

## IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

## KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

## TWÓJ KOSZYK

DODAJ DO KOSZYKA

## CENNIK I INFORMACJE

ZAMÓW INFORMACJE  
O NOWOŚCIACH

ZAMÓW CENNIK

## CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

# Java. Rozmówki

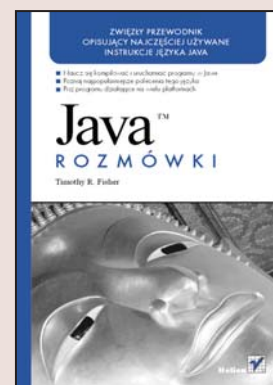
Autor: Timothy R. Fisher

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-0949-9

Tytuł oryginału: [Java Phrasebook](#)

Format: B6, stron: 272



### Zwięzły przewodnik opisujący najczęściej używane instrukcje języka Java

- Naucz się kompilować i uruchamiać programy w Javie
- Poznaj najpopularniejsze polecenia tego języka
- Pisz programy działające na wielu platformach

Już od wielu lat Java jest jednym z najpopularniejszych języków programowania, a znający ją programiści są poszukiwani i cenieni na rynku pracy. Jej atrakcyjność wynika głównie z tego, że kod napisany w tym języku można uruchamiać na wielu platformach, włączając w to różne systemy operacyjne, a także urządzenia przenośne, na przykład telefony komórkowe. Prosty jest także dostęp do wielu narzędzi oraz bezpłatnych bibliotek ułatwiających programowanie w Javie. Ponadto język ten otwiera wielkie możliwości w zakresie tworzenia aplikacji sieciowych.

„Java. Rozmówki” przedstawia skuteczne rozwiązania najczęściej występujących problemów i pomoże Ci błyskawicznie rozpocząć programowanie w tym języku. Dowiesz się, jak kompilować i uruchamiać programy w Javie. Nauczysz się manipulować ciągami, strukturami danych i datami oraz stosować wyrażenia regularne. Poznasz operacje wejścia i wyjścia, a także sposoby pracy z katalogami i plikami. Przeczytasz o aplikacjach sieciowych działających zarówno po stronie klienta, jak i po stronie serwera. Opanujesz efektywne techniki pracy z bazami danych i korzystanie z formatu XML.

- Kompilowanie i uruchamianie programów w Javie
- Współpraca ze środowiskiem
- Obsługa ciągów i liczb
- Korzystanie ze struktur danych
- Operacje wejścia i wyjścia
- Manipulowanie plikami i katalogami
- Praca z klientami i serwerami sieciowymi
- Komunikacja z bazami danych i używanie XML
- Programowanie wielowątkowe

**Dzięki treściwym rozmówkom błyskawicznie poznasz  
praktyczne podstawy programowania w Javie**



# Spis treści

<b>O autorze</b> .....	11
<b>Wprowadzenie</b> .....	13
<b>1 Podstawy</b> .....	17
Kompilowanie programu w Javie .....	19
Uruchamianie programu w Javie .....	21
Ustawianie zmiennej CLASSPATH .....	22
<b>2 Interakcje z otoczeniem</b> .....	25
Odczytywanie zmiennych środowiskowych .....	26
Odczytywanie i ustawianie właściwości systemowych .....	27
Przetwarzanie argumentów wywołania programu .....	28
<b>3 Manipulowanie ciągami</b> .....	31
Porównywanie ciągów .....	32
Wyszukiwanie i wyłuskiwanie podciągów .....	35
Przetwarzanie ciągu znak po znaku .....	37
Odwracanie znaków w ciągu .....	38
Odwracanie wyrazów w ciągu .....	38

## Spis treści

Zamiana wszystkich liter w ciągu na wielkie albo na małe .....	40
Usuwanie zbędnych spacji z początku i końca ciągu .....	41
Przetwarzanie ciągu wyrazów oddzielanych przecinkami .....	42
<b>4 Praca ze strukturami danych .....</b>	<b>47</b>
Zmiana rozmiaru tablicy .....	48
Przeglądanie kolekcji .....	49
Tworzenie odwzorowania .....	51
Porządkowanie kolekcji .....	53
Wyszukiwanie obiektu w kolekcji .....	55
Konwersja kolekcji na tablicę .....	58
<b>5 Daty i godziny .....</b>	<b>59</b>
Określanie bieżącej daty .....	60
Konwersja pomiędzy klasami Date i Calendar .....	61
Wyświetlanie daty (godziny) w zadanym formacie .....	62
Wyodrębnianie dat z ciągów znaków .....	65
Dodawanie i odejmowanie obiektów Date bądź Calendar ....	67
Obliczanie różnicy pomiędzy dwiema datami .....	68
Porównywanie dat .....	69
Określanie numeru dnia w tygodniu, miesiąca w roku albo tygodnia w roku .....	71
Obliczanie czasu trwania operacji .....	72
<b>6 Dopasowywanie wzorców za pomocą     wyrażeń regularnych .....</b>	<b>75</b>
Wyrażenia regularne w Javie .....	76
Wyszukiwanie i dopasowywanie tekstu za pomocą wyrażenia regularnego .....	79
Zastępowanie dopasowanego tekstu .....	82
Wyszukiwanie wszystkich wystąpień wzorca .....	84
Wypisywanie wierszy zawierających wzorzec .....	85
Dopasowywanie znaków nowego wiersza .....	86

<b>7 Liczby</b> .....	<b>89</b>
Sprawdzanie, czy ciąg zawiera poprawną liczbę .....	90
Porównywanie liczb zmiennoprzecinkowych .....	91
Zaokrąglanie liczb zmiennoprzecinkowych .....	93
Formatowanie liczb .....	94
Formatowanie wartości monetarnych .....	97
Konwersja dziesiętnej liczby całkowitej na zapis dwójkowy, ósemkowy bądź szesnastkowy .....	98
Generowanie liczb losowych .....	98
Funkcje trygonometryczne .....	100
Obliczanie logarytmów .....	100
<b>8 Wejście i wyjście</b> .....	<b>103</b>
Odczytywanie tekstu ze standardowego wejścia programu .....	104
Wypisywanie do standardowego wyjścia .....	105
Formatowanie wyjścia .....	106
Otwieranie pliku identyfikowanego przez nazwę .....	112
Wczytanie zawartości pliku do tablicy bajtów .....	112
Wczytywanie danych binarnych .....	113
Przesuwanie pozycji w pliku .....	114
Odczytywanie archiwum JAR albo ZIP .....	115
Tworzenie archiwum ZIP .....	116
<b>9 Praca z katalogami i plikami</b> .....	<b>119</b>
Tworzenie pliku .....	120
Zmiana nazwy pliku lub katalogu .....	122
Usuwanie pliku albo katalogu .....	123
Zmiana atrybutów pliku .....	124
Ustalanie rozmiaru pliku .....	125
Sprawdzanie obecności pliku lub katalogu .....	126

## Spis treści

Przenoszenie pliku lub katalogu .....	127
Ustalanie bezwzględnej ścieżki dostępu na podstawie ścieżki względnej .....	128
Ustalanie, czy ścieżka dostępu określa plik, czy katalog .....	129
Wypisywanie zawartości katalogu .....	131
Tworzenie nowego katalogu .....	135
<b>10 Klienci sieciowe .....</b>	<b>137</b>
Nawiązywanie połączenia z serwerem .....	138
Ustalanie adresów IP i nazw domenowych .....	139
Obsługa błędów sieciowych .....	140
Wczytywanie danych (tekstu) z gniazda .....	142
Zapisywanie do gniazda .....	143
Wczytywanie danych binarnych .....	144
Zapisywanie danych binarnych .....	146
Wczytywanie danych serializowanych .....	148
Zapis serializowanego obiektu .....	149
Pobieranie strony WWW przez HTTP .....	151
<b>11 Serwery sieciowe .....</b>	<b>155</b>
Uruchamianie serwera i przyjmowanie żądań .....	156
Zwracanie odpowiedzi .....	157
Odsyłanie obiektu .....	159
Obsługa wielu klientów .....	161
Udostępnianie treści przez HTTP .....	163
<b>12 Wysyłanie i odbieranie poczty elektronicznej .....</b>	<b>167</b>
Przegląd JavaMail API .....	168
Wysyłanie poczty .....	169
Wysyłanie wiadomości MIME .....	172
Odbieranie poczty .....	175

<b>13 Dostęp do baz danych</b> .....	<b>179</b>
Nawiązywanie połączenia z bazą danych przez JDBC .....	180
Wysyłanie zapytania przez JDBC .....	183
Stosowanie zapytań sparametryzowanych .....	185
Pobieranie wyników zapytania .....	187
Uruchamianie procedury składowanej .....	189
<b>14 XML w Javie</b> .....	<b>193</b>
Analiza XML za pomocą SAX .....	195
Analiza XML za pomocą DOM .....	198
Weryfikowanie poprawności dokumentu względem DTD ...	201
Tworzenie dokumentu XML za pomocą DOM .....	203
Przekształcanie XML za pomocą XSLT .....	206
<b>15 Stosowanie wątków</b> .....	<b>209</b>
Uruchamianie wątku .....	210
Zatrzymywanie wątku .....	213
Oczekiwanie zakończenia wątku .....	214
Synchronizowanie wątków .....	216
Wstrzymywanie wątku .....	220
Wypisywanie listy wątków .....	222
<b>16 Programowanie dynamiczne z introspekcją</b> .....	<b>225</b>
Ustalanie klasy obiektu .....	227
Ustalanie nazwy klasy .....	228
Ujawnianie modyfikatorów klasy .....	229
Ustalanie klasy bazowej .....	230
Ustalanie interfejsów implementowanych przez klasę .....	232
Ujawnianie składowych klasy .....	233
Ujawnianie konstruktorów klasy .....	234
Ujawnianie informacji o metodach .....	236

## Spis treści

Odczytywanie wartości składowych .....	239
Zapisywanie wartości składowych klas .....	240
Wywoływanie metod .....	242
Wczytywanie klasy i dynamiczne tworzenie jej obiektów ....	244
<b>17 Tworzenie pakietów i dokumentacji .....</b>	<b>247</b>
Tworzenie pakietu .....	248
Dokumentowanie klas za pomocą JavaDoc .....	251
Archiwizacja klas w pakiecie JAR .....	254
Uruchamianie programu z archiwum JAR .....	255
<b>Skorowidz .....</b>	<b>257</b>

# Manipulowanie ciągami

**Z**naczną część operacji wykonywanych w programach — niezależnie od języka programowania — polega na manipulowaniu ciągami znaków, ponieważ poza danymi liczbowymi nasze dane mają zazwyczaj właśnie postać ciągów. Nawet dane liczbowe są często traktowane jako proste ciągi znaków. Trudno więc wyobrazić sobie jakiś skończony program, który wcale nie korzystałby z ciągów.

Ćwiczenia zebrane w tym rozdziale ilustrują sposób wykonania najbardziej typowych operacji na ciągach znaków. Język Java oferuje mocne narzędzia do obsługi ciągów znaków i ich przetwarzania. W przeciwieństwie do ciągów w języku C, w Javie ciągi znaków są typami wbudowanymi. Do przechowywania i reprezentowania ciągów znaków w Javie zaimplementowano klasę `String`. Ciągi w Javie nie powinny być traktowane jako proste tablice



## Porównywanie ciągów

znaków, znane z języka C. Wszędzie tam, gdzie w Javie ma być przechowany ciąg znaków, powinieneś zamiast prostej tablicy zastosować klasę `String`.

Ważną cechą klasy `String` w języku Java jest niezmiennosc obiektów tej klasy — raz utworzony ciąg pozostaje niezmienny przez cały swój czas życia w programie. Po utworzeniu obiektu ciągu nie można go zmieniać. Co najwyżej można przypisać ciąg do innego obiektu klasy `String`, ale zmiany zawartości ciągu nie można wykonać na jednym egzemplarzu `String`. Z tego względu w klasie `String` nie znajdziesz żadnej metody ustawiającej wartość ciągu. Jeśli chcesz utworzyć ciąg, do którego będziesz mógł dopisywać dane, to zamiast klasy `String` powinieneś w JDK 1.5 wykorzystać klasę `StringBuilder`; w starszych wydaniach Javy jej odpowiednikiem byłaby klasa `StringBuffer`. Obie klasy przewidują zmienność swoich obiektów, więc dają możliwość zmiany zawartości reprezentowanych ciągów. Słowem, konstruowanie ciągów odbywa się typowo za pośrednictwem klasy `StringBuilder` albo `StringBuffer`, natomiast do przechowywania ciągów i ich przekazywania najlepsza jest klasa `String`.

## Porównywanie ciągów

```
boolean result = str1.equals(str2);  
boolean result2 = str1.equalsIgnoreCase(str2);
```

Jeśli oba porównywane obiekty ciągów `str1` i `str2` będą miały tę samą zawartość, zmienne `result` i `result2`

przyjmą wartości logiczne „prawda” (true). Jeśli ciągi będą różne, result i result2 przyjmą wartość logiczną „fałsz” (false). Pierwsza z metod porównujących — equals() — porównuje odpowiednie znaki obu ciągów z uwzględnieniem wielkości liter. Druga metoda — equalsIgnoreCase() — porównuje ciągi bez uwzględniania wielkości liter i zwraca true, jeśli porównywane ciągi mają tę samą „treść”, z pominięciem różnic w wielkości poszczególnych liter.

Porównywanie ciągów bywa przyczyną wielu błędów, popełnianych zwłaszcza przez niedoświadczonych programistów języka Java. Otóż programista-nowicjusz często próbuje porównać ciągi za pomocą operatora porównania ==. Tymczasem operator ten, gdy zostanie użyty z obiektami String, porównuje referencje obiektów, a nie ich zawartość. Z tego powodu nawet dwa identyczne pod względem wartości obiekty ciągów będą różne, jeśli porównanie odbędzie się za pośrednictwem operatora ==.

Metoda equals() klasy String porównuje zawartość ciągów, a nie jedynie referencje obiektów. Właśnie tak powinno się porównywać ciągi w zdecydowanej większości przypadków. Spójrz na poniższy przykład:

```
String name1 = new String("Tomek");
String name2 = new String("Tomek");
if (name1 == name2) {
    System.out.println("Ciągi są równe.");
}
else {
    System.out.println("Ciągi są różne.");
}
```

### Porównywanie ciągów

Na wyjściu powyższego programu powinieneś zobaczyć napis:

Ciągi są różne.

Spróbujmy teraz porównać te same ciągi metodą `equals()`:

```
String name1 = new String("Tomek");
String name2 = new String("Tomek");
if (name1.equals(name2)) {
    System.out.println("Ciągi są równe.");
}
else {
    System.out.println("Ciągi są różne.");
}
```

Tym razem program powinien wypisać:

Ciągi są równe.

Inną metodą związaną z porównywaniem obiektów klasy `String` jest metoda `compareTo()`. Służy ona do leksyko-graficznego porównywania dwóch ciągów i zwraca nie wartość logiczną, ale liczbową: dodatnią, ujemną albo zerową. Wartość 0 jest zwracana jedynie dla takich dwóch ciągów, dla których metoda `equals()` dałaby wartość `true`. Wartość ujemna wskazuje, że ciąg, na rzecz którego wywołano metodę, jest „pierwszy”, to znaczy alfabetycznie poprzedza drugi ciąg uczestniczący w porównaniu, przekazany w wywołaniu metody. Z kolei wartość dodatnia oznacza, że ciąg, na rzecz którego wywołano metodę, jest alfabetycznie za ciągiem przekazany w wywołaniu. Gwoli ścisłości, porównanie odbywa się na bazie wartości `Unicode` poszczególnych, odpowiadających sobie

znaków porównywanych ciągów. Metoda `compareTo()` ma również swój odpowiednik ignorujący wielkość liter — `compareToIgnoreCase()`. Działa on tak samo, tyle że przy porównywaniu znaków nie uwzględnia wielkości liter. Spójrzmy na następujący przykład:

```
String name1 = "Kowalski";
String name2 = "Nowak";
int result = name1.compareTo(name2);
if (result == 0) {
    System.out.println("Nazwiska są identyczne.");
}
else if (result > 0) {
    System.out.println("Nazwisko name1 jest
    ─alfabetycznie pierwsze.");
}
else if (result < 0) {
    System.out.println("Nazwisko name1 jest
    ─alfabetycznie drugie.");
}
```

W przypadku tego programu powinniśmy otrzymać na wyjściu taki komunikat:

```
Nazwisko name1 jest alfabetycznie drugie.
```

## Wyszukiwanie i wyłuskiwanie podciągów

```
int result = string1.indexOf(string2);
int result = string1.indexOf(string2, 5);
```

Wywołanie pierwszej z powyższych metod powinno umieścić w zmiennej `result` indeks pierwszego wystąpienia

### Wyszukiwanie i wyluskiwanie podciągów

podciągu `string2` w ciągu `string1`. Jeśli w ciągu `string1` nie ma ciągu `string2`, metoda zwróci wartość `-1`.

W drugiej z powyższych metod wartość zwracana będzie zawierać indeks pierwszego wystąpienia podciągu `string2` w ciągu `string1`, ale za piątym znakiem w `string1`. Wartością drugiego argumentu może być dowolna liczba większa od zera. Jeśli będzie większa od długości przeszukiwanego ciągu, metoda zwróci `-1`.

Oprócz wyszukiwania podciągu w ciągu trzeba czasem dowiedzieć się, gdzie znajduje się interesujący nas podciąg, i przy okazji wyluskać go z przeszukiwanego ciągu. Jak dotąd umiemy jedynie zlokalizować potrzebny podciąg w ciągu. Kiedy ustalisz jego indeks, możesz go wyluskać za pomocą metody `substring()` klasy `String`. Metoda `substring()` jest przeciążona, co oznacza, że można ją wywoływać na kilka sposobów. Jeden z nich polega na przekazaniu indeksu podciągu do wyluskania. W tej wersji metoda zwraca podciąg zaczynający się od wskazanego znaku i rozciągający się aż do końca ciągu źródłowego. Kolejny sposób to wywołanie `substring()` z dwoma argumentami: indeksem początku i indeksem końca ciągu.

```
String string1 = "Mój adres to Polna 33";  
String address = string1.substring(13);  
System.out.println(address);
```

W przypadku takiego programu powinniśmy otrzymać na wyjściu napis:

```
Polna 33
```

Na 13. pozycji w ciągu `string1` znajduje się `P`; to jest początek naszego podciągu. Zauważ, że ciągi znaków są zawsze indeksowane od zera, a ostatni znak ciągu znajduje się pod indeksem `-1` (koniec ciągu).

## Przetwarzanie ciągu znak po znaku

```
for (int index = 0; index < string1.length(); index++) {  
    char aChar = string1.charAt(index);  
}
```

Metoda `charAt()` pozwala na pozyskanie pojedynczego znaku wyłuskanego z ciągu spod wskazanej pozycji. Znaki są indeksowane w ciągu od zera, to znaczy mają numery od 0 do liczby równej długości ciągu zmniejszonej o jeden. Powyższa pętla przetwarza kolejne znaki ciągu `string1`.

Alternatywna metoda operowania na poszczególnych znakach wykorzystuje klasę `StringReader`, jak tutaj:

```
StringReader reader = new StringReader(string1);  
int singleChar = reader.read();
```

Mechanizm ten polega na wyłuskiwaniu poszczególnych znaków ciągu za pomocą metody `read()` klasy `StringReader`; znak jest zwracany jako liczba całkowita. Pierwsze wywołanie `read()` zwraca liczbę reprezentującą pierwszy znak ciągu, za każdym kolejnym wywołaniem zwracana jest reprezentacja następnego znaku w ciągu.

Odwracanie znaków w ciągu

## Odwracanie znaków w ciągu

```
String letters = "ABCDEF";  
StringBuffer lettersBuff = new StringBuffer(letters);  
String lettersRev = lettersBuff.reverse().toString();
```

Klasa `StringBuffer` zawiera metodę `reverse()` zwracającą kopię ciągu zawartego w obiekcie `StringBuffer` z odwróconą kolejnością znaków. Obiekt `StringBuffer` daje się łatwo konwertować na ciąg typu `String` — służy do tego metoda `toString()` klasy `StringBuffer`. Dlatego za pomocą tymczasowego, roboczego obiektu `StringBuffer` można w łatwy sposób utworzyć lustrzane odbicie pierwotnego ciągu.

Jeśli korzystasz z JDK 1.5, możesz zamiast klasy `StringBuffer` wykorzystać klasę `StringBuilder`. Klasa ta ma interfejs zgodny z interfejsem klasy `StringBuffer`. Klasa `StringBuilder` oferuje większą wydajność, ale jej metody nie są synchronizowane, co oznacza, że obiekty klasy nie są zabezpieczone pod kątem wykonania wielowątkowego. W aplikacjach wielowątkowych należałoby więc stosować klasę `StringBuffer`.

## Odwracanie wyrazów w ciągu

```
String test = "Odwróć ten ciąg znaków";  
Stack stack = new Stack();  
StringTokenizer strTok = new StringTokenizer(test);  
  
while (strTok.hasMoreTokens()) {  
    stack.push(strTok.nextElement());  
}
```

```
}  
StringBuffer revStr = new StringBuffer();  
while (!stack.empty()) {  
    revStr.append(stack.pop());  
    revStr.append(" ");  
}  
System.out.println("Ciąg pierwotny: " + test);  
System.out.println("Ciąg odwrócony: " + revStr);
```

W przypadku powyższego fragmentu programu powinniśmy otrzymać w wyniku taki zestaw napisów:

```
Ciąg pierwotny: Odwróć ten ciąg znaków  
Ciąg odwrócony: znaków ciąg ten Odwróć
```

Jak widać, odwracanie ciągu wyraz po wyrazie jest nieco bardziej skomplikowane niż odwracanie kolejności poszczególnych znaków. To dlatego, że odwracanie znaków w ciągu jest implementowane w ramach implementacji klasy `StringBuffer`, ale klasa ta nijak nie obsługuje odwracania kolejności poszczególnych wyrazów. Musimy to zadanie oprogramować samodzielnie, wykorzystując do tego celu klasy `StringTokenizer` i `Stack`. `StringTokenizer` służy do wyluskiwania z ciągu kolejnych wyrazów rozdzielanych wskazanym separatorem; wyrazy te są odkładane na stos reprezentowany przez obiekt klasy `Stack`. Po przetworzeniu w ten sposób całego ciągu przeglądamy elementy stosu, zdejmując z niego kolejne słowa w kolejności odwrotnej do kolejności wstawiania na stos. Stos jest wcieleniem kolejki LIFO — *last in, first out*, czyli „ostatni na wejściu, pierwszy na wyjściu”. Wykorzystanie stosu znakomicie ułatwia operację odwracania.



## Zamiana wszystkich liter w ciągu na wielkie albo na małe

Nieco więcej informacji o klasie `StringTokenizer` znajdziesz jeszcze w tym rozdziale, przy okazji omawiania przetwarzania ciągu wyrazów rozdzielanych przecinkami.

### UWAGA

Jeśli korzystasz z JDK 1.5, powinieneś koniecznie zapoznać się z pewną nowinką w postaci klasy `Scanner`. Co prawda, nie omawiam jej tu, ale jest dość ciekawa, ponieważ implementuje prosty skaner tekstowy zdolny do analizowania i wyluskiwania z ciągów wartości typów prostych i podciągów dopasowywanych za pomocą wyrażeń regularnych.

## Zamiana wszystkich liter w ciągu na wielkie albo na małe

```
String string = "Ciąg zawiera maŁE i WIEłkie litery";  
String string2 = string.toUpperCase();  
String string3 = string.toLowerCase();
```

Obie metody służą do zamiany wielkości liter w ciągu znaków — albo na same małe, albo na same wielkie litery. Obie metody zwracają przerobione kopie ciągu, ale nie modyfikują ciągu pierwotnego. Pierwotny ciąg zachowuje oryginalne wielkości znaków.

Metody te znajdują praktyczne zastosowania choćby w operacjach wymagających składowania danych w bazach danych. Niektóre pola w tabelach mogą być zaprojektowane tak, aby zapisywane w nich ciągi zawierały wyłącznie

wielkie albo wyłącznie małe litery. Dzięki omawianym metodom zamiana wielkości liter nie stanowi żadnego problemu.

Zmiana wielkości liter przydaje się także przy przetwarzaniu identyfikatorów kont użytkowników. Zazwyczaj identyfikator użytkownika jest w bazie danych polem o wartościach z dowolnymi wielkościami znaków, w przeciwieństwie do hasła, w którym wielkość znaków ma kolosalne znaczenie. Przy porównywaniu podanego przez użytkownika identyfikatora z tym, który zapisany jest w bazie danych, najlepiej, aby oba porównywane ciągi zostały uprzednio skonwertowane na odpowiednią wielkość liter. Alternatywą byłoby zastosowanie metody `equalsIgnoreCase()` klasy `String`, która realizuje porównanie bez uwzględniania wielkości liter.

## Usuwanie zbędnych spacji z początku i końca ciągu

```
String result = str.trim();
```

Metoda `trim()` usuwa z ciągu zarówno początkowe, jak i końcowe spacje oraz inne znaki odstępów (ang. *white-spaces*) i zwraca „obrany” ciąg. Pierwotny ciąg pozostaje oczywiście niezmienny. Jeśli w ciągu nie ma żadnych początkowych ani końcowych znaków odstępów do usunięcia,

## Przetwarzanie ciągu wyrazów oddzielanych przecinkami

metoda zwraca po prostu pierwotny ciąg. Do znaków odstępów zaliczane są między innymi znaki spacji i tabulacji<sup>1</sup>.

Przydaje się to bardzo przy porównywaniu ciągów wprowadzanych przez użytkownika na wejście programu z ciągami zaszytymi w programie albo odczytywanymi ze znanych źródeł. Programista często całymi godzinami ślęczy nad nie działającym kodem tylko po to, by przekonać się, że to, co wprowadza, niedokładnie zgadza się z oczekiwaniami programu, a różnica polega np. na umieszczeniu na wejściu niewinnej, początkowej spacji. Obcięcie zbędnych znaków odstępów eliminuje takie problemy.

## Przetwarzanie ciągu wyrazów oddzielanych przecinkami

```
String str = "timothy,kerry,timmy,camden";  
String[] results = str.split(",");
```

Metoda `split()` wywołana na rzecz obiektu klasy `String` przyjmuje w wywołaniu ciąg wyrażenia regularnego reprezentujący separator wyrazów ciągu i zwraca tablicę obiektów `String` wyluskanych z ciągu źródłowego na

---

<sup>1</sup> Ściśle mówiąc, do wycinanych znaków zaliczają się wszystkie znaki o kodach mniejszych od `\u0020`, który to kod reprezentuje właśnie spację. Znak tabulacji ma kod `\u0007`; znak nowego wiersza to `\u0010`; znak `\u0013` to znak powrotu karetki itd. — *przyt. tłum.*

podstawie wyrażenia regularnego separatora. Dzięki tej metodzie przetwarzanie ciągów oddzielanych przecinkami jest zupełnie proste. W tym przykładzie najwyczejniej przekazujemy do metody `split()` ciąg separatora, a w odpowiedzi otrzymujemy tablicę ciągów zawierających wyrazy rozpoznane w ciągu źródłowym (`str`) pomiędzy przecinkami. Tablica powinna składać się z następujących elementów:

```
results[0] = timothy
results[1] = kerry
results[2] = timmy
results[3] = camden
```

Inną klasą bardzo przydatną do wykonywania podziału ciągów jest klasa `StringTokenizer`. Wykorzystamy ją zamiast metody `split()` do powtórzenia ostatniego ćwiczenia:

```
String str = "timothy,kerry,timmy,camden";
StringTokenizer st = new StringTokenizer(str, ",");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Ten fragment programu powinien spowodować wypisanie na wyjściu następujących podciągów (każdy w osobnym wierszu):

```
timothy
kerry
timmy
camden
```

## Przetwarzanie ciągu wyrazów oddzielanych przecinkami

Zauważ, że przecinki oddzielające wyrazy zostały pominięte i nie widać ich na wyjściu.

Obiekt klasy `StringTokenizer` można konstruować z jednym, dwoma albo trzema argumentami. W wersji z jednym argumentem argument ten jest traktowany jako ciąg do podziału na wyrazy. Za separator służy w takim przypadku separator domyślny, czyli zestaw znaków oddzielających wyrazy w języku naturalnym: `" \t\n\r\f"`, a więc spacja, znak tabulacji, znak nowego wiersza, znak powrotu karetki i znak wysuwu strony.

Drugi sposób tworzenia obiektu klasy `StringTokenizer` polega na przekazaniu do konstruktora dwóch argumentów. Pierwszy to ciąg do podziału, a drugi to ciąg określający separator, wedle którego dokonany zostanie podział. Przekazanie własnego separatora unieważnia separator domyślny — podział odbywa się wyłącznie na podstawie separatora określonego w wywołaniu konstruktora.

Trzeci sposób konstrukcji obiektu `StringTokenizer` polega na przekazaniu trzech argumentów. Trzeci argument określa, czy do wyodrębnianych wyrazów należy zaliczać również same separatory, czy je pomijać. Argument ten jest wartością boole'owską; wartość `true` oznacza, że w tablicy wyodrębnionych ciągów pojawią się również rozdzielające je symbole podziału. Domyślna wartość to `false` — przy tej wartości separatory są pomijane.

W ramach lektury uzupełniającej koniecznie zajrzyj do rozdziału szóstego, gdzie omawiamy wyrażenia regularne. Pojawiły się one w Javie w wydaniu JDK 1.4 i można nimi zastąpić wiele przypadków użycia klasy `StringTokenizer`. Oficjalna dokumentacja Javy stwierdza, że klasa `StringTokenizer` jest włączana do implementacji Javy ze względu na zgodność wstecz, i zniechęca do jej stosowania w nowszych projektach. Tam, gdzie to możliwe, należy ją zastępować metodą `split()` klasy `String` albo stosować wyrażenia regularne.