i wartości zwracanej przez daną metodę zależą od kontekstu, w którym stosujemy nasze wyrażenie. Przykładowo atrybut `action` można związać z metodą bezparametrową zwracającą wartość typu `String`, natomiast atrybut `actionListener` można związać z metodą otrzymującą pojedynczy parametr typu `ActionEvent` i nie zwracającą wartości. Za dostarczanie wartości parametrów i przetwarzanie wartości zwracanych odpowiada oczywiście kod wywołujący wyrażenie odwołujące się do metody.