Technologie budowy aplikacji pracujących po stronie serwera WWW

Common Gateway Interface (CGI)

- CGI jest interfejsem służącym do uruchamiania zewnętrznych programów po stronie serwera WWW; aplikacje CGI mogą być tworzone w C, Pascal, Fortran, Perl, językach skryptowych, itd.
- Jednym z najpopularniejszych zastosowań CGI jest umożliwianie bieżącego dostępu do baz danych
- Serwer WWW wykonuje aplikację CGI i przesyła wynik jej pracy do klienta WWW
- CGI definiuje interfejs pomiędzy serwerami WWW a aplikacjami CGI, obejmujący parametry wywołania, wejście i wyjście

Adresy URL dla aplikacji CGI

- Przykładowy URL dla bezparametrowej aplikacji CGI:
  - http://www.foo.org/cgi-bin/foo.cgi
- URL przekazuje parametry dla aplikacji CGI poprzez dołączenie znaku zapytania, a za nim listy argumentów i wartości (rozdzielanych "&"):
  - http://www.foo.org/cgi-bin/foo.cgi?a=1
  - http://www.foo.org/cgi-bin/foo.cgi?a=1&b=5&c=3
- Parametry wywołania są dostępne poprzez zmienne środowiskowe lub standardowe wejście

Zmienne środowiskowe CGI

- Serwer WWW przekazuje aplikacji CGI dane przy pomocy zmiennych środowiskowych; w języku C, mogą one być odczytywane przez funkcję getenv()
- Zmienne środowiskowe CGI:
  - PATH_INFO - ścieżka URL do aplikacji CGI
  - QUERY_STRING - argumenty wejściowe aplikacji CGI
  - CONTENT_LENGTH – rozmiar argumentów wejściowych
  - REMOTE_ADDR - adres IP użytkownika wysyłającego żądanie
  - REMOTE_HOST - adres domenowy użytkownika wysyłającego żądanie
  - REMOTE_USER - nazwa użytkownika wysyłającego żądanie
  - SERVER_PORT - numer portu, do którego przybyło żądanie
- Aplikacja CGI zapisuje wyniki na stdout (standard. wyjście)
- Pierwszy wiersz wyniku HTML powinien brzmieć: "Content-type: text/html"

Przykład prostej aplikacji CGI

```
#!/bin/sh

# Get current time and date
DATE=`/bin/date + '%B %d, %Y'`
TIME=`/bin/date + '%T'`

#Send the data to the client
echo "Content-type: text/html"
echo "<HTML><HEAD></HEAD><BODY>"
echo "Current date: $DATE"
echo "Current time: $TIME"
echo "</BODY></HTML>
exit 0
```

Język Perl

- **Perl** (Practical Extraction and Report Language) jest językiem interpretowanym, podobnym do *C* z domieszkami *sed*, *awk, sh*, *csh*, *Pascal*, *FORTRAN*, *BASIC-PLUS*
- Wysoce zoptymalizowany dla przetwarzania tekstów, ale też odpowiedni dla danych binarnych
- Typy danych: skalary ($), tablice indeksowane (listy) (@), tablice asocjacyjne (%)
  - $foo = 3.14159;
  - $foo = 'red';
  - @foo = (1, 3, 5);
  - %frogs = ('green' => 3, 'blue' => 6, 'yellow' => 9);

Operatory języka Perl

- Potęgowanie: ******, ****=**
- Operator zakresowy: **..**
  - for $i (60..75) {do foo($i);}
- Konkatenacja tekstów: **.**, **.=**
  - $x = $y . $z
- Powtórzenie tekstu: **x**, **x=**
  - $bar = '-' x 72;
- Porównania tekstów: **eq**, **ne**, **lt**, **gt**, **le**, **ge**
  - if ($x eq 'foo') {}
- Operacje na plikach: **-e**, **-z**, **-O**, **-T**, itd.
  - if (-e $file) {}

Sterowanie przepływem w języku Perl

if (*expr*) *block* **else** *block*
if (*expr*) *block* **elsif** (*expr*) *block* **else** *block*
while (*expr*) *block*
do *block* **while** *expr*
for (*expr; expr; expr*) *block*
foreach *$var* (*list*) *block*

unless i **until** są zanegowanymi **if** i **while**

Zmienne proste - przykład

```
print "Using a variable before initializing: $var\n";

$test = $num + 5;
print "Adding uninitialized variable num to 5 yields: $test.\n";

$a = 5;
print "The value of variable a is: $a\n";

$a = $a + 5;
print "Variable a after adding 5 is $a.\n";

$b = "A string value";
$a = $a + $b;

print "Adding a string to an integer yields: $a\n";

$number = 7;
$b = $b + $number;

print "Adding an integer to a string yields: $b\n";
```

Using a variable before initializing:
Adding uninitialized variable num to 5 yields: 5.
The value of variable a is: 5
Variable a after adding 5 is 10.
Adding a string to an integer yields: 10
Adding an integer to a string yields: 7

Zmienne o zasięgu lokalnym (ograniczonym do bloku) inicjalizowane są przy użyciu słowa kluczowego „my”, np.:
`my $y = 20;`

Tablice w Perlu: przykład

```
#!/usr/bin/perl
# Print out today's yearday on stdout
($sec, $min, $hour, $day, $mon, $year, @rest) = gmtime();
if ($year == 2000) { #a leap year
  @months = (31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
} else {
  @months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
}
$yearday = 0;
for ($m=0; $m<$mon; $m++) {
  $yearday += @months[$m];
}
$yearday += $day;
print $yearday . "\n";
```

Wyrażenia regularne

- Porównania tekstowe
  - $string =~ /sought_text/;
    - operator "match"
- Wycinanie tekstów
  - $string =~ m/whatever(sought_text)whatever2/;
  - $soughtText = $1;
- Zastępowanie tekstów
  - $string =~ tr/originaltext/newtext/;
    - operator "translate"
  - $string =~ s/originaltext/newtext/;
    - operator "substitute"

Wyrażenia regularne – przykłady

- `if ($mytext =~ /word/)` – sprawdza, czy zmienna zawiera podane słowo
- `if ($mytext =~ /word/i)` – sprawdza, czy zmienna zawiera podane słowo (/i – ignoruj wielkość znaków)
- `if ($mytext =~ /^word/)` – sprawdza, czy zmienna zawiera podane słowo na początku wiersza
- `if ($mytext =~ /word$/)` – sprawdza, czy zmienna zawiera podane słowo na końcu wiersza
- `if ($mytext =~ / \b ( \w+ es ) \b /x ) {print "$1";}` – znajduje słowo rozpoczynające się od liter es (\b – granica słowa, \w+ - jedna lub wiele liter, /x ignorowanie spacji w wyrażeniu regularnym, nawiasy powodują podstawienie znalezionego słowa do $1)
- `if ($phone =~ / \( \d{3} \) \d{3} - \d{3} /x)` – sprawdza, czy zmienna zawiera poprawny numer telefonu

Znaki specjalne

- . dowolny znak
- \w znak alfanumeryczny lub „_”
- \W znak inny niż alfanumeryczny
- \s znak spacji
- \S znak inny niż spacja
- \d cyfra
- \D znak inny niż cyfra
- \t znak tabulacji
- \n znak nowego wiersza

Znaki powtórzenia

- * 0 lub wiele razy
- + 1 lub wiele razy
- ? 1 lub 0 razy
- {n} dokładnie n razy
- {n,} co najmniej n razy
- {n,m} co najmniej n, ale nie więcej niż m razy

Przykład:

`$string =~ /\s*rem/i` - prawda, jeżeli pierwszy czytelny tekst to rem lub REM

Korzystanie z grup znaków ()

```
if($string =~ /(B|b)ill (C|c)linton/)
{print "It is Clinton, all right!\n"}

if($string =~ /(A|E|I|O|U|Y|a|e|i|o|u|y]/)
{print "String contains a vowel!\n"}

if($string =~ /^(Clinton|Bush|Reagan)/i)
{print "$string\n"};
```

Korzystanie z klas znaków []

- Klasy znaków posiadają trzy główne zalety:
  - Skrócona notacja, np. [AEIOUY] zamiast (A|E|I|O|U|Y)
  - Zakresy, np. [A-Z]
  - Maping 1:1 pomiędzy klasami, np. tr/[a-z]/[A-Z]
- Przykład:
  - Wyświetl wszystkie osoby o nazwiskach rozpoczynających się od A, B, C, D, lub E

```
if($string =~ m/^[A-E]/)   {print "$string\n"}
```

Zastąpienia

Podstawienia (poziom słowa)

- Zamień każde "Bill Clinton" na "Al Gore"

```
$string =~ s/Bill Clinton/Al Gore/;
```

- Jak wyżej, lecz ignorując wielkość znaków (np. bIlL ClInToN)

```
$string =~ s/Bill Clinton/Al Gore/i;
```

Translacje (poziom znaku)

- Zamień wszystkie samogłoski na wielkie litery

```
$string =~ tr/[a,e,i,o,u,y]/[A,E,I,O,U,Y]/;
```

- Zamień wszystko na małe litery

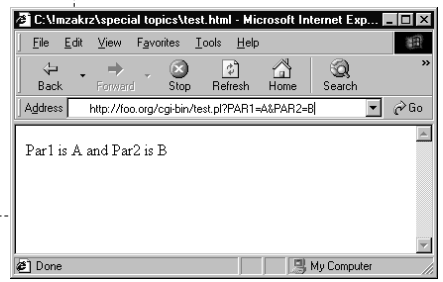
```
$string =~ tr/[A-Z]/[a-z]/;
```

Przykład aplikacji WWW (1)

```
#!/usr/bin/perl
use CGI qw/:standard/;

print header;
open(COUNTREAD, "counter.dat");
  $data = <COUNTREAD>;
  $data++;
close(COUNTREAD);

open(COUNTWRITE, ">counter.dat");
  print COUNTWRITE $data;
close(COUNTWRITE);

print "<CENTER>";
print "<STRONG>You are visitor number</STRONG><BR>";

for ($count = 0; $count < length($data); $count++ )
{
  $number = substr( $data, $count, 1 ) ;
  print "<IMG SRC=\"images/counter/$number.jpg\">";
}
print "</CENTER>";
```

Przykład aplikacji WWW Example (2) Dostęp do zmiennych CGI

```
#!/usr/local_old/bin/perl

use CGI qw/:standard/;

$par1 = param(PAR1);
$par2 = param(PAR2);

print header;

print "Par1 is $par1 and Par2 is $par2\n";
```



Dostęp do baz danych - przykład

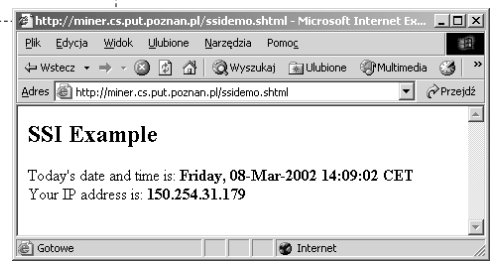
```
use DBI;

my $dbh = DBI->connect('DBI:mysql:scott');
my $sth = $dbh->prepare(
   'SELECT etat, id_prac FROM pracownicy WHERE nazwisko = ?');
my @data;

$sth->execute('KOWALSKI');

while (@data = $sth->fetchrow_array()) {
         my $etat = $data[0];
         my $id = $data[1];
         print "\t$id: $etat $lastname\n";
       }
$sth->finish;
}
$dbh->disconnect;
```

Server Side Includes (SSI)

- Dokumenty HTML, zapisane w systemie plików serwera WWW, mogą zawierać proste komendy, które będą interpretowane przez serwer podczas przekazywania dokumentu do użytkownika
- Komendy SSI zapisywane są w postaci komentarzy HTML, zawierających następujące słowa kluczowe:
  - config (sterowanie przetwarzaniem pliku)
  - include (włączenie zawartości innego dokumentu)
  - echo (wyświetlenie wartości zmiennej środowiskowej SSI)
  - fsize (wyświetlenie rozmiaru podanego pliku)
  - flastmod (wyświetlenie daty ostatniej modyfikacji podanego pliku)
  - exec (wykonanie polecenia systemu operacyjnego lub aplikacji CGI)

SSI - przykład

```
<h2>SSI Example</h2>
Today's date and time is:<b>
<!--#echo var="DATE_LOCAL" -->
</b><br>
Your IP address is:<b>
<!--#echo var="REMOTE_ADDR" -->
</b><br>
```

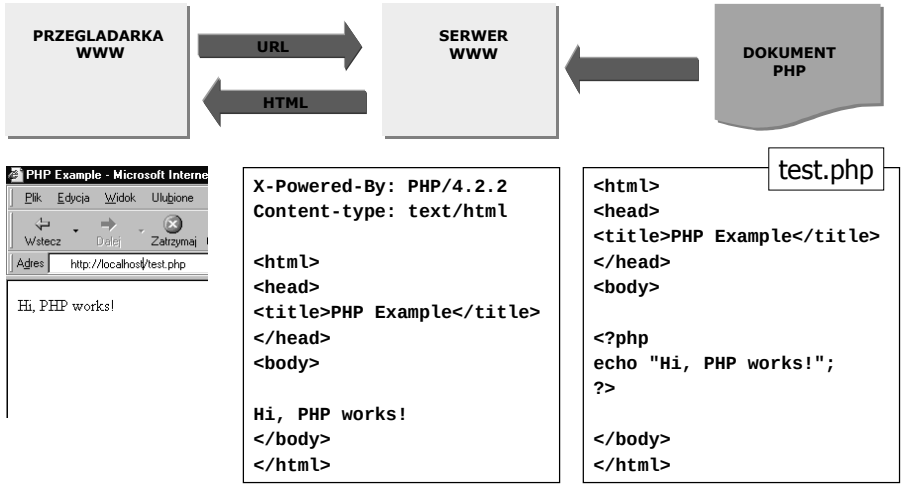


Wprowadzenie do PHP

Czym jest PHP?

- PHP: Hypertext Preprocessor
- Język skryptowy ogólnego przeznaczenia
  - opracowany z myślą o aplikacjach WWW
  - najczęściej zagnieżdżany w HTML
  - zagnieżdżone skrypty uruchamiane po stronie serwera
- Rozwijany na zasadach Open Source
- Szczególnie popularny w środowisku Linux, ale dostępny również na wielu innych platformach np. Microsoft Windows
- Składnia PHP przypomina C, pewne elementy przejęte z Perla

Architektura PHP



Dlaczego PHP?

- PHP jest bardzo łatwy do opanowania
  - szczególnie dla programistów znających C, C++ lub Java
- PHP obsługuje ok. 20 najpopularniejszych SZBD ("native support") oraz standard ODBC
- PHP jest szybki i oszczędza zasoby serwera
- PHP jest dostępny dla wielu systemów operacyjnych i serwerów WWW
- PHP jest rozwijany na zasadach Open Source
  - darmowy
  - rozszerzalny (możliwość dodawania własnych modułów)
  - błędy szybko wykrywane i poprawiane
  - nowe funkcje dodawane zgodnie z wymaganiami rynku
  - łatwo o darmowe wsparcie techniczne ze strony "PHP community"

Historia PHP

- PHP/FI – Rasmus Lerdorf, 1995
  - Personal Home Page/Forms Interpreter (skrypty w Perlu, potem C)
  - stworzony do śledzenia odwiedzin strony domowej
- PHP 2.0 – Rasmus Lerdorf, 1997
  - interpretacja kodu "linia po linii"
- PHP 3.0 – Gutmans & Suraski, 1997
  - interpretacja kodu, ale parser analizujący cały skrypt
  - szybkość działania dla prostych, krótkich skryptów
  - olbrzymia popularność
  - kolejny raz złożoność faktycznych zastosowań przerosła plany twórców
- PHP 4.0 – Zend Technologies, 2000
  - zupełnie nowy "engine" (Zend engine)
  - prawie pełna zgodność z PHP3
  - najpierw kompilacja skryptu (kod pośredni), później uruchomienie
  - wzrost efektywności dla dużych, złożonych skryptów

Obszary zastosowań PHP

- Skrypty server-side na potrzeby aplikacji WWW
  - podstawowe zastosowanie PHP (do tego celu stworzony)
  - praca jako moduł serwera WWW lub w trybie CGI
- Skrypty uruchamiane z linii poleceń
- Aplikacje klienckie z interfejsem użytkownika
  - prawdopodobnie nie najlepszy język do tego celu
  - wymagane rozszerzenie PHP-GTK

Obsługiwane bazy danych

- "Native support":
  - IBM DB2, Informix, MS SQL Server, Oracle, Sybase
  - MySQL, PostgreSQL
  - Adabas D, dBase, Empres, FilePro, Hyperwave, Ingres, InterBase, FrontBase, mSQL, Direct MS-SQL, Ovrimos, Solid, Velocis, Unix dbm
- interfejs ODBC
- abstrakcyjne rozszerzenie dbx
  - pozwala na "przezroczyste" używanie dowolnej bazy obsługiwanej przez to rozszerzenie
  - w tej chwili obsługuje: FrontBase, MS SQL Server, MySQL, ODBC, PostgreSQL, Sybase-CT

Zagnieżdżanie PHP w HTML

- Zawsze dostępne

  1) <?php echo("Zawsze działa\n"); ?>

  2) <script language="php">
     echo ("Dobre dla niektórych edytorów HTML.\n");
     </script>
- Wymagające uaktywnienia w php.ini

  3) <? echo("short_open_tag = on. Kolizja z XML.\n"); ?>
     <?= wyrażenie ?>

  4) <% echo("asp_tags = on.\n"); %>
     <%= $zmienna %>

Przeplatanie PHP i HTML

```
<?php
       if ($wyrażenie) {
?>
         <b>Wyrażenie jest prawdziwe.</b>
<?php
       } else {
?>
         <b> Wyrażenie jest fałszywe.</b>
<?php
       }
?>
```

- Tekst pomiędzy **?>** i **<?php** jest traktowany jak argument **echo**

Oddzielanie instrukcji, komentarze

```
<?php
  echo "Jedna instrukcja";
?>

<?php
  echo "Pierwsza instrukcja"; echo "Druga instrukcja"
?>

<?php
  echo "Instrukcja"; // Jednoliniowy
  echo "Instrukcja"; # Jednoliniowy
  /*
     Komentarz
     wieloliniowy
  */
?>
```

Typy danych

- Typy skalarne (4)
  - boolean, integer, float, string
- Typy złożone (2)
  - array, object
- Typy specjalne (2)
  - resource, NULL

! Typ zmiennych ustala PHP na podstawie kontekstu

! PHP dokonuje automatycznych konwersji typów
  - gdy w danym miejscu spodziewana jest wartość innego typu
  - w przypadku operacji (np. "+") na operandach różnych typów

! Można jawnie ustawić typ zmiennej funkcją **`settype()`** lub przez rzutowanie (jak w C)

! Do sprawdzania typu zmiennej służy rodzina funkcji **`is_`*`nazwatypu`*`()`**

Typ logiczny (boolean)

- Prawda lub fałsz
  (literały **TRUE** i **FALSE** – wielkość znaków dowolna)
- Konwersje wartości innych typów do typu boolean
  - FALSE
    (0, 0.0, pusty łańcuch znaków, łańcuch "0",
    tablice i obiekty bez elementów,
    NULL, niezainicjalizowane zmienne)
  - TRUE
    (pozostałe wartości)

```
if ($x != 0) {
...
}
```

```
if ($x) {
...
}
```

Typy liczbowe

- Liczby całkowite ze znakiem (integer)
  - np. 123, -123, 0654, 0x7FFF
  - rozmiar zależy od platformy (typowo 32-bity)
- Liczby zmiennoprzecinkowe (float)
  - np. 3.14, 4.8e5, 125E-3
  - rozmiar zależy od platformy (typowo 64-bitowy IEEE)

Łańcuchy znaków (1)

- Łańcuchy bajtowych znaków (brak wewnętrznego wsparcia UNICODE)
- Mogą być "dowolnie" długie - brak ograniczeń w języku
- 3 możliwe notacje
  - w apostrofach np. 'Tekst', 'O\'Brien'
    - bez parsowania zmiennych i sekwencji specjalnych (poza \\ i \')
  - w cudzysłowach np. "Tekst", "Hej!\n"
    - z parsowaniem zmiennych i sekwencjami specjalnymi (np. \n \$ ...)
  - heredoc (jak w Perlu)
    - z parsowaniem zmiennych i sekwencjami specjalnymi (np. \n \$ ...)

```
$wiek = 30;
echo "Mam $wiek lat.\n"; // wyświetli: Mam 30 lat. ↵
echo 'Mam $wiek lat.\n'; // wyświetli: Mam $wiek lat.\n
```

Łańcuchy znaków (2)

- Parsowanie zmiennych
  - składnia prosta
  - składnia złożona

```
echo "Kwadrat o boku $a.";
echo "Kwadrat o boku ${a}cm.";
echo "Wartość = $wektor[0].";
```

```
echo "Wartość = {$macierz[0][1]}.";
```

- Konkatenacja łańcuchów znaków (operator ".")

```
echo 'Marek' . " " . 'Wojciechowski';
```

- Indeksowanie łańcucha

```
$napis = "Ola\n";
echo $napis{0}; // Wyświetli: O
$napis{2} = "o";
echo $napis; // Wyświetli: Olo
```

- Wyrażenia regularne
  - styl Perl
  - styl POSIX Extended
- Bogaty zestaw funkcji działających na łańcuchach

Funkcje działające na łańcuchach znaków

- PHP oferuje bogaty zestaw funkcji łańcuchowych, np.
  - **`explode()`** – dzieli łańcuch tworząc tablicę
  - **`implode()`** – łączy elementy tablicy w łańcuch
  - **`echo()/print()`** – wysyła łańcuch na wyjście
  - **`printf(), sscanf(), sprintf(), ..., strlen(), strcmp()`** (dostępne są odpowiedniki niektórych funkcji łańcuchowych z C)
  - **`strstr()`** – szuka łańcucha w innym łańcuchu
  - **`str_replace()`** – podmienia podłańcuch w łańcuchu
  - **`strtolower()/ strtoupper()`**
  - **`md5()`** – haszowa sygnatura łańcucha (algorytm md5)

Tablice

- Tablica w PHP jest uporządkowaną mapą
  - mapa jest typem, przyporządkowującym wartości do kluczy
- Kluczami w tablicach PHP mogą być:
  - nieujemne liczby całkowite
  - łańcuchy znaków
- Tablice w PHP mogą być używane jako
  - "typowe tablice"
  - tablice asocjacyjne
  - listy, kolejki, stosy
- Wartościami w tablicy mogą być inne tablice
  - tablice wielowymiarowe
  - drzewa

Tworzenie tablic

```
$tab = array ('zero', 'jeden', 'dwa');
```

```
$tab = array (0 => 'zero', 1 => 'jeden', 2 => 'dwa');
```

```
$tab[0] = 'zero'; $tab[1] = 'jeden'; $tab[2] = 'dwa';
```

```
$tab[] = 'zero'; $tab[] = 'jeden'; $tab[] = 'dwa';
```

```
$stolice = array ('Czechy' => 'Praga', 'Łotwa' => 'Ryga');
```

```
$stolice['Czechy'] = 'Praga'; $stolice['Łotwa'] = 'Ryga';
```

- Odwołanie do elementu: $tab[wyrażenie_klucz]
- Dodanie nowej wartości do tablicy bez podania klucza przypisuje jej kolejny klucz całkowitoliczbowy (licząc od 0)
- Możliwe jest usunięcie danego elementu z tablicy (**`unset()`**)

Nawigacja po elementach tablicy (1)

```
for ($i = 0; $i < 3; $i++) {
  echo "$i => $tab[$i]\n";
}
```

```
foreach ($stolice as $stolica) {
  echo "$stolica\n";
}
```

```
foreach ($stolice as $kraj => $stolica) {
  echo "$kraj => $stolica\n";
}
```

Nawigacja po elementach tablicy (2)

```
reset($stolice);
while (list($klucz, $wart) = each($stolice)) {
  echo "$klucz => $wart\n";
}
```

- **`list()`** pozwala na przypisanie wartości kilku zmiennym w jednej operacji
  - po prawej stronie przypisania musi być tablica z indeksem numerycznym od 0
  - na liście można pomijać elementy – zostawiając przecinki
- **`each()`** zwraca kolejny element tablicy (klucz i wartość) w postaci 4-elementowej tablicy:
  array(0 => klucz, 1 => wartość,
  "key" => klucz, "value" => wartość)

Funkcje działające na tablicach

- PHP oferuje bogaty zestaw funkcji dla obsługi tablic
- Niektóre użyteczne funkcje działające na tablicach:
  - **`count()`** – zwraca liczbę elementów
  - **`sort()/rsort()`** – sortują tablice (rosnąco/malejąco)
  - **`asort()/arsort()`** – sortują tablice asocjacyjne
  - **`array_walk()`** – uruchamia daną funkcję dla elementów
  - **`in_array()`** – sprawdza czy dana wartość jest w tablicy
  - **`array_search()`** – zwraca klucz dla danej wartości
  - **`array_key_exists()`** – sprawdza czy dany klucz istnieje
  - **`array_intersect()`** – wyznacza część wspólną tablic
  - **`array_push()`**, **`array_pop()`** – na koniec/z końca
  - **`array_unshift()`**, **`array_shift()`** – na pocz./z pocz.

Typy specjalne

- Identyfikator zasobów (resource)
  - typ ten służy do przechowywania odnośników do zewnętrznych źródeł zasobów (np. połączenia z bazą danych)
  - wartości typu tworzone i wykorzystywane przez specjalne funkcje
- NULL
  - wartość NULL oznacza brak przechowywanej wartości
  - wielkość liter w literale **`NULL`** nieistotna

Zmienne

- Nazwy zmiennych w PHP poprzedza się znakiem $
- Wielkość liter w nazwach zmiennych ma znaczenie
- Przykłady nazw zmiennych: $zm, $ZM, $j23, $_xxx
- Zmiennych nie deklaruje się przed użyciem
  - nie ma konieczności ustalania typu zmiennej
  - typ zmiennej zmienia się w zależności od jej wartości
- PHP obsługuje zmienne referencyjne np. $x = &$y
- Użyteczne funkcje:
  - **`isset()`** – sprawdza czy zmienna jest ustawiona
  - **`unset()`** – usuwa zmienną
  - **`empty()`** – sprawdza czy zmienna jest pusta

Zasięg zmiennych

- Zmienne definiowane w funkcjach użytkownika mają charakter lokalny (mogą być statyczne)
- Zmienne globalne nie są bezpośrednio widoczne w funkcjach (!)
  - dostęp przez tablicę asocjacyjną $GLOBALS
  - dostęp przez nazwę po "zadeklarowaniu globalności"

```
$g = 5; // zmienna globalna
function Fun() {
  global $g; // deklaracja globalności – bez niej $GLOBALS["g"]

  static $s = 0; // zmienna statyczna
  $l = ...       // zmienna lokalna
  ...
}
```

Zmienne predefiniowane

- PHP udostępnia zestaw predefiniowanych tablic, zawierających zmienne pochodzące ze środowiska wywołania skryptu (serwer, formularz HTML, ...)
- Tablice te są "superglobalne" – automatycznie dostępne w każdym zasięgu
- Superglobale PHP (od wersji 4.1):
  - $GLOBALS
  - $_SERVER, $_GET, $_POST , $_COOKIE , $_FILES , $_ENV , $_REQUEST , $_SESSION

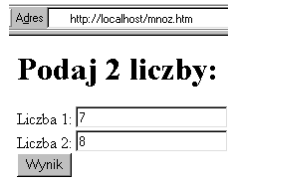
! Istnieją jeszcze predefiniowane tablice:
  - $HTTP_SERVER_VARS, $HTTP_GET_VARS, ...
  - przestarzałe, nie-superglobalne

Przetwarzanie danych z formularzy HTML (1)

- Dane z formularzy HTML są dostępne w PHP poprzez:
  - tablice superglobalne $_GET lub $_POST
  - zmienne globalne (gdy register_globals = on)

mnoz.htm

```
<HTML><BODY><H1>Podaj 2 liczby:</H1>
<FORM ACTION='mnoz.php' METHOD='get'>
Liczba 1:
<INPUT TYPE='text' NAME='p1'><BR>
Liczba 2:
<INPUT TYPE='text' NAME='p2'><BR>
<INPUT TYPE='submit' VALUE='Wynik'>
</FORM></BODY></HTML>
```



mnoz.php

```
...
<?php echo $_GET['p1'] * $_GET['p2'] ?>
<BR><?php echo $p1 * $p2 ?>
...
```

register_globals = on

Przetwarzanie danych z formularzy HTML (2)

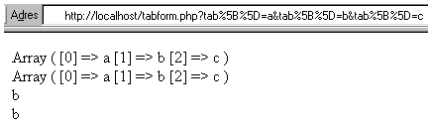
- Dane z kilku pól formularza można przekazać jako tablicę

*.html

```
...
<input name="tab[]">
<input name="tab[]">
<input name="tab[]">
...
```

*.php

```
<HTML><BODY>
<?php print_r($_GET['tab']) ?> <BR>
<?php print_r($tab) ?> <BR>
<?php echo $_GET['tab'][1] ?> <BR>
<?php echo $tab[1] ?> <BR>
</BODY></HTML>
```



- Użyteczne również w polach wielokrotnego wyboru np.
  `<select name="opcje[]" multiple> ... </select>`

Zmienne

- W PHP nazwą zmiennej może być zmienna
  - nazwę zmiennej można określać dynamicznie

```
$a = "Marek";
$$a = "Wojciechowski";
$a = "Maciej";
$$a = "Zakrzewicz";

echo $Marek;      // Wojciechowski
echo $Maciej;     // Zakrzewicz

echo $$a;         // Zakrzewicz
$a = "Marek";
echo $$a;         // Wojciechowski
```

Stałe

- Stała jest nazwą dla prostej wartości
- Przykłady nazw stałych: STALA, _MAX, x
  - wielkość liter istotna (zwyczajowo duże litery)
- Funkcje operujące na stałych:
  - **`define()`** – definiuje stałą
  - **`defined()`** – sprawdza czy stała jest zdefiniowana
  - **`constant()`** – zwraca wartość stałej o nazwie przekazanej dynamicznie
- Niezależnie od miejsca definiowania stałej, po wykonaniu **`define()`**, stała jest wszędzie widoczna
- Zdefiniowanej stałej nie można zmienić ani usunąć

```
define("STALA", 3.14);
echo STALA; // 3.14
if (defined("STALA"))
  echo constant("ST" . "ALA"); // 3.14
```

Operatory

priorytet

Powiązanie	Operator
lewe	,
lewe	or
lewe	xor
lewe	and
prawe	print
lewe	= += -= *= /= .= %= &= |= ^= ~= <<= >>=
lewe	?:
lewe	||
lewe	&&
lewe	|
lewe	^
lewe	&
bez	== != === !==
bez	< <= > >=
lewe	<< >>
lewe	+ - .
lewe	* / %
prawe	! ~ ++ -- (int) (double) (string) (array) (object) @
prawe	[
bez	new

- arytmetyczne
- inkrementacja / dekrementacja
- łańcuchowe
- bitowe
- przypisania
- porównania
- logiczne
- kontroli błędów
- konwersji typów

Instrukcje sterujące

```
if (wyrażenie)
  instrukcja
```

```
if (wyrażenie)
  instrukcja
else
  instrukcja
```

```
if (wyrażenie)
  instrukcja
elsif (wyrażenie)
  instrukcja
else
  instrukcja
```

```
while (wyrażenie)
  instrukcja
```

```
do
  instrukcja
while (wyrażenie)
```

```
for (wyr1; wyr2; wyr3)
  instrukcja
```

```
foreach ($tab as $zm)
  instrukcja
```

```
foreach ($tab as $zm1 => $zm2)
  instrukcja
```

```
switch (wyr) {
  case W1:
    instrukcje
  ...
  case WN:
    instrukcje
 [default:
    instrukcje]
}
```

```
break
break N
```

```
continue
continue N
```

- **`instrukcja`**: pojedyncza lub grupa instrukcji w **`{}`**

Alternatywna składnia instrukcji sterujących

- Alternatywna składnia dostępna dla **`if/else/elsif, while, for, foreach, switch`**
- Brak **`{}`**, zastąpione jawnym kończeniem instrukcji
- Może być użyteczna przy przeplataniu PHP i HTML

```
if (wyrażenie) :
  instrukcje
elsif (wyrażenie) :
  instrukcje
else :
  instrukcje
endif;
```

```
while (wyrażenie) :
  instrukcje
endwhile;
```

```
for (wyr1; wyr2; wyr3) :
  instrukcje
endfor;
```

```
foreach ($tab as $zm) :
  instrukcje
endforeach;
```

```
switch (wyr) :
  case W1:
    instrukcje
  ...
  case WN:
    instrukcje
 [default:
   instrukcje]
endswitch;
```

Funkcje

- Funkcje definiowane przez użytkownika w PHP:
  - definicja rozpoczyna się słowem **`function`**
  - () wymagane nawet gdy brak argumentów
  - ciało funkcji w {}
  - wielkość liter w nazwach funkcji nieistotna (!)
  - nie można przeciążać ani przedefiniować funkcji
  - wywołanie funkcji może poprzedzać jej definicję (PHP4)
  - przy wywołaniu funkcji, w miejscu nazwy może pojawić się zmienna

```
function e(){
  return 2.71;
}

echo e(); // 2.71
```

```
function e(){
  return 2.71;
}
$liczba = 'e';
echo $liczba(); // 2.71
```

Argumenty funkcji

- Można podać domyślne wartości argumentów (ostatnich!)
- Obsługiwane są listy argumentów o zmiennej długości: **`func_num_args()`**,**`func_get_arg()`**,**`func_get_args()`**
- Argumenty można przekazywać przez referencję (jak w C++)
  - gdy zmiany dokonane na nich w funkcji mają być widoczne na zewnątrz
  - nie ma sensu używać referencji dla efektywności (uniknięcie kopiowania), gdyż PHP4 obsługuje kopie jako referencje do momentu ich zmiany (!)

```
function napisz($co = 'Nic') {
  echo $co;
}
napisz('Coś');  // Coś
napisz();       // Nic
```

```
function dodaj5($x,&$y) {
  $x+=5, $y+=5;
}
$a = $b = 0;
dodaj5($a, $b);
echo $a, $b; // 05
```

Zwracanie wartości przez funkcje

- Funkcje mogą zwracać wartości (dowolnego typu, w tym tablice i obiekty)
- Instrukcja **`return`** kończy działanie funkcji i opcjonalnie zwraca wartość
- Funkcja może zwracać referencję
  - użyteczne gdy celem funkcji jest znalezienie zmiennej, do której referencja ma być dowiązania
  - zwracanie referencji zamiast kopii nie poprawia efektywności

```
function suma($a,$b)
{
  return $a + $b;
}

echo suma(4,5); // 9
```

```
function dwie() {
return array ('a','b');
}

list ($a,$b) = dwie();
echo $a, $b; // ab
```

```
function &ref()
{
  ...
  return $zm;
}
$nowar =& ref();
```

Współpraca PHP z bazami danych Oracle

Rozszerzenia PHP dla Oracle

- PHP posiada 2 rozszerzenia do obsługi baz Oracle
  - oracle: współpracujące z Oracle7
  - oci8: współpracujące z Oracle7, 8, 8i, 9i
- Rozszerzenia te wymagają bibliotek klienta Oracle
  - działają w oparciu o interfejs OCI
- Do prawidłowej pracy może być konieczne ustawienie zmiennych środowiskowych Oracle
  - dla użytkownika pod którym pracuje serwer WWW
  - na Win32 – rejestr systemu
- ! Uwagi do konfiguracji PHP + Linux/Apache:
  - Apache musi być zbudowany z biblioteką pthread (OHS w starszych wersjach nie spełnia tego wymogu)
  - PHP musi być zbudowany z opcją --enable-sigchild

Obsługa połączeń z bazą danych

- 3 różne funkcje otwierające – ten sam sposób użycia:
  - **`OCILogon()`** – otwiera połączenie współdzielone na poziomie wywołania strony
  - **`OCINLogon()`** – wymusza nowe połączenie, pozwala na separację współbieżnych transakcji w ramach wywołania strony
  - **`OCIPLogon()`** – otwiera tzw. stałe połączenie (persistent connection)
- Zamknięcie połączenia: **`OCILogoff()`**

```
$conn = OCILogon("scott", "tiger", "db");
...
OCILogoff($conn);
```

nazwa instancji
lub
nazwa usługi

Stałe połączenia z bazą danych

- Współdzielone na poziomie procesu serwera WWW
  - przy żądaniu otwarcia stałego połączenia (**`OCIPLogon()`**) jest ono fizycznie otwierane tylko gdy obsługujący proces serwera WWW nie posiada otwartego połączenia o tych samych parametrach
  - żądanie zamknięcia stałego połączenia przez skrypt (**`OCILogoff()`**) faktycznie go nie zamyka
- Dostępne dla Oracle i innych baz danych
- Dostępne tylko gdy PHP jako moduł serwera
  - w trybie CGI połączenia stałe zachowują się jak zwykłe
- Zaleta: wydajność (uniknięcie kosztu nawiązania połączenia)
- Niebezpieczeństwa:
  - możliwość wyczerpania limitu połączeń na poziomie serwera b.d.
  - potencjalne problemy przy skryptach blokujących dane i/lub korzystających z transakcji

Instrukcje DML, DDL, bloki PL/SQL

- **`OCIParse()`** – parsowanie polecenia SQL
- **`OCIExecute()`** – wykonanie parsowanego polecenia
- **`OCIRowCount()`** – można wykorzystać do odczytu liczby rekordów, których dotyczyło polecenie
- **`OCIFreeStatement()`** – zwolnienie zasobów

```
$stmt = OCIParse($conn, "update emp set sal=sal*1.1");
OCIExecute($stmt);
echo 'Liczba zmian:' . OCIRowCount($stmt);
OCIFreeStatement($stmt);
```

Parametryzacja instrukcji SQL

- Parametryzacja pozwala na wielokrotne uruchomienie raz parsowanego polecenia dla różnych wartości
- **`OCIBindByName()`** – przypisuje zmienne do parametrów polecenia SQL (również zapytań)

```
$num = 8567;
$stmt = OCIParse($conn, "update emp set sal=sal*1.1
                         where empno = :empno");
OCIBindByName($stmt, ":empno", $num, 32);
OCIExecute($stmt);
$num = 8589;
OCIExecute($stmt);
OCIFreeStatement($stmt);
```

Przetwarzanie zapytań (1)

- Po wykonaniu zapytania (**`OCIParse()`**, **`OCIExecute()`**) rekordy pobierane są za pomocą:
  - **`OCIFetch()`** – kolejny rekord do:
    - bufora domyślnego (odczyt z bufora – **`OCIResult()`** wg nazwy lub pozycji kolumny), albo
    - listy zmiennych przypisanych przez **`OCIDefineByName()`**
  - **`OCIFetchInto()`** – kolejny rekord do tablicy:
    - numerycznej (domyślnie) lub asocjacyjnej
    - z pominięciem (domyślnie) lub uwzględnieniem wartości NULL
  - **`OCIFetchStatement()`** – wszystkie rekordy do tablicy
    - odpowiednie dla zapytań zwracających małą liczbę rekordów
- Domyślnie **`OCIFetch()`** i **`OCIFetchInto()`** pobierają rekordy z bazy "po jednym"
  - dla zwiększenia wydajności można to zmienić funkcją **`OCISetPrefetch()`**

Przetwarzanie zapytań (2)

- Przykład **`OCIFetch()`**

```
$stmt = OCIParse($conn, "select empno, ename from emp");

OCISetPrefetch($stmt, 20); // opcjonalnie (20-przykładowo)

OCIDefineByName($stmt, "EMPNO", $num);
OCIDefineByName($stmt, "ENAME", $name);

OCIExecute($stmt);

while(OCIFetch($stmt)) {
  echo "emmpno: $num  ename: $name \n";
}

OCIFreeStatement($stmt);
```

Przetwarzanie zapytań (3)

- Przykład **`OCIFetchInto()`**

```
$stmt = OCIParse($conn, "select empno, ename from emp");

OCISetPrefetch($stmt, 20); // opcjonalnie (20-przykładowo)

OCIExecute($stmt);

while(OCIFetchInto($stmt, $row, OCI_RETURN_NULLS)) {
  echo "emmpno: $row[0]  ename: $row[1] \n";
}

OCIFreeStatement($stmt);
```

Transakcje

- **`OCIExecute()`** posiada opcjonalny drugi parametr – tryb:
  - **`OCI_COMMIT_ON_SUCCESS`** (domyślnie)
  - **`OCI_DEFAULT`** (brak automatycznego zatwierdzania)
- **`OCICommit()`** – zatwierdza niezatwierdzone operacje z podanego połączenia
- **`OCIRollback()`** – wycofuje niezatwierdzone operacje z podanego połączenia

```
$stmt = OCIParse($conn, "insert into emp(empno,ename)
                         values(1111, 'X')");
OCIExecute($stmt, OCI_DEFAULT);  OCIFreeStatement($stmt);

$stmt = OCIParse($conn, "insert into emp(empno,ename)
                         values(1112, 'Y')");
OCIExecute($stmt, OCI_DEFAULT);  OCIFreeStatement($stmt);

OCICommit($conn);   // albo: OCIRollback($conn);
```

Odczyt błędów Oracle

- Do odczytu błędów Oracle służy **`OCIError()`**
- Możliwe parametry **`OCIError()`**:
  - brak: ostatni błąd, który wystąpił
  - "statement handle": ostatni błąd, który wystąpił w danym poleceniu SQL
  - "connection handle": ostatni błąd, który wystąpił w ramach danego połączenia z bazą danych
- Wartości zwrotne **`OCIError()`**:
  - **`FALSE`** – gdy nie było błędu
  - tablica asocjacyjna (indeksy: **`'code'`**, **`'message'`**) gdy był błąd

Active Server Pages

Active Server Pages (ASP)

- Technologia Microsoftu, umożliwiająca tworzenie aplikacji internetowych pracujących po stronie serwera WWW
- Aplikacje ASP są zapisywane w postaci plików, nazywanych dokumentami ASP
- Dokument APS jest plikiem HTML, zawierającym zagnieżdżony kod w języku VBScript
- Dokumenty ASP są przetwarzane w odpowiedzi na żądanie użytkownika, przez serwer WWW (np. Microsoft Internet Information Server, Microsoft Personal web server)
- Język VBScript jest podzbiorem języka Microsoft Visual Basic
- Aplikacje ASP mogą współpracować z systemami baz danych przy użyciu obiektów ADO (ActiveX Data Objects)

Przykład aplikacji ASP (1)

```
<% @LANGUAGE=VBScript %>
<HTML>
<BODY>
<CENTER>
Today is <B><% =now() %></B>
and all is well.<BR><H2>
<% IF hour(now())>12 THEN %>
Good Evening
<% ELSE %>
Good Morning!
<% END IF %>
</H2></CENTER>
</BODY></HTML>
```

wybór języka skryptowego (opcjonalnyl)

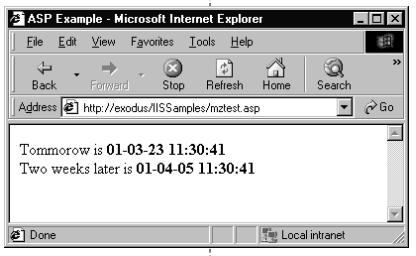
wyświetla wynik funkcji now()

wyświetla, jeżeli warunek był spełniony

wyświetla, jeżeli warunek nie był spełniony

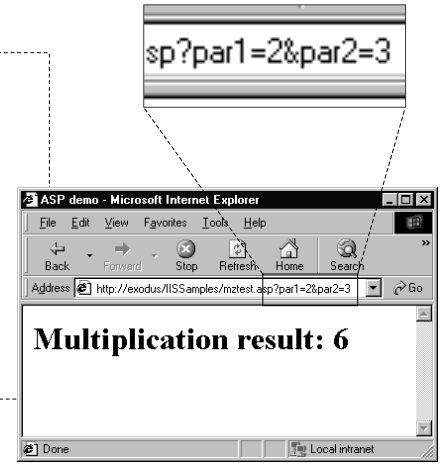
Przykład aplikacji ASP (2)

```
<HTML><HEAD>
<TITLE>ASP Example</TITLE>
</HEAD>
<BODY>
<%
  when=now()
  tommorow=dateadd("d",1,when)
  twoweekslater=dateadd("ww",2,when)
%>
Tommorow is <B> <% =tommorow %> </B><BR>
Two weeks later is <B>
<% =twoweekslater %> </B><BR>
</BODY></HTML>
```



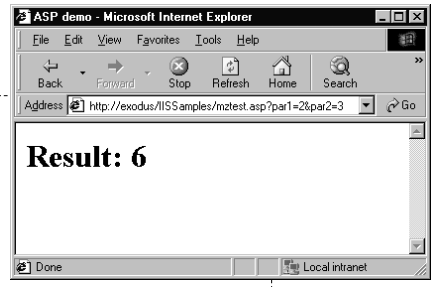
Przekazywanie parametrów wywołania

```
<HTML>
<HEAD>
<TITLE>ASP demo</TITLE>
</HEAD>
<BODY>
<% term1 = Request("par1")
   term2 = Request("par2")
   result = term1 * term2 %>
<H1> Multiplication result:
<% =result %>
</H1>
</BODY>
</HTML>
```



Obiekt Response

```
<HTML>
<HEAD>
<TITLE>ASP demo</TITLE>
</HEAD>
<BODY>
<% term1 = Request("par1")
   term2 = Request("par2")
   result = term1 * term2
   call Response.Write("<H1>Result: " & result & "</H1>") %>
</BODY>
</HTML>
```

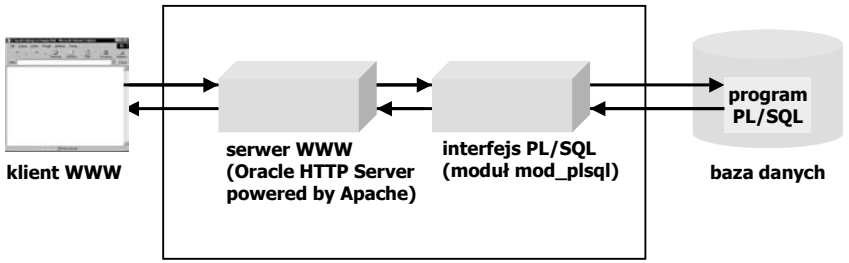


konkatenacja tekstów

wyświetl wewnątrz wynikowego dokumentu HTML

Architektura Oracle mod_plsql

Architektura mod_plsql



Część Oracle9i Application Server (iAS)

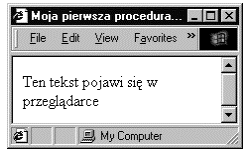
- komponenty dostępne również jako składniki instalacji serwera bazy danych:
Oracle8i Release 3 (8.1.7), Oracle 9i

Funkcjonowanie mod_plsql

1. Przeglądarka WWW użytkownika wysyła żądanie w postaci adresu URL, np. `http://serv01/pls/DAD/procedura01` (mapowanie ścieżek w adresie URL na definicje połączeń z bazą danych stanowi element konfiguracji mod_plsql)
2. Serwer WWW analizuje adres URL i przekazuje żądanie do modułu interfejsu PL/SQL (mod_plsql)
3. Moduł interfejsu PL/SQL nawiązuje połączenie z bazą danych i uruchamia umieszczoną w niej procedurę PL/SQL, np. `procedura01`
4. Uruchomiona procedura PL/SQL generuje kody HTML
5. Wygenerowany dokument HTML jest przekazywany do przeglądarki WWW użytkownika

Przykład procedury PL/SQL

```
create procedure demo is
begin
 htp.htmlOpen;
 htp.headOpen;
 htp.title('Moja pierwsza procedura');
 htp.headClose;
 htp.bodyOpen;
 htp.print('Ten tekst pojawi się w przeglądarce');
 htp.bodyClose;
 htp.htmlClose;
end;
```




```
<HTML>
<HEAD>
<TITLE>Moja pierwsza procedura</TITLE>
</HEAD>
<BODY>
Ten tekst pojawi się w przeglądarce
</BODY>
</HTML>
```

Biblioteka programisty: PL/SQL Web Toolkit

Biblioteka programisty: HTP (1/3)

- Pakiet HTP zawiera procedury, które generują elementy HTML, przesyłane do przeglądarki:
  - htp.htmlOpen = <HTML>
  - htp.htmlClose = </HTML>
  - htp.bodyOpen = <BODY>
  - htp.bodyClose = </BODY>
  - htp.headOpen = <HEAD>
  - htp.headClose = </HEAD>
  - htp.title(*text*) = <TITLE> *text* </TITLE>
  - **htp.print(*text*)** = *text*
  - Htp.prn = *text (bez kończącego znaku końca linii)*
  - htp.bold(*text*) = <B>*text* </B>
  - htp.italic(*text*) = <I> *text* </I>
  - htp.header(*size*, *text*) = <H*size*>*text*</H*size*>
  - htp.para = <P>
  - htp.br =

Tekst może zawierać znaczniki !!!

Biblioteka programisty: HTP (2/3)

- Procedury, które generują elementy HTML, przesyłane do przeglądarki c.d.:
  - htp.olistOpen = <OL>
  - htp.listItem(*text*) = <LI> *text* </LI>
  - htp.olistClose = </OL>
  - htp.ulistOpen = <UL>
  - htp.ulistClose = </UL>
  - htp.line = <HR>
  - htp.anchor(*link, text*) = <A HREF=*link*> *text* </A>
  - htp.img(*url*) = <IMG SRC=*url*>
- Procedury generujące tabelki:
  - htp.tableOpen(*border*) = <TABLE *border*>
  - htp.tableClose = </TABLE>
  - htp.tableRowOpen(*calign, cvalign*) = <TR ALIGN=*calign* VALIGN=*cvalign*>
  - htp.tableRowClose = </TR>
  - htp.tableData(*text*, *calign, cattr*) = <TD ALIGN=*calign cattr*>*text*</TD>
  - htp.tableHeader(*text*) = <TH> *text* </TH>

Biblioteka programisty: HTP (3/3)

- Procedury generujące elementy formularza HTML :
  - htp.formOpen(*proc*) = <FORM ACTION=*proc*>
  - htp.formClose = </FORM>
  - htp.formText(*name, size*) = <INPUT TYPE=TEXT NAME=*name* SIZE=*size*>
  - htp.formRadio(*name, value*) = <INPUT TYPE=RADIO NAME=*name* VALUE=*value*)
  - htp.formSelectOpen(*name*) = <SELECT NAME=*name*>
  - htp.formSelectOption(*text*) = <OPTION> *text*
  - htp.formSelectClose = </SELECT>
  - htp.formHidden(*name, val*) = <INPUT TYPE=HIDDEN NAME=*name* VALUE=*val*>
  - htp.formTextArea(*name, r, c*) = <TEXTAREA NAME=*name* ROWS=*r* COLS=*c*> </TEXTAREA>
  - htp.formSubmit(cvalue=>'*text*') = <INPUT TYPE=SUBMIT VALUE=*text*>
  - htp.formReset(*text*) = <INPUT TYPE=RESET VALUE=*text*>

Biblioteka programisty: HTF

- Pakiet HTF zawiera funkcje, których nazwy pokrywają się z nazwami procedur z pakietu HTP
- Funkcje pakietu HTF nie wysyłają kodów HTML do przeglądarki użytkownika - zwracają one ciągi tekstowe, które mogą być dalej przetwarzane, np.:
  - htp.italic(htf.bold('Tłusty pochyły')) = <I><B>Tłusty pochyły</B></I>
  - htp.anchor('www.oracle.com',htf.img('/img/buttn.gif')) = <A HREF=„www.oracle.com"><IMG SRC=„/img/buttn.gif"></A>

Przykładowa procedura PL/SQL (1)

Wyświetl imiona i nazwiska wszystkich pracowników z tabeli EMP

```
create procedure demo_emp is
begin
 htp.htmlOpen;
 htp.headOpen;
 htp.title('Skład osobowy');
 htp.headClose;
 htp.bodyOpen;
 htp.header(1,'Skład osobowy');
 for kursor in
  (select firstname, lastname from emp order by lastname)
  loop
    htp.print(kursor.firstname);
    htp.print(kursor.lastname);
    htp.br;
  end loop;
 htp.bodyClose;
 htp.htmlClose;
end;
```

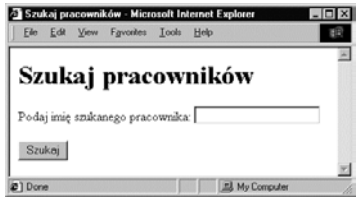
Przykładowa procedura PL/SQL (2)

Wyświetl imiona i nazwiska wszystkich pracowników z tabeli EMP w tabelce HTML


```
create procedure demo_emp is
begin
 htp.htmlOpen; htp.headOpen;
 htp.title('Skład osobowy');
 htp.headClose; htp.bodyOpen;
 htp.header(1,'Skład osobowy');
 htp.tableOpen('border=1');
 htp.tableRowOpen;
 htp.tableHeader('Imię');
 htp.tableHeader('Nazwisko');
 htp.tableRowClose;
 for kursor in
 (select firstname, lastname from emp order by lastname) loop
    htp.tableRowOpen;
    htp.tableData(kursor.firstname);
    htp.tableData(kursor.lastname);
    htp.tableRowClose;
  end loop;
 htp.tableClose;
 htp.bodyClose; htp.htmlClose;
end;
```

Przykładowa procedura PL/SQL (3)

Wyświetl formularz, który pobiera imię szukanego pracownika




```
create procedure seek_emp_form is
begin
 htp.htmlOpen;
 htp.headOpen;
 htp.title('Szukaj pracowników');
 htp.headClose;
 htp.bodyOpen;
 htp.header(1,'Szukaj pracowników');
 htp.formOpen('seek_emp_exec');
 htp.print('Podaj imię szukanego pracownika:');
 htp.formText('P_IMIE');
 htp.br; htp.br;
 htp.formSubmit(cvalue=>'Szukaj');
 htp.formClose;
 htp.bodyClose;
 htp.htmlClose;
end;
```

Przykładowa procedura PL/SQL (4)

Obsługa formularza - wyświetl pracowników o imieniu podanym jako parametr wywołania:

```
create procedure seek_emp_exec (P_IMIE in varchar2) is
begin
 htp.htmlOpen; htp.headOpen;
 htp.title('Wyniki wyszukiwania');
 htp.headClose; htp.bodyOpen;
 htp.header(1,'Pracownicy o imieniu: '||P_IMIE);
 htp.tableOpen('border=1');
 htp.tableRowOpen;
 htp.tableHeader('Imię'); htp.tableHeader('Nazwisko');
 htp.tableRowClose;
 for kursor in (select firstname, lastname from emp
                where firstname=P_IMIE order by lastname) loop
    htp.tableRowOpen;
    htp.tableData(kursor.firstname);
    htp.tableData(kursor.lastname);
    htp.tableRowClose;
  end loop;
 htp.tableClose;
 htp.bodyClose; htp.htmlClose;
end;
```

Dodatkowe pakiety biblioteczne (1/6)

- Procedura wyświetlająca zawartość tabeli bazy danych w formie tabelki HTML: OWA_UTIL.TABLEPRINT

```
create procedure demo is
  x boolean;
begin
  x:=owa_util.tableprint('emp');
end;
```

- Procedura wyświetlająca wynik zapytania SQL w formie tabelki HTML: OWA_UTIL.CELLSPRINT

```
create procedure demo is
  x boolean;
begin
  x:=owa_util.cellsprint('select * from emp');
end;
```

Dodatkowe pakiety biblioteczne (2/6)

- Zmiana domyślnej (text/html) informacji o formacie generowanej zawartości: OWA_UTIL.MIME_HEADER
  - Ustawienie informacji o kodowaniu znaków:

```
create procedure demo is
begin
 owa_util.mime_header('text/html; charset= windows-1250');
  ...
end;
```

  - Wyświetlanie obrazów graficznych zapisanych w bazie danych:

```
create procedure demo is
  x raw;
begin
  owa_util.mime_header('image/jpeg');
  select image into x from my_imgs where image_id=101;
  htp.print(utl_raw.cast_to_varchar2(x));
```

Dodatkowe pakiety biblioteczne (3/6)

- Funkcje dostępu do pseudo-zmiennych środowiskowych CGI:

  OWA_UTIL.GET_CGI_ENV – zwraca wartość podanej zmiennej CGI

```
create procedure demo is
begin
  htp.print(owa_util.get_cgi_env('REMOTE_ADDR'));
end;
```

  OWA_UTIL.PRINT_CGI_ENV – wyświetla wartości wszystkich zmiennych CGI

```
create procedure demo is
begin
  owa_util.print_cgi_env;
end;
```

Dodatkowe pakiety biblioteczne (4/6)

- „Cookies" to trwałe zmienne przechowywane przez serwer WWW w pamięci przeglądarki użytkownika
- Z każdą zmienną związana jest jej nazwa, wartość, opis i czas życia
- Zastosowania: symulacja transakcji, wymuszenie ścieżki nawigacyjnej, kontrola dostępu, itd..
- Dostęp do „cookies" daje pakiet OWA_COOKIE:

```
create procedure licznik_wizyt is
  my_cookie owa_cookie.cookie;
begin
  my_cookie := owa_cookie.get('licznik');
  owa_util.mime_header('text/html',FALSE); --nie zamykaj nagłówka
  if my_cookie.num_vals>0 then
   owa_cookie.send('licznik',to_number(my_cookie.vals(1))+1);
  else
   owa_cookie.send('licznik',2);
  end if;
  owa_util.http_header_close;
  if my_cookie.num_vals > 0 then
     htp.print('To jest twoja '||my_cookie.vals(1)||' wizyta!');
  else
     htp.print('To jest twoja pierwsza wizyta!');
  end if;
end;
```

Dodatkowe pakiety biblioteczne (5/6)

- Protokół HTTP nie obsługuje współbieżności dostępu
- Przykładowy problem: (1) użytkownik A odczytuje dane pracownika X, (2) użytkownik B odczytuje dane pracownika X, (3) użytkownik B modyfikuje dane pracownika X, (4) użytkownik B zapisuje zmiany w bazie danych, (5) użytkownik A rozpoczyna modyfikację danych pracownika X, nie będąc świadomym, że są one już nieaktualne!
- Programista może skorzystać z mechanizmu programowej kontroli współbieżności: przed modyfikacją danych w bazie danych sprawdza się, czy nie nastąpiła ich współbieżna modyfikacja
- Rozwiązanie:

  (1) Zapamiętanie oryginalnych wartości w polach HIDDEN

  (2) W chwili zapisu zmodyfikowanych danych – porównanie aktualnych wartości w bazie danych z zapamiętanymi poprzednio

Dodatkowe pakiety biblioteczne (6/6)

(1) Zapamiętanie oryginalnych wartości w polach HIDDEN

```
create procedure demo is
  cursor cur_1 is
  select rowid, firstname, lastname from emp
  where lastname='Kowalski';
begin
  -- wartości pobrane z kursora wyświetl w formularzu na ekranie
  fetch cur_1 into x;
  ...
  owa_opt_lock.store_values('blk01','emp',x.rowid);
  ...
end;
```

(2) Porównanie aktualnych wartości w bazie danych z zapamiętanymi poprzednio

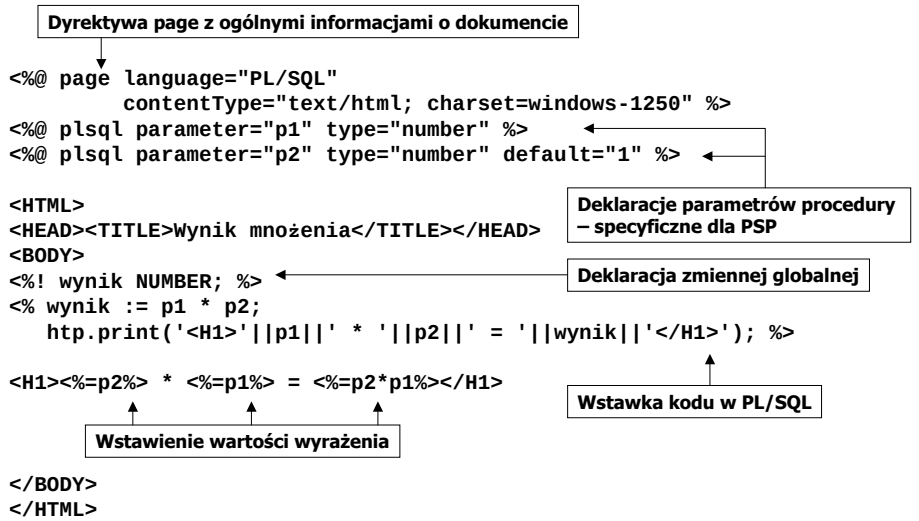
```
create procedure demo2 (old_emp in owa_opt_lock.vcArray, ...) is
  the_rowid rowid;
begin
  the_rowid := owa_opt_lock.get_rowid(old_emp);
  if (owa_opt_lock.verify_values(old_emp)) then
     update ... where rowid=the_rowid
  else htp.print('Inny użytkownik zmodyfikował te dane – powtórz operację');
  end if;
end;
```

PL/SQL Server Pages

Geneza PL/SQL Server Pages (PSP)

- Podstawową wadą procedur PL/SQL (podobnie jak programów CGI, serwletów) jest konieczność programowego generowania nie tylko zmiennych części dokumentów, ale również niezmiennych (statycznych) fragmentów HTML
- Rozwiązaniem problemu jest ogólna metodologia (server pages, server-side scripting) polegająca na zagnieżdżaniu w dokumentach HTML fragmentów kodu generujących fragmenty zmienne (które muszą być wygenerowane programowo):
  - ASP (Active Server Pages)
  - JSP (Java Server Pages)
  - **PSP (PL/SQL Server Pages)**
- Server pages wykorzystują specjalne znaczniki:
  - <% ... %>, <%@ ... %>, <%= ... %>, <%! ... %>
- Server pages umożliwiają korzystanie z narzędzi autorskich dla HTML przy projektowaniu stron

PSP - Przykład



Ładowanie PSP do bazy danych (1/2)

- Plik PSP należy jawnie załadować do bazy danych narzędziem loadpsp
- loadpsp dokonuje transformacji PSP na treść CREATE PROCEDURE

multiply.psp

```
<%@ page language="PL/SQL" contentType="text/html; charset=windows-1250" %>
<%@ plsql parameter="p1" type="number" %>
<%@ plsql parameter="p2" type="number" default="1" %>

<HTML>
<HEAD><TITLE>Wynik mnożenia</TITLE></HEAD>
<BODY>
<%! wynik NUMBER; %>
<% wynik := p1 * p2;
   htp.print('<H1>'||p1||' * '||p2||' = '||wynik||'</H1>'); %>
<H1><%=p2%> * <%=p1%> = <%=p2*p1%></H1>
</BODY>
</HTML>
```

loadpsp -user scott/tiger@ora92 multiply.psp

Procedura MULTIPLY w schemacie SCOTT w bazie ora92

Ładowanie PSP do bazy danych (2/2)

Plik multiply.psp w systemie plików

loadpsp -user scott/tiger@ora92 multiply.psp

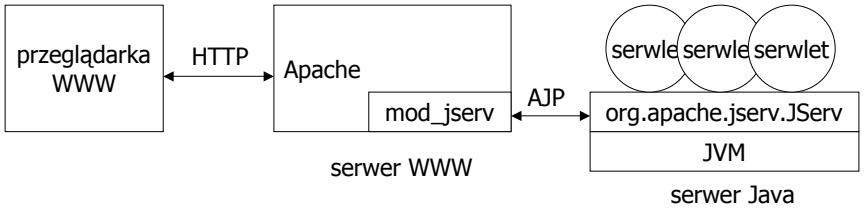
```
PROCEDURE multiply (
p1 IN number,
p2 IN number default 1) AS
 wynik NUMBER;
 BEGIN NULL;
owa_util.mime_header('text/html; charset=windows-1250'); htp.prn('
');
htp.prn('
');
htp.prn('

<HTML>
<HEAD><TITLE>Wynik mnożenia</TITLE></HEAD>
<BODY>
');
htp.prn('
');
 wynik := p1 * p2;
htp.print('<H1>'||p1||' * '||p2||' = '||wynik||'</H1>');
htp.prn('
<H1>');
htp.prn(p2); htp.prn(' * '); htp.prn(p1);
htp.prn(' = '); htp.prn(p2*p1);
htp.prn('</H1>
');
htp.prn('
</BODY>
</HTML>
');
 END;
```

Tworzenie serwletów Java

Zalety stosowania serwletów Java

- Wydajność
  - w porównaniu z CGI, gdzie obsługa każdego żądania wymaga utworzenia nowego procesu w systemie operacyjnym, serwlety Java korzystają
    z wielowątkowości i są buforowane w pamięci operacyjnej
- Wygoda
  - możliwość zastosowania umiejętności programowania w języku Java do konstrukcji aplikacji internetowych
- Funkcjonalność
  - ogromne bogactwo bibliotek i rozwiązań dostępnych dla języka Java może być wykorzystane podczas tworzenia serwletów Java
- Przenaszalność
  - kod i struktura serwletów Java są niezależne od platformy systemowej
    i mogą być uruchamiane w na każdym serwerze Java

Architektura serwera Java: Apache JServ

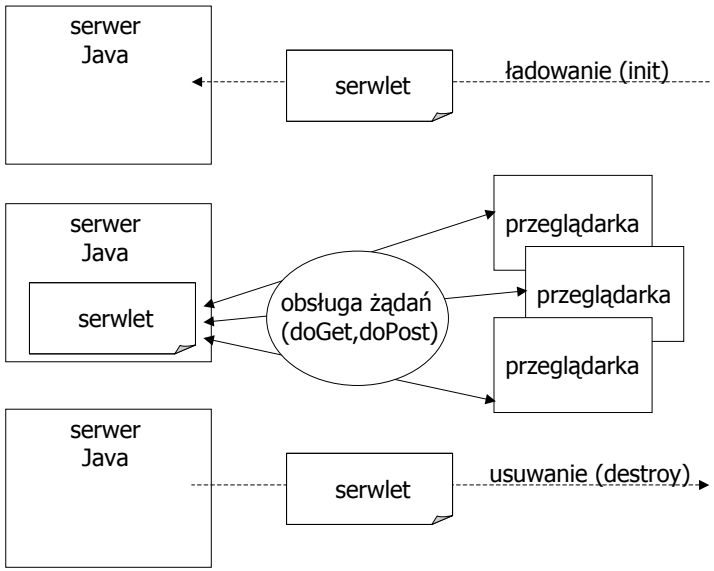


Na żądanie przesłane przez przeglądarkę użytkownika, serwer Apache komunikuje się z działającym programem JServ, od którego żąda uruchomienia wskazanego serwletu Java. Po uruchomieniu, servlet Java generuje kod HTML, przekazywany następnie, poprzez JServ i Apache, na ekran przeglądarki użytkownika.
Załadowany serwlet pozostaje w pamięci operacyjnej i może obsługiwać kolejne żądania.
JServ buforuje treść generowaną przez serwlety.

Charakterystyka serwletów Java

- Serwlet jest programem Java zapisanym w postaci klasy dziedziczonej z klasy `javax.servlet.http.HttpServlet`
- Programista musi dostarczyć własną implementację co najmniej jednej z metod:
  - `public void` **`doGet`**`(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException`
  - `public void` **`doPost`**`(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException`
  - `public void` **`init`**`(ServletConfig config) throws ServletException`
  - `public void` **`destroy`**`()`

Cykl życia serwletu



Przykład prostego serwletu

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("<H1>Moj serwlet!</H1>");
  }
}
```

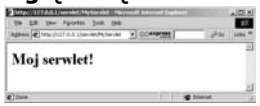
PrintWriter
HttpServlet
HttpServletRequest,
HttpServletResponse

Typ programu: serwlet

Będzie generowany kod HTML

Obiekt `out` reprezentuje wyjście na przeglądarkę

Wysłanie kodu HTML na ekran przeglądarki



Interfejs HttpServletRequest

- Klasa `HttpServletRequest` umożliwia dostęp do pól w treści żądania otrzymanego od przeglądarki

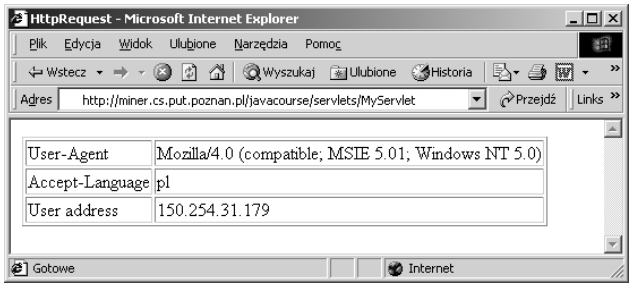
`Cookie[] getCookies()`	Odczytuje tablicę wszystkich zmiennych Cookies przesłanych przez serwer WWW
`String getHeader(n)`	Odczytuje wartość pola *n* dostarczonego w nagłówku HTTP
`String getMethod()`	Odczytuje nazwę typu żądania przeglądarki: GET, POST, PUT
`String getRemoteUser()`	Odczytuje nazwę, pod jaką zalogowany jest użytkownik przeglądarki, która przesłała żądanie
`HttpSession getSession()`	Odczytuje lub tworzy obiekt aktualnej sesji
`String getPathInfo()`	Odczytuje pełną ścieżkę użytą przez przeglądarkę w adresie URL
`String getParameter(n)`	Odczytuje wartość przekazanego przez użytkownika parametru wywołania *n*
`String getRemoteAddr()`	Odczytuje adres IP przeglądarki

Pola w nagłówku żądania HTTP

- `Accept`
  - wykaz formatów danych akceptowanych przez przeglądarkę
- `Accept-Encoding`
  - wykaz formatów kompresji danych, akceptowanych przez przeglądarkę
- `Accept-Language`
  - kod języka narodowego, w których przeglądarka chce otrzymać odpowiedź
- `User-Agent`
  - nazwa programu przeglądarki
- `Referer`
  - adres dokumentu, którego link spowodował wysłanie tego żądania
- `Authorization`
  - dane autoryzujące użytkownika
- `If-Modified-Since`
  - jeżeli dokument nie uległ zmianie od tego czasu, przeglądarka może otrzymać tylko odpowiedź "Not Modified 304"
- `Cookie`
  - lista zmiennych Cookies

Wykorzystanie HttpServletRequest

```
out.println("<table border=1>");
out.println("<tr><td>User-Agent</td><td>"+
            request.getHeader("User-Agent")+"</td>");
out.println("<tr><td>Accept-Language</td><td>"+
            request.getHeader("Accept-Language")+"</td>");
out.println("<tr><td>User address</td><td>"+
            request.getRemoteAddr()+"</td>");
out.println("</table>");
```



Dostęp do parametrów wywołania serwletu

- Przeglądarka użytkownika może w URL żądania umieścić parametry wywołania serwletu, np.:

```
http://serwer/ścieżka/mój_serwlet?x=1&y=2&z=kowalski
```

- Programista uzyskuje dostęp do parametrów wywołania przy pomocy obiektu **`HttpServletRequest`**:

```
nazwisko = request.getParameter("z");
```

Wielkość znaków jest istotna

Przykład serwletu z parametrem (1)

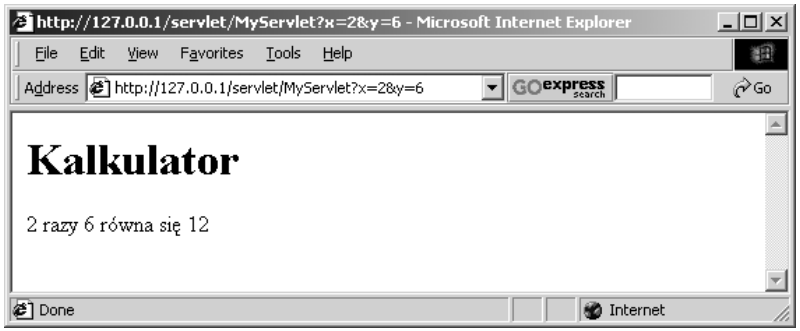
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   int znak;

   response.setContentType("text/html; charset=iso-8859-2");
   PrintWriter out = response.getWriter();
   String fName = request.getParameter("name");

   InputStream fStream = new FileInputStream("/myfiles/"+fName);
   InputStreamReader fReader = new InputStreamReader(
                                  fStream,"ISO-8859-2");
   while ((znak = fReader.read()) != -1) out.print((char)znak);

   fReader.close();
   fStream.close();}}
```

Przykład serwletu z parametrem (2)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
   throws ServletException, IOException {

    int liczba1, liczba2, wynik;

    response.setContentType("text/html;charset=iso-8859-2");
    PrintWriter out = response.getWriter();

    liczba1 = Integer.parseInt(request.getParameter("x"));
    liczba2 = Integer.parseInt(request.getParameter("y"));
    wynik = liczba1*liczba2;

    out.println("<H1>Kalkulator</H1>");
    out.println(liczba1+" razy "+liczba2+" równa się "+wynik);
}}
```

Przykład wywołania kalkulatora: http://.../MyServlet?x=2&y=6
Na ekranie przeglądarki zostanie wyświetlony wynik mnożenia.

Serwlet z parametrem (2) - wynik

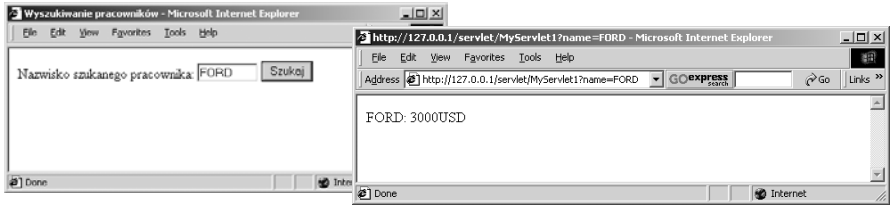


Formularze HTML

- Zamiast zapisywać parametry wywołania serwletu w adresie URL, można zbudować dokument HTML, zawierający formularz (pola tekstowe, przyciski wyboru, listy rozwijane itd.)
- Wartości wprowadzone przez użytkownika do pól formularza stają się parametrami wywołania serwletu
- Nazwy wysyłanych parametrów wywołania serwletu są takie same jak nazwy pól formularza

Przykład formularza do wprowadzania wartości parametrów

```
<HTML>
 <HEAD>
  <TITLE>Wyszukiwanie pracowników</TITLE>
 </HEAD>
 <BODY>
  <FORM ACTION="/servlet/MyServlet" METHOD=GET>
   Nazwisko szukanego pracownika:
   <INPUT TYPE="text" NAME="name" SIZE=8>
   <INPUT TYPE="submit" VALUE="Szukaj">
  </FORM>
 </BODY>
</HTML>
```



Znaczniki HTML służące do tworzenia formularzy (1)

- `<FORM ACTION=url METHOD=m></FORM>`
  - Początek/koniec definicji formularza; po zatwierdzeniu formularza przez użytkownika zostanie wywołana aplikacja *url*, a parametry wywołania będą przekazane metodą *m* (GET lub POST); znaczniki te muszą otaczać wszystkie pozostałe znaczniki formularza
- `<INPUT TYPE='text' NAME=n VALUE=v SIZE=s>`
  - Pole tekstowe o nazwie *n*, wartości domyślnej *v* i rozmiarze *s* znaków abc
- `<INPUT TYPE='password' NAME=n VALUE=v SIZE=s>`
  - Jak wyżej, lecz zawartość wprowadzana do pola nie jest ujawniana na ekranie
- `<INPUT TYPE='radio' NAME=n VALUE=v CHECKED>`
  - Przycisk radiowy należący do grupy przycisków radiowych o nazwie *n*; włączonemu przyciskowi odpowiada wartość *v*; jeżeli pojawia się CHECKED, to przycisk ten będzie domyślnie włączony

Znaczniki HTML służące do tworzenia formularzy (2)

- `<INPUT TYPE='checkbox' NAME=n VALUE=v CHECKED>`
  - Przycisk wyboru o nazwie *n*; włączonemu przyciskowi odpowiada wartość *v*; jeżeli pojawia się CHECKED, to przycisk ten będzie domyślnie włączony
- `<INPUT TYPE='hidden' NAME=n VALUE=v>`
  - Pole tekstowe ukryte o nazwie *n* i wartości *v*; nie wyświetlane na ekranie, jego wartość musi być umieszczona wewnątrz znacznika
- `<INPUT TYPE='submit' NAME=n VALUE=v>`
  - Przycisk zatwierdzający formularz; jego naciśnięcie powoduje wysłanie przez przeglądarkę żądania uruchomienia aplikacji o adresie URL podanym w znaczniku FORM; aplikacji tej zostaną przesłane parametry wywołania oparte o wartości wszystkich elementów formularza (pól tekstowych, przycisków wyboru, itd.); *v* opisuje etykietę przycisku Submit

Znaczniki HTML służące do tworzenia formularzy (3)

- `<INPUT TYPE='image' SRC=url>`
  - Obraz graficzny pełniący funkcję przycisku zatwierdzającego formularz; *url* określa lokalizację pliku obrazu Pomoc
- `<INPUT TYPE='reset' VALUE=v>`
  - Przycisk przywracający wszystkim elementom formularza wartości początkowe; *v* opisuje etykietę przycisku Reset
- `<SELECT NAME=n SIZE=s MULTIPLE></SELECT>`
  - Początek/koniec definicji listy/listy rozwijanej; element otrzyma nazwę *n*, będzie jednocześnie wyświetlać *s* pozycji; jeżeli pojawia się MULTIPLE, to użytkownik ma możliwość wyboru wielu pozycji jednocześnie; wewnątrz tego znacznika znajdują się znaczniki `<OPTION>` Fiat

Znaczniki HTML służące do tworzenia formularzy (4)

- `<OPTION VALUE=v CHECKED></OPTION>`
  - Początek/koniec definicji pojedynczej pozycji listy; wartość *v* będzie wysłana przez formularz, jeżeli pozycja ta zostanie wybrana; jeżeli pojawia się CHECKED, to pozycja ta jest zaznaczona domyślnie; wewnątrz znacznika znajduje się wyświetlany tekst
- `<TEXTAREA NAME=n ROWS=r COLS=c><TEXTAREA>`
  - Obszar tekstowy (wielowierszowe pole tekstowe) o nazwie *n*, *c* kolumnach i *r* wierszach; wewnątrz znacznika znajduje się tekst domyślny

Przykład złożonego formularza

```
<form action='/mv/srch'><table class=table_border width=100%>
  <tr><td><table cellspacing=0 cellpadding=4 width=100%>
    <tr class=row_caption><td colspan=2 align=center>Wyszukaj film</td>
    <tr class=row_odd><td>tytuł</td>
     <td><input type='text' name=tytul></td>
    <tr class=row_even><td>reżyseria</td>
     <td><input type='text' name=director></td>
    <tr class=row_odd><td>obsada</td>
     <td><input type='text' name=cast></td>
    <tr class=row_even><td>gatunek</td>
     <td><select name=category>
         <option>dowolny</option>
         <option>ACTION</option>
         <option>COMEDY</option>
         <option>SCI-FI</option>
         </select></td>
    <tr class=row_odd><td>zwiastun</td>
     <td><input type='checkbox' name=istrailer></td>
     <tr class=row_odd><td colspan=2 align=right><br>
     <input type='submit' value='Szukaj' class=button_even></td>
   </table></td></table></form>
```

Przykład złożonego formularza

GET/POST - metody przekazywania parametrów

- Formularz może przekazywać parametry wywołania serwletu przy użyciu jednej z metod:
  - GET: łańcuch nazw i wartości wszystkich parametrów jest dołączany do adresu URL wołanego serwletu
  - POST: łańcuch nazw i wartości wszystkich parametrów jest wysyłany za nagłówkiem HTTP żądania
- Do wyboru metody służy atrybut `METHOD` znacznika <FORM>
- Jeżeli parametry przekazane są metodą GET, to w serwlecie wywoływana jest metoda `doGet()`; jeżeli zastosowano POST, to w serwlecie wywołana będzie metoda `doPost()`
- W celu stworzenia serwletu uniwersalnego, należy np. w implementacji metody `doPost()` umieścić wywołanie `doGet()`

Interfejs HttpServletResponse

- Klasa `HttpServletResponse` umożliwia dostęp do pól nagłówka odpowiedzi wysyłanej do przeglądarki

`void addCookie(c)`	Umieszcza w nagłówku odpowiedzi zmienną Cookie *c*
`void addHeader(n,v)`	Umieszcza w nagłówku odpowiedzi nowe pole *n* o wartości *v*
`void sendError(v)`	Wysyła nagłówek z kodem błędu *v*
`void sendRedirect(url)`	Wysyła nagłówek przekierowania na adres *url*
`void setHeader(n,v)`	Ustawia w nagłówku odpowiedzi pole *n* na wartość *v*
`PrintWriter getWriter()`	Pobiera kanał komunikacyjny dla zapisu tekstowego do przeglądarki
`ServletOutputStream getOutputStream()`	Pobiera kanał komunikacyjny dla zapisu binarnego do przeglądarki
`void flushBuffer()`	Wymusza wysłanie zawartości bufora wyjściowego do przeglądarki

getOutputStream() - przykład użycia

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   int value;
   FileInputStream fStream = new FileInputStream("/home/logo.jpg");

   response.setContentType("image/jpeg");
   OutputStream out = response.getOutputStream();

   while ((value = fStream.read()) != -1) out.write(value);

   fStream.close();
  }
}
```

Serwlet wysyła do przeglądarki obraz graficzny JPG, odczytany z pliku dyskowego.

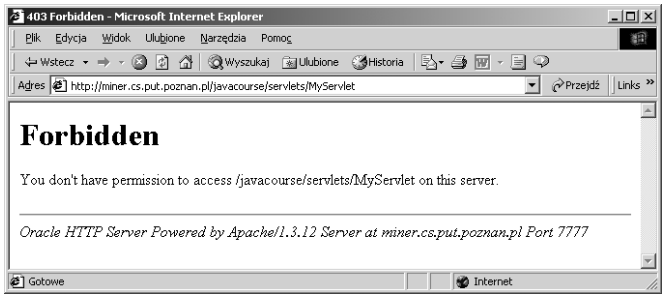
Kody błędów umieszczane w nagłówku odpowiedzi HTTP

- Kody błędów wysyłane przez metodę `sendError()`

`SC_BAD_REQUEST`	(400) Niewłaściwy format żądania
`SC_UNAUTHORIZED`	(401) Wymagana autoryzacja klienta
`SC_FORBIDDEN`	(403) Odmowa dostępu
`SC_NOT_FOUND`	(404) Dokument nie znaleziony
`SC_GONE`	(410) Dokument zmienił adres, a nowa lokalizacja nie jest znana
`SC_SERVICE_UNAVAILABLE`	(503) System jest chwilowo przeciążony lub niedostępny
`SC_NOT_MODIFIED`	(304) Dokument nie uległ modyfikacji od poprzedniego dostępu

Wykorzystanie HttpServletResponse

```
if (!request.getRemoteAddr().equals("150.254.31.178"))
  response.sendError(HttpServletResponse.SC_FORBIDDEN);
else
  out.println("<h1>Welcome!</h1>");
```



Problem polskich znaków (1)

- Jeżeli serwlet będzie generować dokumenty zawierające znaki z polskich stron kodowych, to wywołanie metody `setContentType` należy zapisać w następujący sposób:

```
response.setContentType("text/html;charset=windows-1250");
```

lub:

```
response.setContentType("text/html;charset=iso-8859-2");
```

Problem polskich znaków (2)

- Niezależnie od ustawień kodowania znaków narodowych, wartości wprowadzane do pól formularza są reprezentowane przy użyciu kodów ISO-8859-**1**
- W celu poprawnego przetwarzania wartości zawierających polskie znaki, programista musi przekodować otrzymane wartości tekstowe na WINDOWS-1250 lub ISO-8859-2:

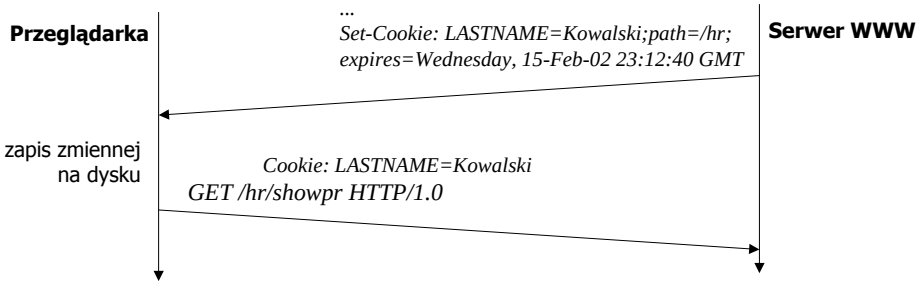
```
String par1_pl;
par1 = new String(
  request.getParameter("par1").getBytes("ISO8859_1"),"ISO8859_2");
```

lub:

```
String par1_pl;
par1 = new String(
  request.getParameter("par1").getBytes("ISO8859_1"),"WINDOWS-1250");
```

Zmienne Cookies

- Serwer WWW może w nagłówku HTTP odpowiedzi umieścić dane znakowe (zmienne Cookies), które przeglądarka zapisze w swojej pamięci (i ewentualnie na dysku)
- Przy ponownych odwołaniach przeglądarki do tego samego serwera WWW, w nagłówkach HTTP żądań będą każdorazowo odsyłane zapisane wcześniej zmienne Cookies



Własności zmiennych Cookies

- Nazwa i wartość
  - nazwa i wartość zmiennej Cookie; są to jedyne obowiązkowe elementy
- Data ważności
  - zmienne Cookies mogą być automatycznie usuwane przez przeglądarkę po upływie określonego czasu
- Adres domenowy
  - przeglądarka umieszcza w nagłówku żądania HTTP tylko te zmienne Cookies, których adres domenowy pokrywa się z końcówką adresu domenowego w URL żądania
- Ścieżka
  - przeglądarka umieszcza w nagłówku żądania HTTP tylko te zmienne Cookies, których ścieżka stanowi prefiks ścieżki URL danego żądania

Definiowanie zmiennych Cookies

- Klasa `javax.servlet.http.Cookie` służy do definiowania zmiennych Cookies, które będą wysyłane lub odbierane od przeglądarki

`Cookie(n,s)`	Tworzy zmienną Cookie o nazwie *n* i wartości *s*
`String getDomain()` `void setDomain(s)`	Odczytuje/ustawia adres domenowy, dla którego przeznaczona jest zmienna Cookie
`String getMaxAge()` `void setMaxAge(v)`	Odczytuje/ustawia czas życia zmiennej Cookie, liczony w sekundach (-1 -> ulotne)
`String getName()` `void setName(s)`	Odczytuje/ustawia nazwę zmiennej Cookie
`String getPath()` `void setPath(s)`	Odczytuje/ustawia prefiks ścieżki URL na serwerze, dla której przeznaczona jest zmienna Cookie
`String getValue()` `void setValue(s)`	Odczytuje/ustawia wartość zmiennej Cookie

Wysyłanie zmiennych Cookies przy użyciu `HttpServletResponse.addCookie()`

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html; charset=iso-8859-2");
   PrintWriter out = response.getWriter();

   Cookie myCookie = new Cookie("cookie1","James Bond");
   myCookie.setMaxAge(24*60*60);
   response.addCookie(myCookie);

   out.println("<h1>Wysłano Cookie</h1>");
  }
}
```

Odczytywanie zmiennych Cookies przy użyciu `HttpServletRequest.getCookies()`

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends

   public void doGet(HttpServl
                     HttpServl
      throws ServletException,

   response.setContentType("text/html; charset=iso-8859-2");
   PrintWriter out = response.getWriter();

   out.println("<h1>Odebrane Cookies</h1>");

   Cookie[] allCookies = request.getCookies();
   for (int i=0; i<allCookies.length; i++)
     out.println(allCookies[i].getName()+": "+
                 allCookies[i].getValue()+"<br>");
  }
}
```

Interfejs HttpSession

- Protokół HTTP jest bezstanowy - nie potrafi rozpoznać, że nowe żądanie pochodzi od użytkownika, który już wcześniej przesłał inne żądanie
- Interfejs `javax.servlet.http.HttpSession` implementuje pojęcie sesji - umożliwia identyfikację użytkownika, który wielokrotnie odwołuje się do serwletu/serwletów
- Nieużywana sesja wygasa automatycznie

`Object getAttribute(n)` `void setAttribute(n,o)`	Zapamiętuje na czas sesji obiekt pod podaną nazwą / odczytuje obiekt o podanej nazwie
`long getCreationTime()`	Odczytuje czas rozpoczęcia sesji, liczony w ms od 1.01.1970
`String getId()`	Odczytuje jednoznaczny identyfikator sesji
`long getLastAccessedTime()`	Odczytuje czas ostatniej operacji wykonanej w ramach sesji (1.01.1970)
`void invalidate()`	Zamyka sesję i usuwa wszystkie jej obiekty

Obsługa sesji przy użyciu `HttpServletRequest.getSession()`

```
// uwaga: wymaga wersji 2.2 biblioteki javax.servlet
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("<h1>Rozpoczęcie sesji</h1>");
   HttpSession mySes = request.getSession(true);
   mySes.setAttribute("username", "James Bond");

  }
}
```

Obsługa sesji przy użyciu `HttpServletRequest.getSession()`

```
// uwaga: wymaga wersji 2.2 biblioteki javax.servlet
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet2 extends HttpServlet {

   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("<h1>Stan sesji</h1>");
   HttpSession mySes = request.getSession(false);
   out.println((String) mySes.getAttribute("username"));
  }
}
```

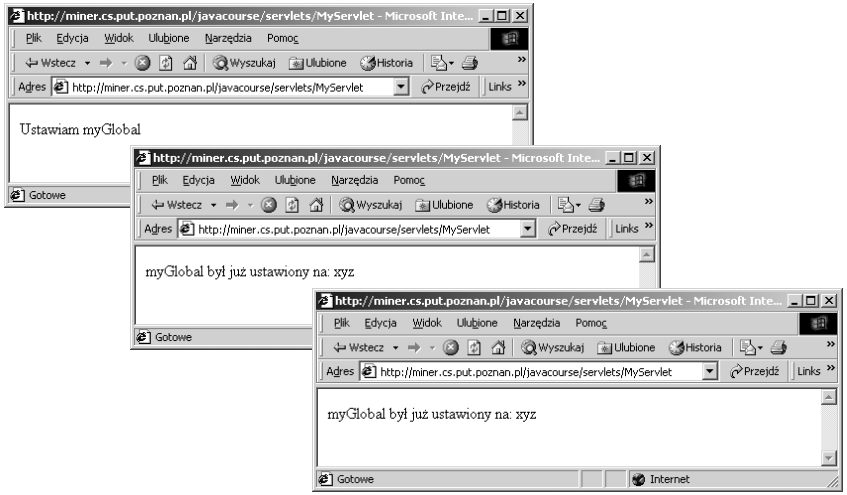
Serwlety - pozostałe funkcje

- Gdy otrzymanych będzie wiele żądań wykonania tego samego serwletu, to będą one realizowane przez oddzielne wątki w ramach jednego obiektu serwletu - oznacza to współdzielenie zmiennych globalnych (zmienne metod są prywatne)
- Serwlet może zapisywać komunikaty tekstowe do pliku logu serwera
- Serwlet może skorzystać z dodatkowych parametrów inicjalizacyjnych, zapisanych w pliku konfiguracyjnym serwera

Współdzielenie zmiennych globalnych przez wątki jednego serwletu (1)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   String myGlobal = null;

   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html; charset=iso-8859-2");
   PrintWriter out = response.getWriter();

   if (myGlobal==null) {
    out.println("Ustawiam myGlobal");
    myGlobal = "xyz";
   } else {
    out.println("myGlobal był już ustawiony na:");
    out.println(myGlobal);}
}}
```

Współdzielenie zmiennych globalnych przez wątki jednego serwletu (1)



Zapisy do pliku logu serwera

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   if (request.getParameter("id")==null) {
      log("Błąd parametru wywołania");
      out.println("Podaj parametr wywołania id");}
   else  out.println("Otrzymano:"+request.getParameter("id"));
  }
}
```

jserv.log:

```
...
[13/02/2002 18:37:14:849 CET] MyServlet: Błąd parametru wywołania
```

Odczyt parametrów inicjalizacyjnych

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
   String msg;
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println(msg);
   }
   public void init(ServletConfig config) {
    msg = config.getInitParameter("welcome_message");
   }
}
```

zone.properties:

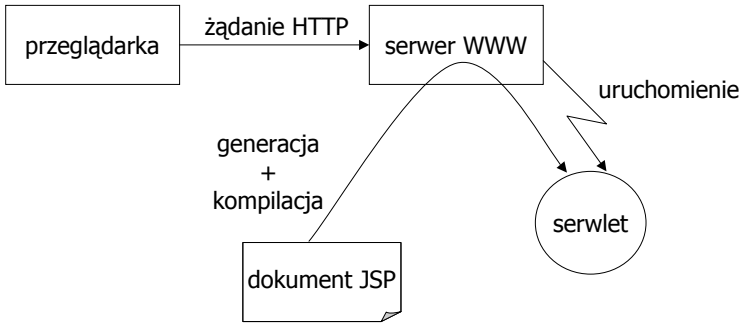
```
...
servlet.MyServlet.initArgs=welcome_message=Welcome Everybody!
```

Tworzenie aplikacji JavaServer Pages

Czym są aplikacje JavaServer Pages?

- Podstawowa wada serwletów Java: brak możliwości oddzielenia kodu w języku Java generującego dynamiczną zawartość dokumentu od części statycznej (HTML) odpowiedzialnej za ogólny wygląd strony
- Technologia JavaServer Pages (JSP) jest rozszerzeniem architektury serwletów Java
- Aplikacje JavaServer Pages mają postać dokumentów HTML (XML) zawierających zagnieżdżony kod w języku Java
- W odpowiedzi na żądanie użytkownika, serwer WWW wykonuje zagnieżdżony kod i zwraca wygenerowany dokument HTML
- Typowy sposób przetwarzania dokumentów JSP polega na ich translacji do serwletów, które są następnie kompilowane i uruchamiane przez serwer WWW

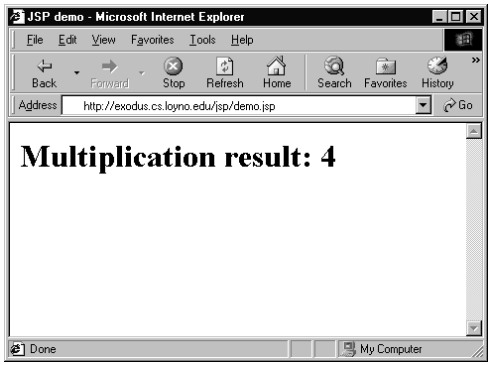
Architektura JSP



Przy pierwszym żądaniu, na podstawie treści dokumentu JSP generowany jest kod źródłowy równoważnego serwletu. Serwlet ten jest następnie kompilowany i uruchamiany, po czym pozostaje w pamięci operacyjnej na rzecz obsługi ponownych żądań.

JSP - Przykład

```
<HTML>
<HEAD>
<TITLE>JSP demo</TITLE>
</HEAD>
<BODY>
<%@ page language="java" %>
<%! int result; %>
<% result = 2*2; %>
<H1> Multiplication result:
<%= result %>
</H1>
</BODY>
</HTML>
```



Porównanie równoważnych: serwletu i JSP

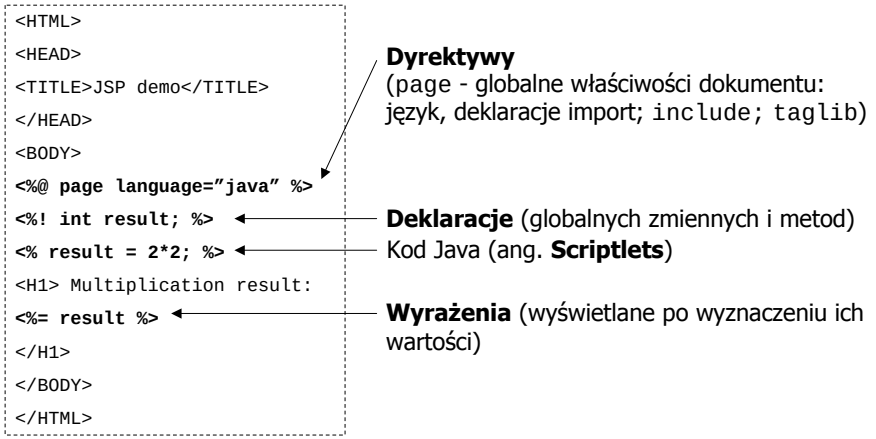
JSP

```
<HTML>
<HEAD>
<TITLE>JSP demo</TITLE>
</HEAD>
<BODY>
<%@ page language="java" %>
<%! int result; %>
<% result = 2*2; %>
<H1> Multiplication result:
<%= result %>
</H1>
</BODY>
</HTML>
```

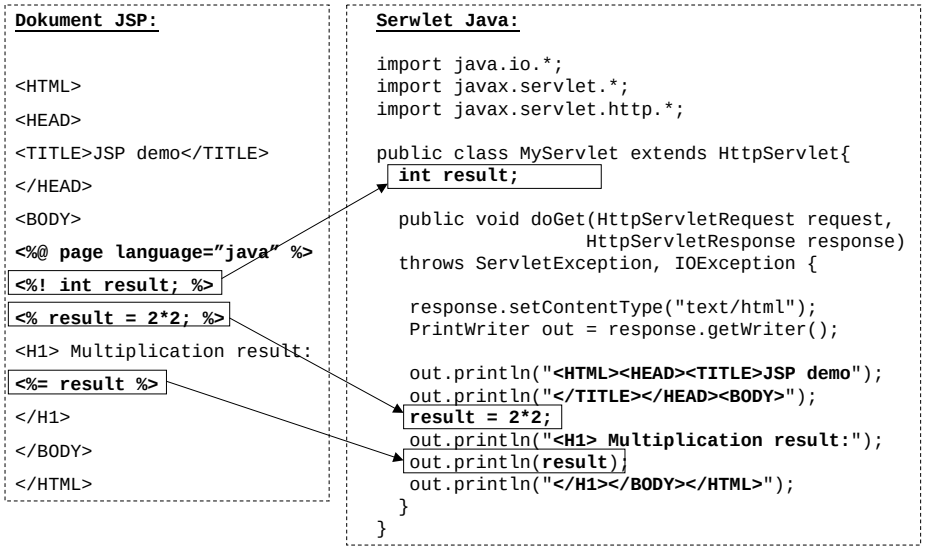
Serwlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
   int result;
   public void doGet(
     HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException {
 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
 out.println("<HTML>");
 out.println("<HEAD>");
 out.println("<TITLE>JSP demo</TITLE>");
 out.println("</HEAD>");
 out.println("<BODY>");
 result = 2*2;
 out.println(" <H1> Multiplication result:");
 out.println(result);
 out.println("</H1>");
 out.println("</BODY>");
 out.println("</HTML>");
}}
```

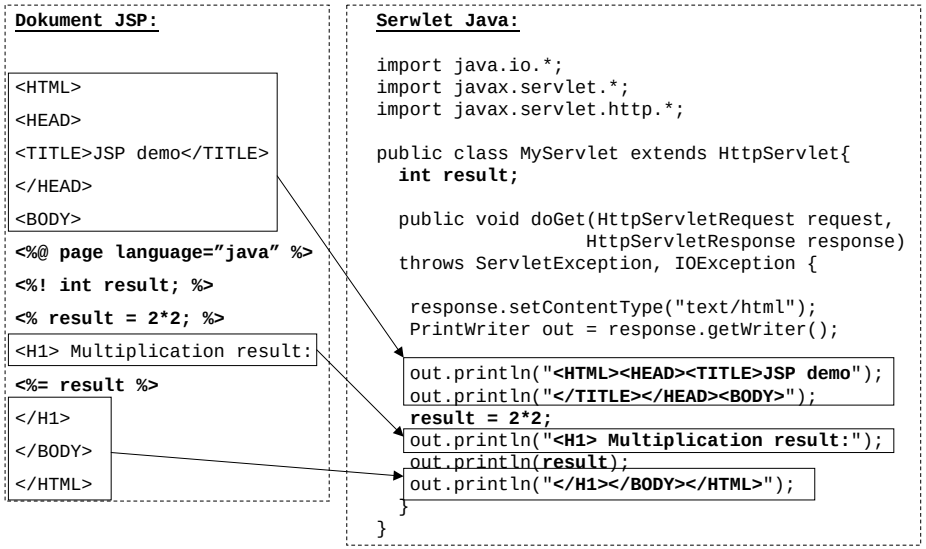
Komponenty dokumentu JSP



Generacja serwletu (1)



Generacja serwletu (2)



Obiekty dostępne w dokumentach JSP (1)

- Programista JSP może korzystać z nastepujących obiektów, tworzonych dla niego automatycznie:
  - `request (javax.servlet.http.HttpServletRequest)`
    Umożliwia dostęp do pól w treści żądania otrzymanego od przeglądarki (jak w metodach `doGet()/doPost()` serwletu)
  - `response (javax.servlet.http.HttpServletResponse)`
    Umożliwia dostęp do pól nagłówka odpowiedzi przesyłanej do przeglądarki (jak w metodach `doGet()/doPost()` serwletu)
  - `page (javax.servlet.jsp.HttpJspPage)`
    Reprezentuje obiekt serwletu, który powstał na podstawie tego dokumentu (`this`)
  - `session (javax.servlet.http.HttpSession)`
    Reprezentuje obiekt sesji użytkownika (jak `getSession()` w serwlecie)

Obiekty dostępne w dokumentach JSP (2)

Programista JSP może korzystać z nastepujących obiektów, tworzonych dla niego automatycznie:

- `application (javax.servlet.ServletContext)`

  Reprezentuje obiekt, który jest współdzielony przez wszystkie dokumenty JSP obsługiwane przez tę samą JVM
- `out (javax.servlet.jsp.JspWriter)`

  Reprezentuje kanał wyjściowy dla zapisu do przeglądarki (jak `getWriter()` w serwlecie)
- `config (javax.servlet.ServletConfig)`

  Umożliwia dostęp do parametrów inicjalizacyjnych serwletu (jak w metodzie `init()` serwletu)

Problem polskich znaków

- Aby aplikacja JSP poprawnie wyświetlała zawarte w niej znaki narodowe, należy zastosować dyrektywę wskazującą zastosowany standard kodowania:

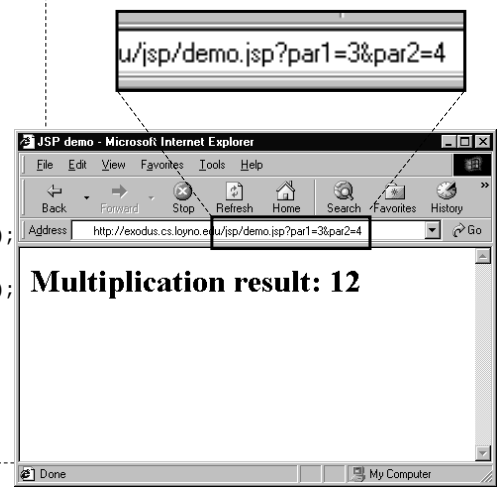
```
<%@ page contentType="text/html; charset=iso-8859-2" %>
```

Przekazywanie parametrów

- Aplikacji JSP mogą być przekazane parametry wejściowe
- Parametry mogą być umieszczone w adresie URL lub wysyłane przez formularz HTML
- Implementacja mechanizmu przekazywania parametrów jest taka sama jak w przypadku serwletów: dostęp do wszystkich parametrów wywołania jest możliwy poprzez obiekt `request`
- Aby odczytać wartość parametru należy wywołać metodę `request.getParameter()`

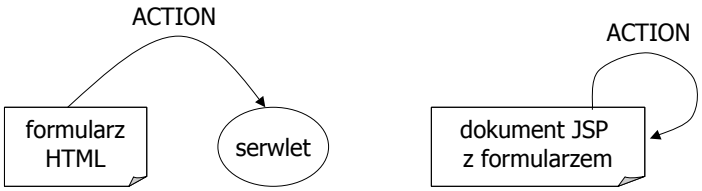
Przekazywanie parametrów - Przykład

```
<HTML>
<HEAD>
<TITLE>JSP demo</TITLE>
</HEAD>
<BODY>
<%@ page language="java" %>
<%  int result;
    int term1, term2; %>
<% term1 = Integer.parseInt(
        request.getParameter("par1"));
   term2 = Integer.parseInt(
        request.getParameter("par2"));
   result = term1 * term2; %>
<H1> Multiplication result:
<%= result %>
</H1>
</BODY>
</HTML>
```



Przekazywanie parametrów do JSP za pomocą formularzy HTML

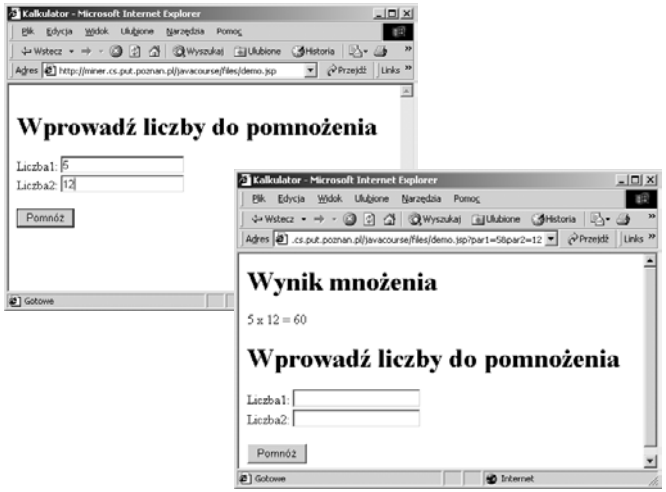
- Technologia JSP pozwala na zintegrowanie formularza HTML z programem obsługi tego formularza
- Jeżeli znacznik <FORM>, definiujący formularz, nie określi wartości atrybutu ACTION, wtedy aplikacją obsługującą formularz będzie ta sama, która ten formularz wyświetliła



Przykład zintegrowanej aplikacji JSP (1)

```
<HTML>
 <HEAD><TITLE>Kalkulator</TITLE></HEAD>
 <BODY>
  <%@ page contentType="text/html; charset=iso-8859-2" %>
  <% int liczba1, liczba2;
   if (request.getParameter("par1")!=null){
     liczba1 = Integer.parseInt(request.getParameter("par1"));
     liczba2 = Integer.parseInt(request.getParameter("par2")); %>
    <H1>Wynik mnożenia</H1>
    <%= liczba1 %> x <%= liczba2 %> = <%= liczba1*liczba2 %> <BR>
  <% } %><BR>
  <H1>Wprowadź liczby do pomnożenia</H1>
  <FORM>
   Liczba1:<INPUT TYPE='text' NAME='par1'><BR>
   Liczba2:<INPUT TYPE='text' NAME='par2'><BR><BR>
   <INPUT TYPE='submit' VALUE='Pomnóż'>
  </FORM>
 </BODY>
</HTML>
```

Przykład zintegrowanej aplikacji JSP (2)



Wykorzystywanie zewnętrzych klas Java

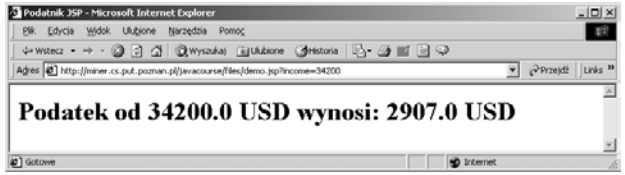
- Kod w języku Java zagnieżdżony w dokumencie JSP może osiągać duże rozmiary utrudniając tym samym pielęgnację aplikacji
- Zaleca się, aby programiści implementowali złożony kod w oddzielnej klasie/klasach Java, do której następnie w prosty sposób odwoływać się może dokument JSP; pozwala to na zmniejszenie rozmiaru dokumentów JSP
- Obiekty tworzone wewnątrz "<%" i "%>" posiadają zasięg ograniczony do jednego wywołania, natomiast zasięg obiektów tworzonych przy pomocy znacznika `<jsp:useBean/>` może być jawnie wyspecyfikowany przez programistę
- Omawiane klasy zewnętrzne są często nazywane JavaBeans

Klasy zewnętrzne - przykład (1)

```
public class TaxCalc {
 public double getTax(double income) {

  double tax;

  if (income < 2450)
    tax = 0.05 * income;
  else if (income < 6100)
    tax = (0.07*(income-2450)+123);
  else
    tax = (0.09*(income-6100)+378);
  return tax;
  }
}
```

Klasy zewnętrzne - przykład (2)

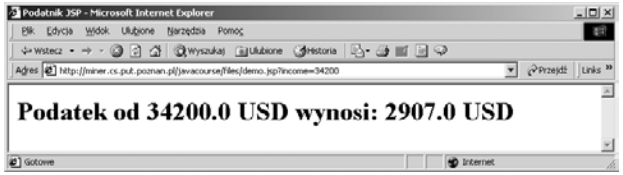
```
<%@ page import="TaxCalc" %>
<HEAD>
 <TITLE>Podatnik JSP</TITLE>
</HEAD>
<BODY>
 <% TaxCalc tc = new TaxCalc();
    double t, i;
    i = Double.parseDouble(request.getParameter("income"));
    t = tc.getTax(i);
  %>

 <H1> Podatek od <%= i %> USD wynosi: <%= t %> USD </H1>
</BODY>
```



Korzystanie z klas zewnętrznych przy pomocy <jsp:useBean/>

```
<HEAD>
 <TITLE>Podatnik JSP</TITLE>
</HEAD>
<BODY>
 <jsp:useBean id="tc" class="TaxCalc"/>
 <% double t, i;

    i = Double.parseDouble(request.getParameter("income"));
    t = tc.getTax(i);
 %>

 <H1> Podatek od <%= i %> USD wynosi: <%= t %> USD </H1>
</BODY>
```

pamiętaj o "/"!



Zasięg obiektów tworzonych przy pomocy <jsp:useBean/>

- Zasięg obiektu utworzonego na podstawie klasy zewnętrznej można określić jawnie za pomocą atrybutu `scope` znacznika `<jsp:useBean/>`
- Dopuszczalne wartości atrybutu `scope`:
  - `page` (zasięg ograniczony do bieżącego dokumentu JSP)
  - `request` (zasięg ograniczony do obsługi bieżącego żądania)
  - `session` (zasięg ograniczony do całej sesji użytkownika)
  - `application` (współdzielenie przez wszystkie aplikacje pracujące na tej samej JVM)
- Korzystanie z obiektów o zasięgu większym niż bieżący dokument pozwala na:
  - oszczędniejsze gospodarowanie zasobami
  - współdzielenie obiektów przez dokumenty obsługujące to samo żądanie

Zasięg obiektów - przykład

```
<BODY>
 <jsp:useBean id="username" class="LongClass" scope="session"/>
 <% username.str = "James Bond"; %>
 Ustawiono username: <%= username.str %>
</BODY>
```


```
<BODY>
 <jsp:useBean id="username" class="LongClass" scope="session"/>
 Odczytano username: <%= username.str %>
</BODY>
```


```
public class LongClass {
 public String str;
}
```

JSP - pozostałe znaczniki standardowe

- `<jsp:include/>`
  - Umieszcza wewnątrz generowanej odpowiedzi zawartość innego dokumentu, np.:
    `<jsp:include page="/hr/news.jsp" flush="true" />`
    (flush="true" jest wymagane ze względu na bug)
- `<jsp:forward/>`
  - Powoduje natychmiastowe przekierowanie żądania pod inny adres, np.:
    `<jsp:forward page="/hr/newmain.jsp"/>`

Biblioteki znaczników JSP

- Standard JSP 1.1 umożliwia pozwala programistom na rozszerzanie zbioru podstawowych znaczników JSP o znaczniki implementujące dodatkowe funkcje przydatne podczas konstrukcji aplikacji
- Znaczniki definiowane przez programistę są zapisywane w bibliotekach znaczników (Tag Libraries), a następnie wykorzystywane w taki sam sposób, jak znaczniki standardowe
- W porównaniu z klasami zewnętrznymi, biblioteki znaczników oferują podobną funkcjonalność, lecz są wygodniejsze i prostsze w użyciu
- Wielu producentów oferuje własne biblioteki znaczników

Przykład wykorzystania biblioteki znaczników (OracleJSP Tag Library for SQL)

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="jml" %>
<BODY>
  <H1>Lista plac:</H1>
  <jml:dbOpen
   url="jdbc:oracle:thin:@miner.cs.put.poznan.pl:1521:dmine"
   user="scott" password="tiger" connId="con1">
  </jml:dbOpen>
  <jml:dbQuery connId="con1">
     select empno as nr, ename as nazwisko,
            job as stanowisko, sal as pensja from emp
  </jml:dbQuery>
  <jml:dbClose connId="con1" />
 </BODY>
```

Lista plac:

NR	NAZWISKO	STANOWISKO	PENSJA
7369	SMITH	CLERK	800
7499	ALLEN	SALESMAN	1600
7521	WARD	SALESMAN	1250
7566	JONES	MANAGER	2975
7654	MARTIN	SALESMAN	1250
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7788	SCOTT	ANALYST	3000
7839	KING	PRESIDENT	5000
7844	TURNER	SALESMAN	1500
7876	ADAMS	CLERK	1100
7900	JAMES	CLERK	950
7902	FORD	ANALYST	3000
7934	MILLER	CLERK	1300

Wykorzystywanie bibliotek znaczników

- W celu poinformowania środowiska JSP o chęci wykorzystywania zewnętrznej biblioteki znaczników, należy umieścić wewnątrz dokumentu następującą deklarację:

```
<%@ taglib uri="/tlt" prefix="tlt" %>
```

gdzie:

`uri` - ścieżka dostępu do pliku TLD definiującego bibliotekę znaczników

`prefix` - wybrany przez użytkownika prefiks, który będzie używany w dokumencie do wyróżniania znaczników pochodzących z tej biblioteki

Biblioteka OracleJSP Tag Library for SQL

- Biblioteka OracleJSP Tag Library for SQL jest dołączana do wersji instalacyjnej serwera bazy danych Oracle8i i Oracle9i.
- Znaczniki:
  - `<dbOpen>`: otwiera połączenie z bazą danych
  - `<dbClose>`: zamyka połączenie z bazą danych
  - `<dbQuery>`: wykonuje zapytanie SQL
  - `<dbCloseQuery>`: zamyka otwarte zapytanie SQL
  - `<dbNextRow>`: przetwarza rekordy wynikowe zapytania SQL
  - `<dbExecute>`: wykonuje polecenie DML lub DDL

Składnia znaczników OracleJSP Tag Library for SQL

```
<sql:dbOpen[connId="connection-id"]
 user="username"
 password="password"
 URL="databaseURL" >
...
</sql:dbOpen>
```

```
<sql:dbClose connId="connection-id" />
```

```
<sql:dbQuery
[queryId="query-id"]
[connId="connection-id"]
[output="HTML|XML|JDBC"]>
... zapytanie SELECT ...
</sql:dbQuery>
```

`output=HTML/XML` spowoduje automatyczne wyświetlenie wyniku zapytania
`output=JDBC` umożliwi programiście przetwarzanie wyniku przy użyciu `<dbNextRow>`

```
<sql:dbCloseQuery
 queryId="query-id" />
```

```
<sql:dbNextRow
 queryId="query-id" >
... Przetwarzanie rekordu ...
</sql:dbNextRow >
```

dostęp do pól rekordu przy użyciu notacji:
`query_id.getString("kolumna")`

```
<sql:dbExecute
[connId="connection-id"]
[output="yes|no"]>
... DML lub DDL ...
</sql:dbExecute >
```

OracleJSP Tag Library for SQL - przykład zaawansowany

```
<%@ taglib uri="sqltaglib.tld" prefix="sql" %>
<%@ page contentType="text/html; charset=iso-8859-2" %>
<BODY>
 <sql:dbOpen URL="jdbc:oracle:thin:@miner.cs.put.poznan.pl:1521:dmine"
  user="javacourse" password="ploug" connId="con1">
 </sql:dbOpen>
 <sql:dbQuery connId="con1" output="jdbc" queryId="myquery">
  select start_date data, first_name||last_name klient, title tytuł
  from rentals, movies, customers
  where rentals.movie_id=movies.id
  and   rentals.customer_id=customers.id
  </sql:dbQuery>
 <table cellpadding=8 cellspacing=0>
  <sql:dbNextRow queryId="myquery">
   <tr>
    <td>
      <%= myquery.getString(1) %>
    <td>
      <%= myquery.getString(2) %>
    <td>
      <%= myquery.getString(3) %>
  </sql:dbNextRow>
  <sql:dbClose connId="con1" />
</BODY>
```

Tworzenie bibliotek znaczników

Definiowanie znaczników: rodzaje znaczników

- Znaczniki proste

```
<tlt:greeting />
```

- Znaczniki z atrybutami

```
<tlt:greeting size=1 />
```

- Znaczniki z zawartością

```
<tlt:greeting>
  to Poznan
</tlt:greeting>
```

- Znaczniki tworzące obiekty JSP

```
<tlt:lookup id="tx" type="UserTransaction" />
<% tx.begin(); %>
```

Definiowanie znaczników: algorytm

1. Utwórz klasy znaczników i klasy pomocnicze
2. Utwórz plik TLD (tag library descriptor)

Definicja prostego znacznika: przykład
Klasa obsługi znacznika

```
package examples;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;

public class SimpleTag extends TagSupport {
   public int doStartTag() throws JspException {
     try {
          pageContext.getOut().print("Hello!");
         }
     catch (Exception ex) {
          throw new JspTagException("SimpleTag: " +
                                    ex.getMessage());}
     return SKIP_BODY;
    }
   public int doEndTag() {
     return EVAL_PAGE;
   }
}
```

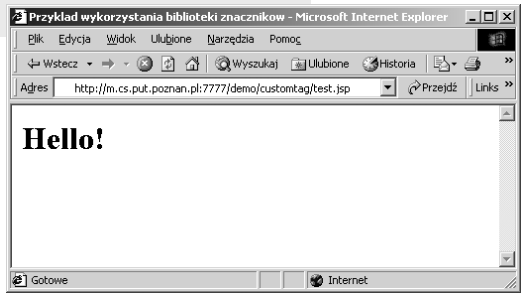
The tag displays Hello!

Definicja prostego znacznika: przykład
Plik TLD

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
  <!DOCTYPE taglib PUBLIC
        "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
  <info>Example tag library</info>
  <tag>
   <name>greetings</name>
   <tagclass>examples.SimpleTag</tagclass>
  </tag>
</taglib>
```

Definicja prostego znacznika: przykład
Użycie znacznika w aplikacji JSP

```
<%@ taglib uri="/WEB-INF/test.tld" prefix="jml" %>
<HTML>
 <HEAD>
   <TITLE>Tag library example </TITLE>
 </HEAD>
 <BODY>
  <H1><jml:greetings /></H1>
 </BODY>
</HTML>
```



Definicja znacznika z atrybutami: przykład
Klasa obsługi znacznika

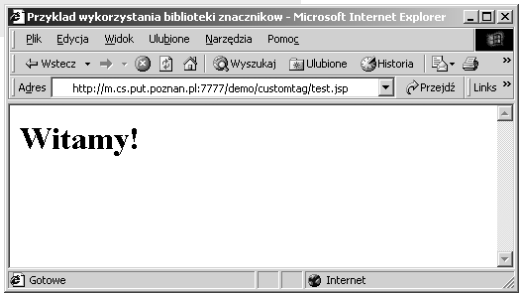
```
package examples;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;

public class SimpleTag extends TagSupport {
   private String selected_language;
   public int doStartTag() throws JspException {
      try {
          if (selected_language.equals("polish"))
            pageContext.getOut().print("Witamy!");
          else
            pageContext.getOut().print("Hello!");
          }
      catch (Exception ex) {
          throw new JspTagException("SimpleTag: " + ex.getMessage());}
      return SKIP_BODY;
    }
   public int doEndTag() {
      return EVAL_PAGE;
   }
   public void setLang(String l) {
      selected_language = l;
   }
}
```

Definicja znacznika z atrybutami: przykład
Plik TLD

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
  <!DOCTYPE taglib PUBLIC
        "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
  <info>Tag library example</info>
  <tag>
   <name>greetings</name>
   <tagclass>examples.SimpleTag</tagclass>
   <attribute>
      <name>lang</name>
      <required>no</required>
   </attribute>
   </tag>
</taglib>
```

Definicja znacznika z atrybutami: przykład
Użycie znacznika w aplikacji JSP

```
<%@ taglib uri="/WEB-INF/test.tld" prefix="jml" %>
<HTML>
 <HEAD>
   <TITLE>Tag library example </TITLE>
 </HEAD>
 <BODY>
  <H1><jml:greetings lang="polish"/></H1>
 </BODY>
</HTML>
```



Znacznik z zawartością: przykład
Klasa obsługi znacznika

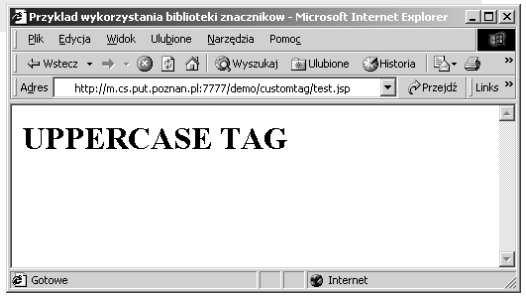
```
package examples;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;

public class AdvancedTag extends BodyTagSupport {
   public int doAfterBody() throws JspTagException {
      BodyContent bc = getBodyContent();
      String body_text = bc.getString();

      try {
        getPreviousOut().print(body_text.toUpperCase());
      }
      catch (Exception ex) {
          throw new JspTagException("AdvancedTag: " +
                      ex.getMessage());}
      return SKIP_BODY;
   }
}
```

Znacznik z zawartością: przykład
Plik TLD

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
  <!DOCTYPE taglib PUBLIC
        "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
  <info>Tag library example</info>
  <tag>
   <name>uppercase</name>
   <tagclass>examples.AdvancedTag</tagclass>
  </tag>
</taglib>
```

Znacznik z zawartością: przykład
Użycie znacznika w aplikacji JSP

```
<%@ taglib uri="/WEB-INF/test.tld" prefix="jml" %>
<HTML>
 <HEAD>
   <TITLE>Tag library example</TITLE>
 </HEAD>
 <BODY>
  <H1><jml:uppercase> uppercase tag </jml:uppercase></H1>
 </BODY>
</HTML>
```



Dostęp do baz danych z programów Java

Standardy JDBC i SQLJ

Czym jest JDBC ?

- JDBC jest standardowym interfejsem do współpracy aplikacji Java z relacyjną bazą danych
- JDBC definiuje standardowe interfejsy
  - interfejsy są implementowane przez sterowniki JDBC
  - standard dopuszcza rozszerzenia producentów
- Zaprojektowany na podstawie X/Open SQL Call Level Interface (podobny do ODBC)
- Interfejs dla dynamicznych instrukcji SQL
- Zgodny ze standardem SQL 92
- Opracowany przez Sun Microsystems
- Powszechnie wspierany przez producentów systemów baz danych i narzędzi programistycznych

JDBC – Historia standardu

- JDBC 1.0
  - podstawowa obsługa baz danych
  - część JDK 1.1
- JDBC 2.0
  - wsparcie dla aplikacji Java intensywnie korzystających z baz danych
  - JDBC 2.0 core API – część Java 2 SDK, Standard Edition
  - JDBC 2.0 Optional Package (Standard Extension) – część Java 2 SDK, Enterprise Edition; integralna część technologii Enterprise JavaBeans
- JDBC 3.0
  - różne rozszerzenia funkcjonalne
  - JDBC 3.0 core API i Optional Package - część J2SE 1.4

Czym jest SQLJ ?

- Dotyczy integracji technologii baz danych z językiem Java:
  - zagnieżdżania instrukcji SQL w kodzie Java (statyczny SQL)
  - wykorzystywania statycznych metod Java jako składowanych procedur i funkcji SQL
  - używania czystych klas Java jako typów danych SQL
- Wynik współpracy firm Oracle, Sybase, IBM, Tandem (obecnie część Compaq), JavaSoft i Informix
- Powszechnie implementowany: Oracle (8i/9i Server, JDeveloper, ...), IBM, Sybase, ...

SQLJ – Historia standardu

- JSQL (Oracle, IBM, Tandem)
  - SQL zagnieżdżony w języku Java
- SQLJ (Oracle, IBM, Compaq/Tandem, Sybase, Informix)
  - SQLJ Część 0: Zagnieżdżony SQL (standard ANSI – 1998, standard ISO – 2000)
  - SQLJ Część 1: Procedury składowane w języku Java (standard ANSI – 1999)
  - SQLJ Część 2: Klasy Java jako typy SQL (standard ANSI – 2000)

JDBC, SQLJ – Przykłady

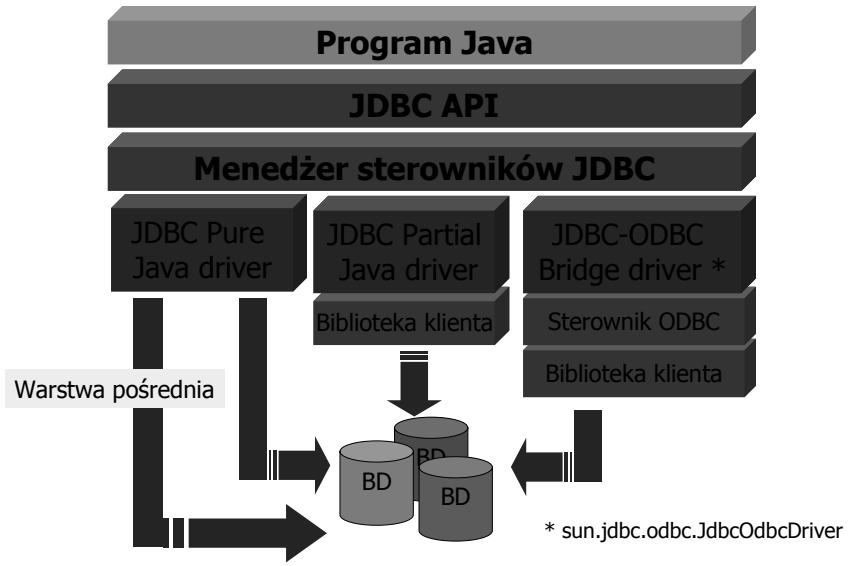
- Sparametryzowana instrukcja SQL w JDBC:

```
int empNo = 200;
PreparedStatement ps = conn.prepareStatement(
    "UPDATE emp SET sal = sal * 1.1 WHERE empno = ?");
ps.setInt(1, empNo);
ps.executeUpdate();
ps.close();
```

- Sparametryzowana instrukcja SQL w SQLJ:

```
int empNo = 200;
#sql { UPDATE emp SET sal = sal * 1.1 WHERE empno = :empNo };
```

Architektura JDBC



Typy sterowników JDBC

- Typ I – Most JDBC-ODBC
  - umożliwia połączenie z każdą bazą danych, dla której istnieje sterownik ODBC
- Typ II – Sterownik napisany częściowo w Javie, wykorzystujący biblioteki klienta bazy danych
  - efektywne rozwiązanie
  - wymaga preinstalowanego oprogramowania klienta bazy danych
- Typ III – Uniwersalny sterownik w czystej Javie, z obsługą specyficznych baz danych w warstwie pośredniej
  - najbardziej elastyczna architektura
- Typ IV – Sterownik w czystej Javie, komunikujący się bezpośrednio z serwerem bazy danych
  - nie wymaga bibliotek klienta bazy danych
  - odpowiedni dla dostępu do bazy danych z apletów

Sterowniki JDBC Oracle

- JDBC Thin (typ IV)

```
oracle.jdbc.OracleDriver
```

  - w 100% napisany w czystej Javie
  - może być pobrany przez sieć wraz z apletem Java
- JDBC OCI (typ II)
  - wykonuje wywołania OCI do "fabrycznego" sterownika, preinstalowanego po stronie klienta
  - wykorzystywany wyłącznie w aplikacjach języka Java
- Server-side internal driver
  - wykorzystywany przez aplikacje Java uruchamiane wewnątrz serwera Oracle (np. Java Stored Procedures)
- Server-side Thin driver
  - wykorzystywany przez aplikacje Java uruchamiane wewnątrz serwera Oracle do nawiązywania połączeń z innymi serwerami

Sterowniki JDBC IBM

- IBM DB2 Universal Database
  - sterownik dla aplikacji po stronie serwera (typ II)

```
COM.ibm.db2.jdbc.app.DB2Driver
```

  - sterownik dla apletów (typ III)
    - komunikuje się z bazą danych poprzez serwer apletów (ang. applet server)

```
COM.ibm.db2.jdbc.net.DB2Driver
```

- IBM AS/400 Database
  - IBM AS/400 "native" JDBC driver (typ II)
    - Korzysta z SQL CLI – ograniczony do platformy AS/400

```
com.ibm.db2.jdbc.app.DB2Driver
```

  - IBM AS/400 "toolbox" JDBC driver (typ IV)
    - 100% napisany w czystej Javie

```
com.ibm.as400.access.AS400JDBCDriver
```

Funkcjonalność JDBC 1.0 API

- Otwieranie połączeń z bazami danych
- Uzyskiwanie informacji o możliwościach serwera bazy danych
- Wykonywanie instrukcji DML i DDL
- Przetwarzanie wyników zapytań
- Prekompilowane, sparametryzowane polecenia SQL
- Wywoływanie podprogramów składowanych
- Transakcje (zatwierdzanie, wycofywanie)

Podstawowe klasy JDBC (pakiet java.sql)

- **`DriverManager`**
- **`Connection`**
- **`Statement`**
- **`PreparedStatement`**
- **`CollableStatement`**
- **`ResultSet`**
- **`DatabaseMetaData`**
- **`ResultSetMetaData`**
- **`SQLException`**

Rejestrowanie sterowników JDBC (1)

- Sterowniki JDBC muszą się rejestrować w menedżerze sterowników (**`DriverManager`**)
- Sterowniki rejestrują się automatycznie podczas ich ładowania (w przypadku większości JVM)

```
try {
  // Oracle JDBC driver (Thin + OCI)
  Class.forName("oracle.jdbc.OracleDriver");
  // IBM DB2 Universal Database "app" driver
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
  // JDBC-ODBC bridge driver
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException e) { ... }
```

Rejestrowanie sterowników JDBC (2)

- W przypadku niektórych JVM załadowanie klasy nie rejestruje automatycznie sterownika (!)
- JDK 1.1.x na niektórych platformach (AIX, OS/2)

```
Class.forName("oracle.jdbc.OracleDriver").newInstance();
```

- Maszyny wirtualne Java firmy Microsoft

```
try {
  DriverManager.registerDriver(
    new oracle.jdbc.OracleDriver());
}
catch (SQLException e) { ... }
```

Nawiązywanie połączenia z bazą danych

- Menedżer sterowników zarządza sterownikami JDBC i służy do otwarcia połączenia z bazą danych
- Baza danych jest wskazywana przez podanie JDBC URL, identyfikującego sterownik JDBC i bazę danych:
  `jdbc:<subprotocol>:<connectString>`
- Na podstawie JDBC URL menedżer sterowników (**`DriverManager`**) wybiera odpowiedni sterownik JDBC spośród zarejestrowanych sterowników

Format JDBC URL dla różnych sterowników JDBC

JDBC-ODBC Bridge	**`jdbc:odbc:<odbc_data_source>`**
Oracle OCI	**`jdbc:oracle:oci:@<service_name>`**
Oracle Thin	**`jdbc:oracle:thin:@<host>:<port>:<sid>`**
IBM DB2 "app"	**`jdbc:db2:<db_name>`**
IBM DB2 "net"	**`jdbc:db2://<host>:<port>/<db_name>`**
IBM AS/400 "native"	**`jdbc:db2:<db_name>`**
IBM AS/400 "toolbox"	**`jdbc:as400://<system>`**

Podłączanie się do Oracle - Przykłady

JDBC-ODBC Bridge

```
Connection conn;
try {
   conn = DriverManager.getConnection("jdbc:odbc:Finance",
                                      "scott","tiger");
} catch (SQLException e) {…}
```

Oracle OCI

```
Connection conn;
try {
   conn = DriverManager.getConnection("jdbc:oracle:oci:@findb",
                                      "scott","tiger");
} catch (SQLException e) {…}
```

Oracle Thin

```
Connection conn;
try {
   conn = DriverManager.getConnection(
      "jdbc:oracle:thin:@host1:1521:orcl", "scott","tiger");
} catch (SQLException e) {…}
```

Podłączanie się do DB2 - Przykłady

JDBC-ODBC Bridge

```
Connection conn;
try {
   conn = DriverManager.getConnection("jdbc:odbc:Finance",
                                      "scott","tiger");
} catch (SQLException e) {…}
```

"app" driver

```
Connection conn;
try {
   conn = DriverManager.getConnection("jdbc:db2:Finance",
                                      "scott","tiger");
} catch (SQLException e) {…}
```

"net" driver

```
Connection conn;
try {
   conn = DriverManager.getConnection(
     "jdbc:db2://host1:2001/Finance ", "scott","tiger");
} catch (SQLException e) {…}
```

Wyjątek SQLException

- Wywołania JDBC mogą generować wyjątek **`java.sql.SQLException`**
- Metody zawierające wywołania JDBC muszą obsługiwać ten wyjątek lub deklarować możliwość jego generowania
- Wyjątek **`SQLException`** niesie następujące informacje:
  - kod "SQL state" (zgodny ze specyfikacją XOPEN SQL)
  - tekstowy komunikat o błędzie
  - numeryczny kod błędu (specyficzny dla danego DBMS)

```
try {
   conn = DriverManager.getConnection("jdbc:odbc:Finance",
                                      "scott","tiger");
} catch (SQLException e) {
  System.err.println("Error: " + e);
  String sqlState = e.getSQLState();
  String message = e.getMessage();
  int errorCode = e.getErrorCode();
  ...
}
```

Pobieranie metadanych dotyczących bazy danych

- Metadane opisujące bazę danych można odczytać w postaci obiektu **`DatabaseMetaData`** za pomocą obiektu reprezentującego bieżące połączenie

```
Connection conn; …
try {
  DatabaseMetaData dm = conn.getMetaData();
  String  s1 = dm.getURL();
  String  s2 = dm.getSQLKeywords();
  boolean b1 = dm.supportsTransactions();
  boolean b2 = dm.supportsSelectForUpdate();
}
catch (SQLException e) {…}
```

Polecenia SQL w JDBC

- Polecenie **`Statement`**
  - wykonywanie zapytań lub operacji DML/DDL
- Polecenie **`PreparedStatement`** (wywiedzione z **`Statement`**)
  - wykonywanie poleceń prekompilowanych
  - możliwość zaszycia zmiennych
  - przydatne gdy to samo polecenie jest wykonywane kilkukrotnie dla różnych wartości
- Polecenie **`CallableStatement`** (wywiedzione z **`PreparedStatement`**)
  - wywoływanie procedur i funkcji składowanych w bazie danych
  - zachowana możliwość zaszywania zmiennych

Klasa Statement (1)

- Wykonywanie zapytań (metoda **`executeQuery()`** zwracająca zbiór wynikowy **`ResultSet`**)

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery
                  ("SELECT ename, sal FROM emp");
...
} catch (SQLException e) {…}
```

- Wykonywanie poleceń DML i DDL (metoda **`executeUpdate()`**)

```
try {
  Statement stmt = conn.createStatement();
  stmt.executeUpdate
            ("DELETE FROM emp WHERE empno = 3981");
} catch (SQLException e) {…}
```

Klasa Statement (2)

- Metoda **`execute()`** wykorzystywana dla poleceń, których natura nie jest znana (SELECT, UPDATE, ...) lub zwracających więcej niż jeden zbiór wynikowy **`ResultSet`**)
  - sprawdzenie czy instrukcja zwróciła liczbę modyfikacji: **`int getUpdateCount()`**
  - zwrócenie bieżącego zbioru wynikowego: **`ResultSet getResultSet()`**
  - przejście do kolejnego ze zwróconych zbiorów wynikowych: **`boolean getMoreResults()`**

Przetwarzanie zbiorów wynikowych

- Obiekt **`ResultSet`** przechowuje tabelę danych wynikowych zwróconych przez zapytanie SQL
- Obsługa kursorów
  - kursor startuje od pozycji przed pierwszym rekordem zbioru wynikowego
  - metoda **`next()`** przesuwa kursor na następny rekord (zwraca **`true`**, gdy znaleziono kolejny rekord)
- Dane odczytuje się przy pomocy metod **`getXXX()`** np. **`getInt()`**, **`getString()`**, które odwzorowują wyniki w równoważne typy języka Java
- Dostęp do pól bieżącego rekordu odbywa się przez numer na liście wyrażeń w zapytaniu lub nazwę atrybutu

Przetwarzanie zbiorów wynikowych - Przykłady

```
try {
  ResultSet rset = stmt.executeQuery(
                "SELECT ename, sal FROM emp");
  while (rset.next()) {
    String ename   = rset.getString(1);
    int sal = rset.getInt(2);
  }
} catch (SQLException e) {…}
```

```
try {
  ResultSet rset = stmt.executeQuery(
                "SELECT ename, sal FROM emp");
  while (rset.next()) {
    String ename   = rset.getString("ENAME");
    int sal = rset.getInt("SAL");
  }
} catch (SQLException e) {…}
```

JDBC – kompletny przykład

```
import java.sql.*;

public class MyDemo {
   public static void main(String[] args)
   throws ClassNotFoundException, SQLException {
     Connection conn;
     Statement stmt;
     ResultSet rset;

     DriverManager.registerDriver(
         new oracle.jdbc.driver.OracleDriver());
     conn = DriverManager.getConnection(
        "jdbc:oracle:thin:@srv1:1521:orcl","scott","tiger");
     stmt = conn.createStatement();
     rset = stmt.executeQuery("select ename,sal from emp");
     while (rset.next()) {
        System.out.print(rset.getString("ename")+" ");
        System.out.println(rset.getString("sal"));
     }
     rset.close(); stmt.close(); conn.close();
}}
```

Konwersje typów między SQL i Java

- Dla każdego typu SQL istnieje jeden lub więcej typów Java, do których możliwa jest konwersja:

  CHAR -> String, int, double, float, java.math.BigDecimal, ...
  NUMBER -> int, double, float, java.math.BigDecimal, ...
  DATE -> Date, Time, Timestamp, String
  ...
- Konwersja z SQL do Java może wiązać się z utratą precyzji
- Producenci systemów baz danych mogą dostarczać typy Java pozwalające na uniknięcie utraty precyzji przy konwersji z SQL (np. Oracle)
- Specyficzne typy Java do obsługi typów Oracle SQL:

  CHAR -> oracle.sql.CHAR
  NUMBER -> oracle.sql.NUMBER
  DATE -> oracle.sql.DATE
  ...
- Do konwersji do typów **`oracle.sql.*`** służą odpowiednie metody **`getXXX()`** np. **`getCHAR()`**

Obsługa wartości NULL w JDBC

- Wynikiem odczytu z bazy danych wartości NULL jest:
  - **`null`** w przypadku użycia metod **`getXXX()`** zwracających obiekty (np. **`getObject()`**, **`getBigDecimal()`**)
  - **`0`** w przypadku użycia **`getByte()`**, **`getShort()`**, **`getInt()`**, **`getLong()`**, **`getFloat()`** lub **`getDouble()`**
  - **`false`** w przypadku użycia **`getBoolean()`**
- Aby sprawdzić czy wartością danego atrybutu jest NULL, należy odczytać wartość atrybutu, a następnie na rzecz zbioru wynikowego wywołać metodę **`wasNull()`**

```
ResultSet rs = st.executeQuery("SELECT ename, comm FROM emp");
while (rs.next()) {
  System.out.print(rs.getString(1)+" ");
  double d = rs.getDouble(2);
  if (!rs.wasNull()) System.out.print(d);
}
```

Strumienie w JDBC

- Wartości pól o bardzo dużych rozmiarach mogą być przetwarzane poprzez strumienie Java
- Wartości z pól typu LONG i LONG RAW są domyślnie przetwarzane w trybie strumieniowym
- Tryb strumieniowy można wyłączyć dla LONG lub LONG RAW albo włączyć dla CHAR, VARCHAR lub RAW przez tzw. redefinicję typu kolumny

Strumienie w JDBC - Przykład

- Odczyt filmu z kolumny LONG RAW tabeli *trailers* w bazie danych i zapisanie go do pliku w formacie avi

```
// trailers(title varchar(80), video LONG RAW)
ResultSet rs = stmt.executeQuery
        ("SELECT video FROM trailers WHERE title='Fan'");
if (rs.next()) {
  InputStream aviStream = rs.getBinaryStream(1);
  try {
    FileOutputStream f = new FileOutputStream("fan.avi");
    int chunk;
    while ((chunk = aviStream.read()) != -1)
      f.write(chunk);
  } catch (Exception e) { ... }
    finally { if (f != null) f.close(); }
}
```

Odczytywanie metadanych z ResultSet

- Obiekt **`ResultSet`** może posłużyć do odczytania obiektu **`ResultSetMetaData`** udostępniającego metadane o zbiorze wynikowym

```
try {
  ResultSet rset = … ;
  ResultSetMetaData md = rset.getMetaData();

  while (rset.next()) {
    for (int i = 0; i < md.getColumnCount(); i++) {
      String lbl = md.getColumnLabel();
      String typ = md.getColumnTypeName(); …
  }
} catch (SQLException e) {…}
```

Klasa PreparedStatement

- Zalecane, gdy istnieje potrzeba wielokrotnego wykonania tego samego polecenia, lecz z różnymi parametrami:
- Parametry wstawia się za pomocą znaku „?”

```
try {
  Connection conn = DriverManager.getConnection(…);

  PreparedStatement pstmt =
    conn.prepareStatement("UPDATE emp SET sal = ?");
  …
} catch (SQLException e) {…}
```

Wiązanie zmiennych i wykonywanie polecenia PreparedStatement

- Przed wywołaniem polecenia **`PreparedStatement`** należy związać parametry z konkretnymi wartościami
- Parametry wiąże się metodami **`setXXX()`** klasy **`PreparedStatement`**

```
try {
  PreparedStatement pstmt =
    conn.prepareStatement("UPDATE emp SET sal = ?");
  …
  pstmt.setInt(1, 1000);
  pstmt.executeUpdate();

  pstmt.setInt(1, 1200);
  pstmt.executeUpdate();
  …
} catch (SQLException e) {…}
```

Przypisywanie wartości NULL parametrom PreparedStatement

- W języku Java zmienne typów prostych (np. **`int`**, **`double`**) nie mogą przyjmować wartości **`null`**
- Do przypisywania wartości NULL parametrowi polecenia **`PreparedStatement`** służy metoda **`setNull()`**
- Jednym z parametrów **`setNull()`** jest kod typu danych JDBC, odpowiadającego typowi atrybutu, któremu przypisywana jest wartość NULL

```
try {
  PreparedStatement ps =
    conn.prepareStatement("UPDATE emp SET comm = ?");
  ps.setNull(1, Types.DOUBLE);
  ps.executeUpdate();
} catch (SQLException e) {…}
```

Wywoływanie procedur i funkcji składowanych (1)

- Do wywoływania procedur i funkcji składowanych służą obiekty klasy **`CallableStatement`**
  - parametry IN są ustawiane przez **`setXXX()`**, jak w przypadku **`PreparedStatement`** (podobnie INOUT)
  - wartości zwrotne funkcji i parametry OUT muszą być zarejestrowane w celu określenia ich typów (podobnie INOUT)
- Przykład wywołania procedury składowanej

```
// PROCEDURE setSal(p_ename IN VARCHAR2, p_sal IN NUMBER);

CallableStatement cs =
      conn.prepareCall( "{call setSal(?,?)}" );
cs.setString(1, "Smith");
cs.setDouble(2, 1200.50);
cs.executeUpdate();
```

Wywoływanie procedur i funkcji składowanych (2)

- Przykład wywołania funkcji składowanej

```
// FUNCTION getSal(p_ename IN VARCHAR2, p_job OUT VARCHAR2)
// RETURN NUMBER;

CallableStatement cs =
      conn.prepareCall( "{? = call getSal(?,?)}" );

cs.registerOutParameter(1, Types.NUMERIC);
cs.setString(2, "SMITH");
cs.registerOutParameter(3, Types.VARCHAR);

cs.executeUpdate();
System.out.println("Smith zarabia " + cs.getDouble(1) +
                   " jako  " + cs.getString(3));
```

Transakcje

- Przetwarzanie transakcyjne zależy od właściwości **`autoCommit`** obiektu **`Connection`**
  - domyślnie **`true`**, co oznacza oddzielną transakcję dla każdego polecenia SQL (każda instrukcja jest automatycznie zatwierdzana)
  - do zmiany trybu służy metoda **`setAutoCommit()`**
  - gdy **`autoCommit`** == **`false`**:
    - **`commit()`** - zatwierdzenie transakcji
    - **`rollback()`** - wycofanie transakcji

```
Connection conn = DriverManager.getConnection(…);
conn.setAutoCommit(false);  // zmiana trybu
...                         // polecenia SQL
conn.commit();              // zatwierdzenie
...                         // polecenia SQL
conn.rollback();            // wycofanie transakcji
```

Poziomy izolacji transakcji

- Poziomy izolacji transakcji:
  - **`TRANSACTION_NONE`** (transakcje nie są wspierane)
  - **`TRANSACTION_READ_UNCOMMITTED`**
  - **`TRANSACTION_READ_COMMITTED`**
  - **`TRANSACTION_REPEATABLE_READ`**
  - **`TRANSACTION_SERIALIZABLE`**
- Zmiana i odczyt poziomu izolacji transakcji:

```
Connection conn = DriverManager.getConnection(…);
conn.setTransactionIsolation(TRANSACTION_READ_COMMITTED);
...
int level = conn.getTransactionIsolation();
```

JDBC w różnych typach aplikacji Java

- JDBC jest uniwersalnym standardem dla wszystkich typów aplikacji Java (aplikacje samodzielne, aplety, serwlety, JavaServer Pages, JavaBeans, EJBs, procedury składowane)
- Ograniczenia dotyczące apletów
  - powinny korzystać ze sterowników typu IV ("pure Java")
  - środowisko przeglądarki ogranicza swobodę nawiązywania połączeń sieciowych do serwera,
    z którego aplet został pobrany (rozwiązanie: aplety podpisane, Oracle Connection Manager)
- Specyfika procedur składowanych Java
  - automatycznie dostępne, otwarte połączenie z bieżącą bazą danych

JDBC w serwerze Oracle 8i/9i

- Program Java uruchomiony w serwerze:
  - posiada automatycznie otwarte połączenie z bieżącym serwerem
  - nie może korzystać z trybu **`autoCommit`**
- Uzyskanie obiektu **`Connection`** reprezentującego domyślne połączenie :

Zalecane

```
Connection conn =
   new oracle.jdbc.OracleDriver().defaultConnection();
```

```
Connection conn =
   DriverManager.getConnection("jdbc:default:connection:");
```

```
Connection conn =
   DriverManager.getConnection("jdbc:oracle:kprb:");
```

JDBC w serwlecie – Przykład (1)

EmpServlet.java

```
import java.sql.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class EmpServlet extends HttpServlet {
  Connection conn;
  public void init(ServletConfig config)
                   throws ServletException {
    super.init(config);
    try {
      Class.forName("oracle.jdbc.OracleDriver");
      conn = DriverManager.getConnection(
       "jdbc:oracle:thin:@host:1521:SID", "scott", "tiger");
    } catch (Exception e) {}
  }
```

JDBC w serwlecie – Przykład (2)

EmpServlet.java

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<HTML><BODY>");
  if (conn != null) {
    try {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery("SELECT ename, sal FROM emp");
      while (rset.next())
          out.println(rset.getString(1)+": "+rset.getDouble(2)+"<BR>");
      rset.close(); stmt.close();
      } catch (SQLException e) {}
    }
     out.println("</BODY></HTML>");
  }
```

JDBC w serwlecie – Przykład (3)

EmpServlet.java

```
  public void destroy()
  {
    if (conn != null)
    {
      try
      {
        conn.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

JDBC w JSP – Przykład

empJSP.jsp

```
<%@ page language="java" import="java.sql.*" %>
<HTML><HEAD><TITLE>Employees</TITLE></HEAD><BODY BGCOLOR="#FFFFFF">
<% try
   {
     DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
     Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@host:1521:SID", "scott", "tiger");
     Statement stmt = conn.createStatement();
     ResultSet rset = stmt.executeQuery("SELECT ename, sal FROM emp");
     while (rset.next())
       out.println(rset.getString(1)+": "+rset.getDouble(2)+"<BR>");
     rset.close(); stmt.close(); conn.close();
    } catch (Exception e) { out.println(e); }
%>
</BODY></HTML>
```

Nowe możliwości JDBC 2.0 API

- JDBC 2.0 core API – pakiet **`java.sql`**
  - przewijalne zbiory wynikowe (ang. scrollable result sets)
  - programowe modyfikacje bazy danych
  - modyfikacje bazy danych w trybie wsadowym (ang. batch mode)
  - wsparcie dla typów danych SQL 99 (SQL3)
- JDBC 2.0 Optional Package (Standard Extension package) – pakiet **`javax.sql`**
  - wsparcie dla rozproszonych transakcji
  - obsługa pul połączeń (ang. connection pooling)
  - łączenie się z bazami danych przez nazwy logiczne (JNDI)
  - zbiory rekordów (ang. rowsets)

Rozszerzenia funkcjonalne zbiorów wynikowych

- JDBC 2.0 ułatwia tworzenie graficznych interfejsów do przetwarzania wyników zapytań oferując:
  - przewijalne (ang. scrollable) zbiory wynikowe
  - możliwość programowej modyfikacji danych poprzez zbiory wynikowe
- Aby uzyskać zbiór wynikowy o rozszerzonej funkcjonalności należy podać dwa dodatkowe parametry tworząc obiekt **`Statement`** lub **`PreparedStatement`**:
  - typ (ang. type):
    - TYPE_FORWARD_ONLY (domyślnie)
    - TYPE_SCROLL_INSENSITIVE
    - TYPE_SCROLL_SENSITIVE
  - współbieżność (ang. concurrency):
    - CONCUR_READ_ONLY (domyślnie)
    - CONCUR_UPDATABLE

Tworzenie zbiorów wynikowych JDBC 2.0 - przykłady

```
Statement stmt = conn.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery (
               "SELECT ename, sal FROM emp");
```

```
PreparedStatement pstmt = conn.prepareStatement(
                 "SELECT ename FROM emp WHERE sal > ?"),
                 ResultSet.TYPE_SCROLL_SENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
pstmt.setDouble(1, 1000.0);
ResultSet rs = pstmt.executeQuery ();
```

Weryfikacja właściwości zbiorów wynikowych

- Dany sterownik JDBC może nie wspierać wszystkich właściwości zbiorów wynikowych przewidzianych przez JDBC 2.0
- Istnieje możliwość odczytu właściwości utworzonego zbioru wynikowego:

```
int type = rs.getType();
// 1003 -> ResultSet.TYPE_FORWARD_ONLY
// 1004 -> ResultSet.TYPE_SCROLL_INSENSITIVE
// 1005 -> ResultSet.TYPE_SCROLL_SENSITIVE
```

```
int concurrency =  rs.getConcurrency();
// 1007 -> ResultSet.CONCUR_READ_ONLY
// 1008 -> ResultSet.CONCUR_UPDATABLE
```

Korzystanie z przewijalnych zbiorów wynikowych

- Nawigacja po rekordach zbioru wynikowego

```
rs.next(); rs.previous(); // używane w warunkach pętli
rs.first(); rs.last(); rs.beforeFirst(); rs.afterLast();
rs.absolute(n);  // n-ty rekord od początku
rs.absolute(-n); // n-ty rekord od końca
rs.relative(n);  // n-ty rekord po bieżącym
rs.relative(-n); // n-ty rekord przed bieżącym
```

- Odczyt pozycji kursora

```
int currRow = rs.getRow(); // numer rekordu w zbiorze
if (rs.isFirst()) { ... };
if (rs.isLast()) { ... };
if (rs.isBeforeFirst()) { ... };
if (rs.isAfterLast()) { ... };
```

Modyfikowanie danych poprzez zbiory wynikowe

- JDBC 2.0 pozwala na modyfikowanie danych w zbiorze wynikowym metodami Java zamiast poleceń SQL (UPDATE, INSERT, DELETE)
- Dla zapytań na jednej tabeli pobierających klucz główny
- Przy UPDATE i DELETE należy najpierw przejść do żądanego rekordu

```
Statement st=conn.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs=st.executeQuery("SELECT empno,ename,sal FROM emp");
```

UPDATE

```
rs.updateDouble("SAL", 1200.0); // updateDouble(3, 1200.0)
rs.updateRow(); // KONIECZNE !!!
```

INSERT

```
rs.moveToInsertRow();
rs.updateInt("EMPNO", 8734);
rs.updateString("ENAME", "KLOSS");
rs.updateDouble("SAL", 1200.0);
rs.insertRow(); // KONIECZNE !!!
rs.moveToCurrentRow(); // Opcjonalnie
```

DELETE

```
rs.deleteRow();
```

Widoczność zmian

- Transakcja widzi zmiany dokonane przez siebie
- Widoczność zmian dokonanych przez inne transakcje zależy od poziomu izolacji
- Wpływ widocznych zmian dokonanych przez inne transakcje na otwarte zbiory wynikowe zależy od typu zbioru wynikowego (i implementacji!!!):
  - **`TYPE_SCROLL_INSENSITIVE`** – zmiany nie są widoczne
  - **`TYPE_SCROLL_SENSITIVE`** – widoczne wyniki UPDATE, wyniki INSERT i DELETE widoczne lub nie – zależnie od implementacji
  - **`TYPE_FORWARD_ONLY`** – zachowanie zależy od implementacji i zapytania
- Nawet zmiany dokonane poprzez zbiór wynikowy mogą być w nim niewidoczne (!) w pewnych implementacjach
- Wymuszenie odświeżenia bieżącego rekordu w zbiorze wynikowym typu **`TYPE_SCROLL_SENSITIVE`**:

```
rs.refreshRow();
```

Weryfikacja właściwości funkcjonalnych na danej platformie

- Dostępność wielu właściwości funkcjonalnych oraz sposób funkcjonowania pewnych mechanizmów zależy od platformy (DBMS, sterownik JDBC)
- Właściwości platformy można zweryfikować poprzez metadane dotyczące bieżącego połączenia

```
boolean b;
DatabaseMetaData dmd = conn.getMetaData();
// czy dany typ zbioru wynikowego jest dostępny?
b=dmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE);
// czy zbiór wynikowy widzi zmiany przez siebie dokonane?
b=dmd.ownUpdatesAreVisible(ResultSet.TYPE_SCROLL_INSENSITIVE);
// czy sterownik JDBC wspiera tryb wsadowy?
b=dmd.supportsBatchUpdates();
...
```

Modyfikowanie bazy danych w trybie wsadowym (1)

- W JDBC 2.0 obiekty **`Statement`**, **`PreparedStatement`** i **`CallableStatement`** zachowując dodatkową funkcjonalność posiadają związaną ze sobą listę poleceń
- Na liście mogą znaleźć się instrukcje zwracające liczbę modyfikacji (INSERT, UPDATE, DELETE, instrukcje DDL)
- Lista nie może zawierać poleceń zwracających **`ResultSet`** (SELECT)
- Instrukcje z listy mogą być przesłane do bazy danych w trybie wsadowym (ang. batch), co może poprawić efektywność
- Przetwarzanie wsadowe jest dostępne przy wyłączonym trybie **`autoCommit`** (!)

Modyfikowanie bazy danych w trybie wsadowym (2)

- Przykład przetwarzania wsadowego z użyciem **`Statement`**:

```
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO emp(empno, ename, sal) " +
              "VALUES(9954, 'SMYTHE', 1234.5)");
stmt.addBatch("INSERT INTO emp(empno, ename, sal) " +
              "VALUES(9955, 'GRAY', 2234.5)");
int [] updateCounts = stmt.executeBatch();
conn.commit();
conn.setAutoCommit(true);
```

Modyfikowanie bazy danych w trybie wsadowym (3)

- Przykład z użyciem **`PreparedStatement`**:

```
conn.setAutoCommit(false);
PreparedStatement ps = conn.prepareStatement(
              "INSERT INTO emp(empno, ename, sal) " +
              "VALUES(?, ?, ?)");
ps.setInt(1, 9954); ps.setString(2, "SMYTHE");
ps.setDouble(3, 1234.5);
ps.addBatch();
ps.setInt(1, 9955); ps.setString(2, "GRAY");
ps.setDouble(3, 2234.5);
ps.addBatch();
int [] updateCounts = ps.executeBatch();
conn.commit();
conn.setAutoCommit(true);
```

Wsparcie dla typów danych standardu SQL 99 (SQL3)

- Interfejsy JDBC 2.0 odpowiadające typom danych SQL3:
  - **`Blob`** -> odpowiada wartości BLOB w SQL
  - **`Clob`** -> odpowiada wartości CLOB w SQL
  - **`Array`** -> odpowiada wartości ARRAY w SQL
  - **`Struct`** -> odpowiada wartości typu strukturalnego w SQL
  - **`Ref`** -> odpowiada wartości REF w SQL
- Metody **`getXXX()`**, **`setXXX()`** i **`updateXXX()`** dla typów SQL3:

Typ SQL3	getXXX	setXXX	updateXXX
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
strukturalny	getObject	setObject	updateObject
REF	getRef	setRef	updateRef

Obiekty BLOB w JDBC

- Operacje na dużych obiektach binarnych (BLOB) w JDBC realizowane są poprzez interfejs **`java.sql.Blob`** (możliwy jest również bezpośredni dostęp strumieniowy)
- Metody interfejsu **`Blob`**:
  - **`getBytes(long pos, int length)`**
  - **`setBytes(long pos, byte[] bytes)`**
  - **`length()`**
  - **`getBinaryStream()`**
  - **`setBinaryStream(long pos)`**
  - **`...`**

Obiekty BLOB w JDBC – Przykład (1)

- Odczyt obrazu jpg z kolumny BLOB tabeli *covers* w bazie danych poprzez interfejs **`Blob`** i zapisanie go do pliku

```
// covers(movie_id number, image BLOB)
ResultSet rs = stmt.executeQuery
        ("SELECT image FROM covers WHERE movie_id = 1");
if (rs.next()) {
  Blob b = rs.getBlob(1);
  byte[] jpgData = b.getBytes(1, (int) b.length());
  try {
    FileOutputStream f = new FileOutputStream("cover.jpg");
    f.write(jpgData);
  } catch (Exception e) { ... }
    finally { if (f != null) f.close(); }
}
```

Obiekty BLOB w JDBC – Przykład (2)

- Odczyt obrazu jpg z kolumny BLOB tabeli *covers* w bazie danych poprzez strumień i zapisanie go do pliku

```
// covers(movie_id number, image BLOB)
ResultSet rs = stmt.executeQuery
        ("SELECT image FROM covers WHERE movie_id = 1");
if (rs.next()) {
  InputStream jpgStream = rs.getBinaryStream(1);
  try {
    FileOutputStream f = new FileOutputStream("cover.jpg");
    int chunk;
    while ((chunk = jpgStream.read()) != -1)
      f.write(chunk);
  } catch (Exception e) { ... }
    finally { if (f != null) f.close(); }
}
```

Strukturalne typy danych SQL 99

```
CREATE TYPE address AS OBJECT
(street CHAR(20),
 house INTEGER
);
/
```

```
CREATE TABLE people
(name CHAR(20),
 home address)
;
```

- Domyślnie wartości strukturalnych typów danych SQL 99 są mapowane w obiekty **`java.sql.Struct`**
- Istnieje możliwość tworzenia specjalizowanych klas odpowiadających typom strukturalnym SQL 99 (ang. custom Java classes)
  - mechanizm ten ułatwia operacje na wartościach strukturalnych typów danych
  - mechanizm ten wymaga dodania stosownego wpisu w tzw. mapie typów (ang. type map)
  - custom Java classes można tworzyć "ręcznie" lub korzystając z narzędzi np. Oracle JPublisher

Obsługa strukturalnych typów danych poprzez interfejs Struct

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery(
                 "select name, home from people");

while (rset.next())
{
  System.out.print(rset.getString(1) + " ");
  Struct h = (Struct) rset.getObject(2);
  Object [] atts = h.getAttributes();
  System.out.println("Home: " + atts[0] + " " + atts[1]);
}
rset.close();
stmt.close();
```

Klasy Java odpowiadające typom strukturalnym SQL 99

```
public class Address implements SQLData {
  public String street;
  public int house;
  private String sql_type;
  public String getSQLTypeName() { return sql_type; }

  public void readSQL(SQLInput stream, String type)
              throws SQLException
  {
    sql_type = type;
    street = stream.readString();  house = stream.readInt();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeString(street);  stream.writeInt(house);
  }
}
```

Obsługa strukturalnych typów danych poprzez mapowanie (1)

```
java.util.Map map = conn.getTypeMap();
map.put("ADDRESS", Class.forName("Address"));

Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery (
                 "select name, home from people");
while (rset.next())
{
  System.out.print(rset.getString(1) + " ");
  Address h = (Address) rset.getObject(2);
  System.out.println(h.street + " " + h.house);
}
rset.close();
stmt.close();
```

Obsługa strukturalnych typów danych poprzez mapowanie (2)

```
java.util.Map map = conn.getTypeMap();
map.put("ADDRESS", Class.forName("Address"));

Address h;
stmt = conn.createStatement ();
rset = stmt.executeQuery (
       "select home from people where name = 'CLARK'");
if (rset.next()) {
  h = (Address) rset.getObject(1);
}
rset.close();
stmt.close();
...
PreparedStatement ps = conn.prepareStatement(
          "update people set home = ? where name = 'CLARK'");
h.street = "Oak Street"; h.house = 99;
ps.setObject(1, h);
ps.executeUpdate();
ps.close();
```

JDBC 2.0 Optional Package

- Integralna część technologii Enterprise JavaBeans (EJBs)
- Wsparcie dla Java Naming and Directory Interface (JNDI)
  - możliwość posługiwania się logicznymi nazwami baz danych
  - uniknięcie konieczności zaszywania w aplikacji fizycznych parametrów połączenia i typu sterownika JDBC
- Pule połączeń
  - wykorzystane połączenia nie są zamykane lecz zwracane do puli
  - redukcja kosztu operacji połączenia z bazą danych (zamiast tworzenia nowego połączenia – pobranie z puli)
- Rozproszone transakcje
- Zbiory rekordów (obiekty **`RowSet`**)
  - zbiory wynikowe (obiekty **`ResultSet`**), które mogą funkcjonować jako komponenty JavaBean
  - mogą pozostawać odłączone od swojego źródła danych z możliwością propagacji zmian

Źródła danych (ang. DataSources) w JDBC 2.0 Optional Package

- Obiekt **`DataSource`** reprezentuje źródło danych (np. DBMS)
- W JDBC 2.0 **`DataSource`** w zakresie uzyskiwania połączenia z bazą danych jest alternatywą dla **`DriverManager`**
- Źródło **`DataSource`** może wspierać:
  - transakcje rozproszone
  - obsługę pul połączeń
- **`DataSource`** może zostać zarejestrowane w JNDI
- Aplikacja łącząc się z bazą danych:
  - wyszukuje źródło danych w JNDI przez jego logiczną nazwę
  - poprzez obiekt **`DataSource`** uzyskuje obiekt **`Connection`**
- Połączenie uzyskane poprzez **`DataSource`** może brać udział w transakcjach rozproszonych i/lub funkcjonować w ramach puli połączeń zależnie od właściwości źródła

Korzystanie ze źródeł danych – Przykład (Oracle)

- Utworzenie źródła danych i rejestracja w JNDI

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci8");
ods.setServerName("host1");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("orcl");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");
...
Context ctx = new InitialContext();
ctx.bind("jdbc/FinanceDB", ods);
```

- Otwarcie połączenia z bazą danych

```
Context ctx = new InitialContext();
OracleDataSource ds
   = (OracleDataSource)ctx.lookup("jdbc/FinanceDB");
Connection conn = ds.getConnection();
```

Nowe własności JDBC 3.0

- Core API i Optional Package w J2SE (1.4)
- Rozszerzenia funkcjonalne obejmują m.in.:
  - udoskonalenie obsługi pul połączeń
  - instrukcja SAVEPOINT w transakcjach
  - metadane dotyczące parametrów prekompilowanych poleceń (ang. prepared statements)
  - zbiory wynikowe otwarte po zatwierdzeniu transakcji
  - obsługa dowolnych tabelarycznych źródeł danych

SQLJ Część 0: Zagnieżdżony SQL

- Prosty, zwarty język pozwalający na zagnieżdżanie instrukcji SQL w kodzie Java
- Standard obsługujący statyczny SQL (składnia instrukcji znana na etapie tworzenia kodu)
- Standard zapewniający binarną przenaszalność aplikacji miedzy różnymi systemami baz danych
- Standard dopuszczający możliwość rozszerzeń ze strony producentów systemów baz danych

Zalety SQLJ w porównaniu z JDBC

- Możliwość sprawdzenia poprawności instrukcji SQL na etapie kompilacji
- Silna kontrola zgodności typów
  - przy przetwarzaniu wyników zapytań
  - na poziomie połączeń z bazą danych
- Prostszy i bardziej zwarty kod źródłowy
- Możliwość dostosowywania wygenerowanych aplikacji do konkretnej platformy

Podstawy składni SQLJ

- Klauzule SQLJ (deklaracje i instrukcje) rozpoczynają się od **`#sql`**, a kończą średnikiem
- Treść poleceń SQL zawarta jest w nawiasach klamrowych; przed zamykającym nawiasem nie stawia się średnika (!)
- Treść SQL może być rozłożona na kilka wierszy

```
#sql {
  UPDATE emp SET sal = sal * 1.1
  WHERE sal < 1000
};
```

Wyrażenia zewnętrzne Java (1)

- Polecenia SQL mogą odwoływać się do zmiennych i metod Java poprzez tzw. wyrażenia zewnętrzne (ang. host expressions)
- Wyrażenia zewnętrzne poprzedzane są dwukropkiem
- W najprostszej postaci wyrażenia zewnętrzne mają postać zmiennych zewnętrznych (ang. host variables), czyli odwołań do zmiennych z otaczającego kodu Java

```
double maxSal = 1000.0;
#sql {
  UPDATE emp SET sal = sal * 1.1
  WHERE sal < :maxSal
};
```

Wyrażenia zewnętrzne Java (2)

- Dla każdego wyrażenia zewnętrznego po dwukropku można podać tryb jego przekazania (IN, OUT, INOUT)
- Na ogół trybu nie trzeba jawnie specyfikować, gdyż domyślnie przyjmowany jest właściwy
- Przykłady wyrażeń zewnętrznych:
  `:var`, **`:(var1+var2)`**, **`:OUT(var1+var2)`**, **`:(count++)`**

```
double maxSal = 1000.0;
#sql {
  UPDATE emp SET sal = sal * 1.1
  WHERE sal < :IN maxSal
};
```

Obsługa wyjątków w SQLJ

- Instrukcje SQLJ mogą generować wyjątek **`java.sql.SQLException`**
- Metody zawierające instrukcje SQLJ muszą obsługiwać ten wyjątek lub deklarować możliwość jego generowania

```
import java.sql.*;
...
try {
  #sql { UPDATE emp SET sal = sal * 1.1 };
}
catch (SQLException e) {
  System.err.println("Error: " + e);
  String sqlState = e.getSQLState();
  String message = e.getMessage();
  int errorCode = e.getErrorCode();
  ...
}
```

Przetwarzanie zapytań w SQLJ

- Wyniki zapytań przypisywane są do iteratorów i następnie przetwarzane za ich pośrednictwem
- Iteratory odpowiadają zbiorom wynikowym JDBC, ale oferują kontrolę zgodności typów
- Iterator jest wystąpieniem zadeklarowanej wcześniej klasy iteratora
- Rodzaje iteratorów:
  - iteratory nazwane
  - iteratory pozycyjne
  - iteratory ze słabą kontrolą typów
- Dla zapytań zwracających dokładnie jeden rekord istnieje instrukcja SELECT ... INTO ...

Iteratory nazwane

- Deklaracja klasy iteratora nazwanego zawiera nazwy i typy jego pól np.:
  `#sql public iterator EmpNameIter(String ename, double sal);`
- Przed słowem **`iterator`** mogą pojawiać się standardowe modyfikatory klasy (**`public`**, **`static`**, ...)
- Klasa iteratora nazwanego zawiera metody:
  - **`next()`** – przejście do kolejnego rekordu
  - **`close()`** – zamknięcie iteratora
  - metody dostępu do pól o nazwach odpowiadających nazwom pól iteratora (w powyższym przykładzie: **`ename()`** i **`sal()`**)

Iterator nazwany - Przykład

```
#sql iterator EmpNameIter(String ename, double sal);
...
EmpNameIter iter;
double minSal = 1000.00;
#sql iter = {SELECT ename, sal FROM emp
             WHERE sal > :minSal };
while (iter.next()) {
  String n = iter.ename();
  double s = iter.sal();
  System.out.println(n + "   " + s);
}
iter.close();
```

Zasady korzystania z iteratorów nazwanych

- Zawsze należy zamykać wykorzystany iterator
- W przypadku gdy nazwa kolumny w zapytaniu nie jest prawidłowym identyfikatorem Java, należy skorzystać w zapytaniu z aliasu dla kolumny
- Kolejność wyrażeń na liście SELECT zapytania nie jest istotna
- Zapytanie może zwracać więcej kolumn niż przewiduje to iterator
- Wielkość liter w nazwach pól iteratora nie ma znaczenia przy sprawdzaniu ich zgodności z nazwami kolumn zapytania

Iteratory pozycyjne

- Deklaracja klasy iteratora pozycyjnego zawiera typy jego pól np.:
  `#sql public iterator EmpPosIter(String, double);`
- Przed słowem **`iterator`** mogą pojawiać się standardowe modyfikatory klasy (**`public`**, **`static`**, …)
- Klasa iteratora pozycyjnego nie zawiera metod dostępu do pól iteratora
- Przejście do kolejnych rekordów z pobraniem wartości do zmiennych zewnętrznych jest realizowane za pomocą instrukcji FETCH ... INTO ...
- W nawigacji po rekordach iteratora pozycyjnego wykorzystywana jest metoda **`endFetch()`**, zwracająca prawdę gdy iterator nie znajduje się na rekordzie

Iterator pozycyjny - Przykład

```
#sql iterator EmpPosIter(String, double);
...
EmpPosIter iter;
double minSal = 1000.00;
String empName;
double empSal;
#sql iter = {SELECT ename, sal FROM emp
             WHERE sal > :minSal };
while (true) {
  #sql { FETCH :iter INTO :empName, :empSal };
  if (iter.endFetch()) break;
  System.out.println(empName + ": " + empSal);
}
iter.close();
```

Zasady korzystania z iteratorów pozycyjnych

- Zawsze należy zamykać wykorzystany iterator
- Kolejność wyrażeń na liście SELECT zapytania jest istotna
- Zapytanie musi zwracać tyle kolumn, ile pól zawiera iterator
- Ze względu na sposób działania instrukcji FETCH i metody **`endFetch()`** zalecane jest przetwarzanie danych z iteratora zgodnie z pokazanym wcześniej przykładem

Zapytania zwracające jeden rekord

- Instrukcja SELECT ... INTO ... pozwala uniknąć konieczności stosowania iteratorów, gdy zapytanie z natury zwraca jeden rekord

```
double maxSal;
#sql {
       SELECT max(sal) INTO :maxSal
       FROM emp
};
```

- Istnienie tej konstrukcji jest jedną z zalet SQLJ w porównaniu z JDBC
- W standardzie JDBC wszystkie zapytania są przetwarzane poprzez zbiory wynikowe

Wywoływanie procedur i funkcji składowanych

- Przykład wywołania procedury składowanej:
  `#sql { CALL PROC_1(:IN a, :OUT b, :INOUT c) };`
- Przykład wywołania funkcji składowanej:
  `#sql res = { VALUES(FUN_1(:IN a, :OUT b, :INOUT c)) };`
- UWAGA: Dla wszystkich parametrów konieczne jest podanie trybu ich przekazania, zgodnego z nagłówkiem wywoływanego podprogramu

Instrukcja przypisania (SET)

- Służy do przypisania wartości zmiennej zewnętrznej Java w ramach instrukcji SQL
- Użyteczna w przypadku gdy przypisywana wartość jest przetworzonym wynikiem funkcji składowanych (w innych sytuacjach wystarczy normalne wywołanie funkcji lub instrukcja przypisania w kodzie Java)

```
// FUN_1 i FUN_2 – funkcje składowane
// x – zmienna w kodzie Java typu zgodnego z typami
// zwrotnymi funkcji FUN_1 i FUN_2

#sql { SET :x = FUN_1() + FUN_2() };
```

Obsługa wartości NULL w SQLJ

- W SQLJ odpowiednikiem wartości NULL języka SQL jest wartość **`null`** języka Java
- Typy proste Java (**`int`**, **`double`**, ...) nie mogą przyjmować wartości **`null`**
- Do obsługi wartości NULL, należy wykorzystać klasy otaczające (**`Integer`**, **`Double`**, ...)

```
Double commission;
int empNo = 3456;
#sql { SELECT comm INTO :commission
       FROM emp WHERE empno = :empNo };
if (commission != null)
  System.out.println("comm: " + commission.doubleValue());
```

Otwieranie połączeń z bazą danych

- Gdy aplikacja korzysta z jednego połączenia należy wykonać następujące kroki:
  - Załadowanie sterownika JDBC
  - Utworzenie nowego kontekstu połączenia i uczynienie go domyślnym
- Tworząc kontekst połączenia należy podać (w zależności od wybranego konstruktora):
  - URL, użytkownika, hasło, ustawienie **`autoCommit`**
  - istniejący obiekt **`Connection`**
  - ...

```
import sqlj.runtime.ref.DefaultContext;
...
Class.forName("oracle.jdbc.OracleDriver");
DefaultContext.setDefaultContext(
  new DefaultContext("jdbc:oracle:thin:@localhost:1521:orcl",
                     "scott", "tiger", false));
```

Korzystanie z wielu połączeń z bazami danych

- Gdy aplikacja korzysta z wielu połączeń należy:
  - Załadować sterowniki JDBC
  - Zadeklarować klasy kontekstów połączeń (gdy różne schematy)
  - Utworzyć nowe nazwane konteksty połączenia jako instancje zadeklarowanych klas
  - Opcjonalnie uczynić jeden z kontekstów domyślnym
  - Dla każdej instrukcji SQLJ, która ma wykonać się w innym kontekście niż domyślny, podać jawnie odpowiedni kontekst:

```
#sql [connContext] { ... };
#sql [connContext] var = { ... };
```

Korzystanie z wielu połączeń z bazami danych - Przykład

```
import sqlj.runtime.ref.DefaultContext;
...
#sql context Ctx1;
#sql context Ctx2;
...
Class.forName("oracle.jdbc.OracleDriver");
...
Ctx1 myCtx1 = new Ctx1("jdbc:oracle:thin:@host1:1521:orcl",
                       "scott", "tiger", false));
Ctx2 myCtx2 = new Ctx2("jdbc:oracle:thin:@host2:1526:orcl",
                       "brown", "lion", false));
...
#sql [myCtx1] { UPDATE emp SET sal = sal + 100 };
#sql [myCtx2] { DELETE FROM emp_NY WHERE salary > 2000 };
DefaultContext.setDefaultContext(myCtx1);
#sql { UPDATE emp SET sal = sal + 100 };
```

Zamykanie kontekstów połączeń z bazą danych

- Zalecane jest zamykanie kontekstów połączeń z bazą danych gdy nie są już potrzebne
- Każda z klas kontekstów połączenia (również **`DefaultContext`**) posiada metodę **`close()`**
- Przed zamknięciem kontekstu połączenia należy zakończyć transakcję
  - standard nie specyfikuje jak ma zakończyć się transakcja, gdy zamykany jest kontekst połączenia
  - sposób zakończenia transakcji może być różny w zależności od tego czy kontekst połączenia jest zamykany jawnie, czy też w wyniku działania mechanizmu Garbage Collection

Obsługa połączeń z bazą danych (Oracle)

- Oracle SQLJ dostarcza klasę **`Oracle`** upraszczającą korzystanie z instancji klasy **`DefaultContext`**:
- Metody klasy **`Oracle`**:
  - **`connect()`** – rejestruje sterownik JDBC Oracle; tworzy instancję kontekstu; ustawia i zwraca domyślny kontekst (gdy już wcześniej był ustawiony – zwraca **`null`**)
  - **`getConnection()`** – rejestruje sterownik JDBC Oracle; tworzy i zwraca instancję kontekstu
  - **`close()`** – zamyka domyślny kontekst
- Uwagi dotyczące parametrów metod **`connect()`** i **`getConnection()`**:
  - podstawowe parametry to JDBC URL, użytkownik i hasło
  - opcjonalnie można podać jeszcze tryb **`autoCommit`** (domyślnie **`false`**)
  - podobnie jak w przypadku konstruktorów **`DefaultContext`**, akceptowane są również inne zestawy parametrów

Obsługa połączeń z bazą danych (Oracle) - Przykład

```
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
...
Oracle.connect("jdbc:oracle:thin:@host1:1521:orcl",
                "scott", "tiger");
DefaultContext myCtx2 = Oracle.getConnection (
"jdbc:oracle:thin:@host2:1526:orcl", "brown", "lion");
...
Oracle.close();
myCtx2.close();
```

Sterowanie przebiegiem transakcji

- Gdy dla danego kontekstu połączenia włączony jest tryb **`autoCommit`**:
  - każda instrukcja stanowi oddzielną transakcję
  - zmiany są automatycznie zatwierdzane
- Gdy tryb **`autoCommit`** jest wyłączony należy korzystać z instrukcji:

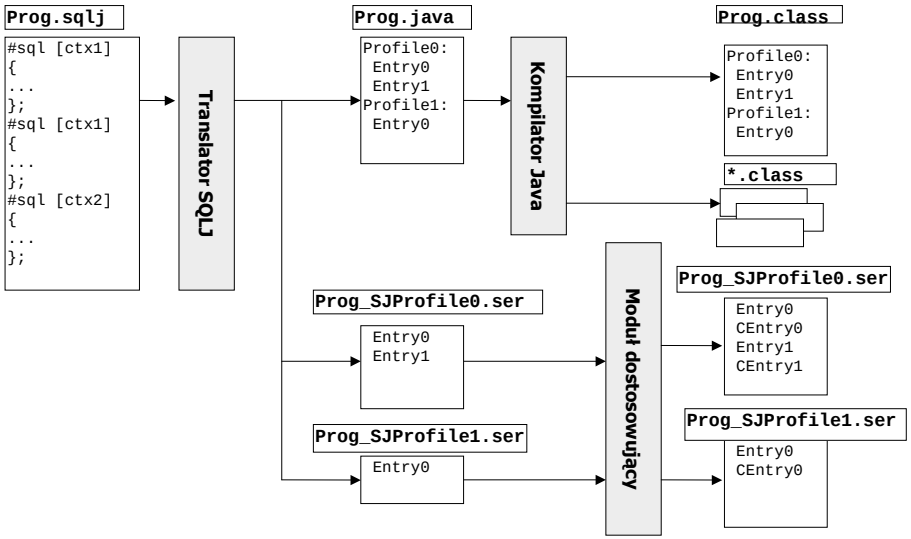
```
#sql { COMMIT };
#sql { ROLLBACK };
#sql { SET TRANSACTION READ ... };
#sql { SET TRANSACTION ISOLATION LEVEL ...};
```

Kontekst wykonania

- Dla każdej instrukcji SQLJ można podać kontekst wykonania (niezależny od kontekstu połączenia):
  `#sql [execContext] { ... };`
  `#sql [connContext, execContext] { ... };`
- Kontekst wykonania umożliwia:
  - śledzenie liczby dokonanych zmian dla UPDATE
  - kontrolę przebiegu zapytań (np. timeout)

```
sqlj.runtime.ExecutionContext ec =
                    new sqlj.runtime.ExecutionContext();
#sql [ec] { UPDATE emp SET sal = sal * 2 WHERE sal < 1000 };
System.out.println(ec.getUpdateCount() + "rows updated.");
```

Translacja programów SQLJ



Wykrywanie błędów na etapie translacji

- Translator SQLJ przeprowadza kontrolę składniową oraz kontrolę semantyczną w trybie on-line lub off-line (w zależności od parametrów wywołania translatora)
- Kontrola składniowa obejmuje:
  - sprawdzenie składni poleceń SQLJ (np. {} ;)
  - sprawdzenie składni wyrażeń zewnętrznych Java
- Kontrola semantyczna off-line obejmuje:
  - sprawdzenie poprawności typów wyrażeń Java w instrukcjach SQL
  - określenie natury instrukcji SQL (SELECT, INSERT, itd.)
- Kontrola semantyczna on-line obejmuje to co off-line oraz:
  - sprawdzenie składni instrukcji SQL
  - sprawdzenie zgodności typów SQL i Java
  - sprawdzenie poprawności odwołań do obiektów bazy danych

Wywoływanie translatora SQLJ

- Ogólna składnia: **sqlj *opcje lista_plików***
- W przypadku translacji programu lista plików może zawierać pliki **`.sqlj`** i **`.java`**
- W przypadku dostosowywania programu lista plików może zawierać pliki **`.ser`** i **`.jar`**
- Opcja –user włącza kontrolę semantyczną on-line i podaje parametry połączenia dla klas kontekstów połączeń
- Inne użyteczne opcje: -driver, -profile, -ser2class, -props, ...

```
sqlj App.sqlj Support.java
```

```
sqlj –user=scott/tiger@jdbc:oracle:oci:@lab Prog.sqlj
```

```
sqlj -user@myCtx=scott/tiger -user=brown/lion App.sqlj
```

Środowisko uruchomieniowe SQLJ

- Translator SQLJ generując kod Java zastępuje zagnieżdżone instrukcje SQL odwołaniami do środowiska uruchomieniowego SQLJ (ang. SQLJ runtime)
- SQLJ runtime jest napisany w czystej Javie, ma postać zbioru pakietów Java umieszczonych w archiwum np. **`runtime.zip`**
- SQLJ runtime wykonuje instrukcje SQL programu korzystając z informacji zawartych w plikach profili
- Typowe implementacje SQLJ runtime realizują dostęp do bazy danych poprzez sterownik JDBC (np. Oracle, IBM)

Równoważność z JDBC 2.0

- Obsługa złożonych typów danych:
  - typy danych JDBC 2.0 (BLOB, CLOB, ...)
  - typy danych SQL 99 (SQL3) wspierane poprzez odpowiadające im klasy Java – custom Java classes (np. stworzone przez JPublisher)
- Przewijalne iteratory (opcjonalnie wrażliwe na zmiany)
  - nazwane
  - pozycyjne

Obiekty BLOB w SQLJ – Przykład (1)

- Odczyt obrazu jpg z kolumny BLOB tabeli *covers* w bazie danych poprzez interfejs **`Blob`** i zapisanie go do pliku

```
// covers(movie_id number, image BLOB)
Blob b;
#sql {SELECT image into :b
      FROM covers where movie_id = 1};
FileOutputStream f = null;
byte[] jpgData = b.getBytes(1, (int) b.length());
try {
  f = new FileOutputStream("cover.jpg");
  f.write(jpgData);
}
catch (IOException e) { System.out.println(e); }
finally { if (f != null) f.close(); }
```

Obiekty BLOB w SQLJ – Przykład (2)

- Odczyt obrazu jpg z kolumny BLOB tabeli *covers* w bazie danych strumieniowo poprzez interfejs **`Blob`**

```
// covers(movie_id number, image BLOB)
Blob b;
#sql {SELECT image into :b
      FROM covers where movie_id = 1};
FileOutputStream f = null;
InputStream is = b.getBinaryStream();
try {
  f = new FileOutputStream("cover.jpg");
  int chunk;
  while ((chunk = is.read()) != -1) f.write(chunk);
}
catch (IOException e) { System.out.println(e); }
finally { if (f != null) f.close(); }
```

Przewijalny iterator nazwany - Przykład

```
#sql public static ScrNameIter implements
     sqlj.runtime.Scrollable with (sensitivity=SENSITIVE)
     (String ename, double sal);
ScrNameIter ni;
#sql ni  = { SELECT ename, sal FROM emp };
...
ni.previous();
...
ni.first();
...
ni.absolute(n);
...
ni.relative(n);
...
```

Przewijalny iterator pozycyjny - Przykład

```
#sql public static ScrPosIter implements
     sqlj.runtime.Scrollable (String, double);
ScrPosIter iter;
#sql iter  = { SELECT ename, sal FROM emp };
...
#sql { FETCH PRIOR FROM :iter INTO :x, :y };
...
#sql { FETCH FIRST FROM :iter INTO :x, :y };
...
#sql { FETCH ABSOLUTE :n FROM :iter INTO :x, :y };
...
#sql { FETCH RELATIVE :n FROM :iter INTO :x, :y };
...
```

Współistnienie SQLJ i JDBC

- Instrukcje SQLJ i wywołania JDBC mogą współistnieć w kodzie Java
  - SQLJ dla statycznych instrukcji SQL
  - JDBC dla dynamicznych instrukcji SQL
- Współpraca SQLJ i JDBC bazuje na konwersjach między:
  - połączeniami JDBC a kontekstami połączeń SQLJ
  - zbiorami wynikowymi JDBC a iteratorami SQLJ

JDBC -> SQLJ

```
MyContext myCtx = new MyContext(conn);
ResultSet rs = ... ;
#sql iter = { CAST :rs };
```

SQLJ -> JDBC

```
Connection conn = myCtx.getConnection();
#sql iter = { SELECT ... };
ResultSet rs = iter.getResultSet();
```

SQLJ w różnych typach aplikacji Java

- SQLJ jest uniwersalnym standardem dla wszystkich typów aplikacji Java (podobnie jak JDBC):
  - aplikacje samodzielne
  - aplety
  - serwlety
  - JavaServer Pages (rozszerzenie pliku **`*.sqljsp`**)
  - JavaBeans
  - EJBs
  - procedury składowane Java

SQLJ w serwerze Oracle 8i/9i

- Kod SQLJ może być uruchamiany w serwerze Oracle9i (również 8i) w ramach:
  - procedur i funkcji składowanych
  - wyzwalaczy
  - Enterprise JavaBeans
  - obiektów CORBA
  - serwletów i JSP (poprzez moduł OSE)
- Serwer Oracle9i (8i) posiada wbudowany translator SQLJ
- Program SQLJ uruchomiony w serwerze:
  - posiada automatycznie otwarte połączenie z bieżącym serwerem
  - ma ustawiony domyślny kontekst połączenia na połączenie z bieżącym serwerem
  - nie może korzystać z trybu **`autoCommit`**

Rozszerzenia standardu SQLJ w Oracle9i (1)

- Obsługa specyficznej składni Oracle SQL
- Anonimowe bloki PL/SQL jako zagnieżdżone polecenia SQL w instrukcjach SQLJ
- Klasa **`oracle.sqlj.runtime.Oracle`** upraszczająca łączenie z bazą danych
  - automatyczna rejestracja sterownika Oracle JDBC (metody **`connect()`** i **`getConnection()`**)
  - automatyczne ustawienie domyślnego kontekstu (metoda **`connect()`**)

Rozszerzenia standardu SQLJ w Oracle9i (2)

- Opcja generacji kodu specyficznego dla Oracle
  - translacja programu SQLJ do kodu Java + JDBC
  - mniej komponentów aplikacji (brak profili)
  - większa efektywność aplikacji (bezpośrednie korzystanie z JDBC)
  - ograniczona przenaszalność
- Rozszerzenia dotyczące obsługiwanych typów danych
  - obsługa instancji typów klas otaczających dla specyficznych typów danych Oracle (**`oracle.sql.*`**)
  - obsługa instancji klas odpowiadających strukturalnym typom danych SQL 99 (custom Java classes)
  - instancje iteratorów i zbiorów wynikowych jako parametry wejściowe i wyjściowe w dowolnym kontekście
- Wsparcie dla dynamicznych instrukcji SQL (!)

Dynamiczny SQL w programach SQLJ – Dotychczasowe rozwiązania

- Wykorzystanie SQLJ dla statycznego SQL, a JDBC dla dynamicznego SQL w ramach jednego programu
  - instrukcje SQLJ i wywołania JDBC mogą się przeplatać
  - instrukcje SQLJ i wywołania JDBC mogą współdzielić połączenie z BD
  - możliwa jest konwersja między iteratorami SQLJ a zbiorami wynikowymi JDBC
- Wykorzystanie anonimowych bloków PL/SQL (tylko Oracle)

```
String polecenie;
polecenie = ...   // budowa treści dynamicznego polecenia SQL
#sql { begin
         execute immediate :polecenie;
       end };
```

(!) Powyższe rozwiązania nie pozwalają na weryfikację instrukcji SQL na etapie kompilacji (translacji)

Metawyrażenia związane w Oracle9i

- Oracle9i SQLJ pozwala na zagnieżdżanie w instrukcjach SQLJ tzw. metawyrażeń związanych (ang. meta bind expressions)

```
:{ wyr_związane_Java }
```

```
:{ wyr_związane_Java :: zastępczy_kod_SQL }
```

- W trakcie działania programu przed wykonaniem instrukcji SQLJ zagnieżdżone w niej metawyrażenia związane są zastępowane wartościami ich wyrażeń związanych Java (treść polecenia SQL ma charakter dynamiczny)
- Na etapie translacji metawyrażenia związane są zastępowane zastępczym kodem SQL (jeśli nie jest podany, nie jest przeprowadzana kontrola semantyczna w trybie on-line instrukcji SQLJ)

Dynamiczny SQL w Oracle9i SQLJ – Przykład 1

```
String table = "emp_boston";
String whereCond = "sal>2000";
#sql { DELETE FROM :{table :: emp}
       WHERE :{whereCond :: ename='BROWN'}
};
```

- Na etapie translacji:

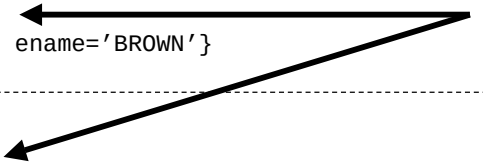
```
DELETE FROM emp
WHERE ename='BROWN';
```

- W trakcie działania programu:

```
DELETE FROM emp_boston
WHERE sal>2000;
```

Dynamiczny SQL w Oracle9i SQLJ – Przykład 2

```
String table = "emp_boston";
String whereCond = "sal>2000";
#sql { DELETE FROM :{table}
       WHERE :{whereCond :: ename='BROWN'}
};
```



- Na etapie translacji:

```
BRAK MOŻLIWOŚCI ANALIZY SEMANTYCZNEJ W TRYBIE ON-LINE
```

- W trakcie działania programu:

```
DELETE FROM emp_boston
WHERE sal>2000;
```

Zakres stosowalności metawyrażeń związanych

- Metawyrażenie związane może pojawić się w miejscu:
  - nazwy tabeli
  - nazwy kolumny na liście SELECT
  - całości lub części klauzuli WHERE
  - nazwy roli, schematu, katalogu lub pakietu w instrukcji DDL lub DML
  - wyrażenia lub literału SQL
- Metawyrażenie związane nie może:
  - rozpoczynać instrukcji SQL
  - zawierać słowa kluczowego INTO lub zwracać listy INTO polecenia SELECT INTO
  - występować w następujących instrukcjach i klauzulach: CALL, VALUES, SET, COMMIT, ROLLBACK, FETCH INTO, CAST

SQLJ w serwlecie – Przykład (1)

SqljEmpServlet.java

```
import java.sql.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.sqlj.runtime.Oracle;

public class SqljEmpServlet extends HttpServlet {
  #sql iterator EmpNameIter(String ename, double sal);
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
      Oracle.connect("jdbc:oracle:thin:@host:1521:SID",
                     "scott","tiger");
    }
    catch (Exception e) {}
  }
```

SQLJ w serwlecie – Przykład (2)

SqljEmpServlet.java

```
public void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException,IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<HTML><BODY>");
  try {
    EmpNameIter iter;
    #sql iter = {SELECT ename, sal FROM emp };
    while (iter.next()) {
      String n = iter.ename(); double s = iter.sal();
      out.println(n+": "+s+"<BR>");
    }
    iter.close();
  } catch (SQLException e) { out.println(e); }
   out.println("</BODY></HTML>");
}
```

SQLJ w serwlecie – Przykład (3)

SqljEmpServlet.java

```
 public void destroy()
  {
    try
    {
      Oracle.close();
    }
    catch (SQLException e)
    {
    }
  }
}
```

SQLJ w JSP – Przykład

SqljEmpJsp.sqljsp

```
<%@ page language="sqlj" import="java.sql.*,
                                 oracle.sqlj.runtime.Oracle" %>
<HTML><HEAD><TITLE>Employees (SQLJ)</TITLE></HEAD><BODY>
<%
  #sql iterator EmpNameIter(String ename, double sal);
  try {
    Oracle.connect("jdbc:oracle:thin:@host:1521:SID","scott","tiger");
    EmpNameIter iter;
    #sql iter = {SELECT ename, sal FROM emp };
    while (iter.next()) {
      String n = iter.ename();  double s = iter.sal();
      out.println(n+": "+s+"<BR>");
    }
    iter.close();  Oracle.close();
  } catch (Exception e) {  out.println(e); }
%>
</BODY></HTML>
```

SQLJ Część 1: Metody Java jako procedury SQL

- Statyczne metody klas Java jako procedury i funkcje składowane SQL
- Oferują przenaszalność kodu i bogactwo bibliotek Java
- Po załadowaniu klas Java do bazy danych należy wyspecyfikować nagłówki SQL dla metod
- Parametrom przekazywanym w SQL w trybach OUT i INOUT odpowiadają w metodach Java jednoelementowe tablice
- Operacje na bazie danych w ciele podprogramu realizowane poprzez JDBC i SQLJ:
  - SQLJ: automatycznie ustawiony domyślny kontekst
  - JDBC: otwarte domyślne połączenie, ale należy utworzyć obiekt **`Connection`**

```
Connection con =
DriverManager.getConnection("jdbc:default:connection:");
```

Nagłówki SQL dla metod Java – Przykłady (standard)

```
public class Finance {
  ...
  public static void raise (String who, double money)
           throws SQLException { ... }
  public static double maxSal (int deptNo, String [] eName)
           throws SQLException { ... }
}
```

```
sqlj.install_jar('file:~/classes/Fin.jar', 'fin_jar', 0);
```

```
CREATE PROCEDURE raise ( name VARCHAR, money NUMERIC)
MODIFIES SQL DATA
EXTERNAL NAME 'fin_jar:Finance.raise'
LANGUAGE JAVA PARAMETER STYLE JAVA;

CREATE FUNCTION max_sal ( dep INTEGER, name OUT VARCHAR)
RETURNS NUMERIC READS SQL DATA
EXTERNAL NAME 'fin_jar:Finance.maxSal'
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

```
CONTAINS SQL
READS SQL DATA
MODIFIES SQL DATA
NO SQL
```

Nagłówki SQL dla metod Java – Przykłady (Oracle)

```
public class Finance {
  ...
  public static void raise (String who, double money)
           throws SQLException { ... }
  public static double maxSal (int deptNo, String [] eName)
           throws SQLException { ... }
}
```

```
CREATE PROCEDURE raise ( name VARCHAR2, money NUMBER)
AS LANGUAGE JAVA
NAME 'Finance.raise(java.lang.String, double)';

CREATE FUNCTION max_sal ( dep NUMBER, name OUT VARCHAR2)
RETURN NUMBER
AS LANGUAGE JAVA
NAME 'Finance.maxSal(int, java.lang.String []) return double';
```

SQLJ Część 2: Klasy Java jako typy SQL

- Bezpośrednie wsparcie dla obiektów Java w bazach danych wspierających SQL
- Wykorzystanie klas Java jako typów:
  - kolumn w tabelach lub perspektywach
  - parametrów podprogramów składowanych
- Alternatywa lub uzupełnienie dla abstrakcyjnych typów danych SQL 99 (SQL3)

Klasy Java jako typy SQL przykład (standard)

```
CREATE TYPE person_t
EXTERNAL NAME 'Person' LANGUAGE JAVA
(
  ss_no INTEGER EXTERNAL NAME 'ssn',
  name VARCHAR(200) EXTERNAL NAME 'name',
  METHOD age() RETURNS INTEGER
         EXTERNAL NAME 'age'
);
/
```

- Klasa Java musi istnieć i być załadowana do bazy danych np.

```
sqlj.install_jar('file:~/classes/PersJar.jar',
                        'persons_classes_jar', 0);
```

Klasy Java jako typy SQL – przykład (Oracle)

```
CREATE TYPE person_t AS OBJECT
EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
(
  ss_no NUMBER(9) EXTERNAL NAME 'ssn',
  name VARCHAR2(200) EXTERNAL NAME 'name',
  MEMBER FUNCTION age RETURN NUMBER
         EXTERNAL NAME 'age () return int'
);
/
```

- Klasa Java musi istnieć i być załadowana do bazy danych
- W Oracle klasa Java służąca jako baza dla typu SQLJ musi implementować interfejs **`SQLData`** lub **`ORAData`**

Porównanie JDBC i SQLJ

	JDBC	**SQLJ**
Instrukcje SQL	dynamiczne	statyczne
Kontrola typów	słaba	silna
Kontrola poprawności SQL	runtime	translator
Składnia	API	zwarta
Standard	Sun	ANSI / ISO
Przenaszalność	Tak	Tak