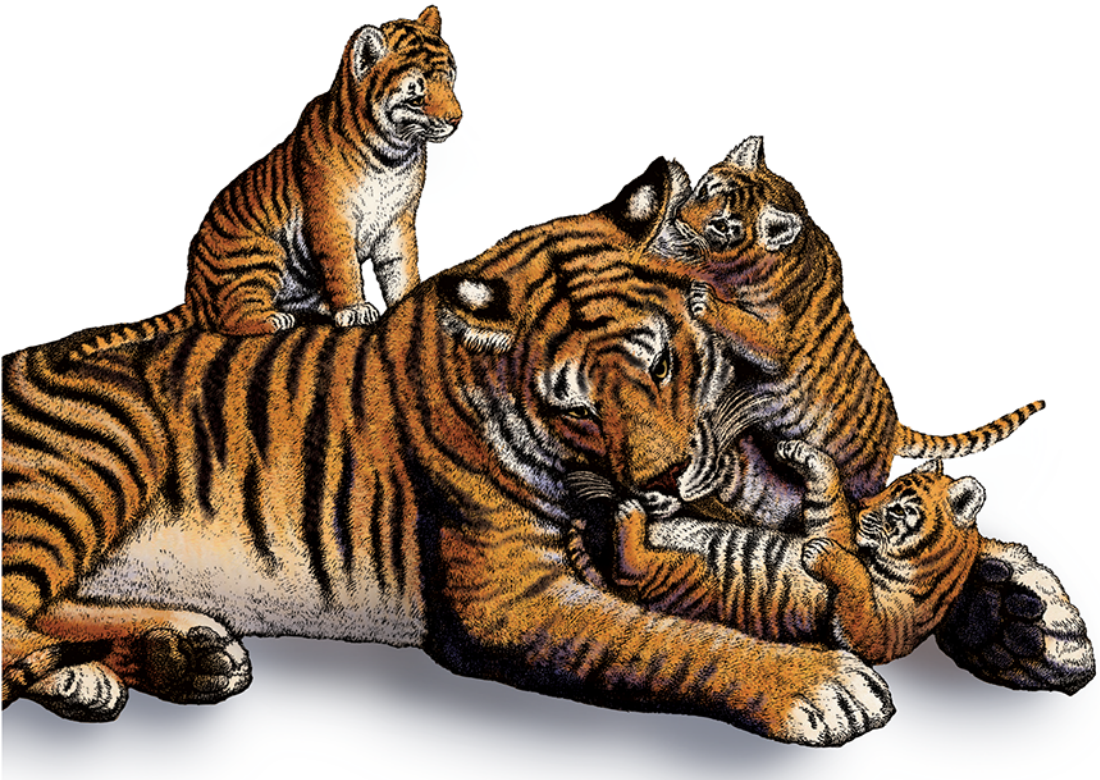
O'REILLY®

Wydanie V

Nauka Javy

Wprowadzenie do tworzenia aplikacji do rzeczywistych zastosowań



Marc Loy
Patrick Niemeyer
Daniel Leuck

Helion

Tytuł oryginału: Learning Java: An Introduction to Real-World Programming with Java, 5th Edition

Tłumaczenie: Lech Lachowski, z wykorzystaniem fragmentów książki "Java. Wprowadzenie" w przekładzie Rafała Jońcy

ISBN: 978-83-283-7128-6

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/najav5.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/najav5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 8.

Praca z tekstem i podstawowe narzędzia

Jeśli czytasz rozdziały tej książki po kolei, poznałeś już podstawowe konstrukcje języka Java, w tym jego aspekty obiektowe i niektóre kwestie związane z wątkami. Nadszedł czas zmienić nieco obszar naszych zainteresowań i rozpocząć omawianie interfejsu programistycznego aplikacji (ang. *application programming interface* — API) Javy, czyli kolekcji klas, które składają się na standardowe pakiety Javy i są dostarczane z każdą implementacją tego języka. Podstawowe pakiety Javy są jedną z jego najbardziej wyróżniających cech. Wiele innych języków obiektowych posiada podobne funkcjonalności, ale żaden nie ma tak szerokiego zestawu znormalizowanych interfejsów API i narzędzi jak Java. Jest to zarówno powód, jak i efekt sukcesu Javy.

Łańcuchy znaków — klasa String

Zaczniemy od przyjrzenia się klasie `String` Javy (a mówiąc ściślej, klasie `java.lang.String`). Ponieważ praca z typami `String` jest tak fundamentalna, istotą jest dobre zrozumienie sposobu ich zaimplementowania oraz tego, co można z nimi zrobić. Obiekt `String` hermetyzuje sekwencję znaków zakodowanych w standardzie Unicode. Wewnętrznie znaki te są przechowywane w zwykłej tablicy Javy, ale obiekt `String` zazdrośnie jej strzeże i umożliwia dostęp do niej tylko poprzez własny interfejs API. Jest to związane z koncepcją, że obiekty `String` są **niemutowalne** (ang. *immutable*), czyli niezmienne. Po utworzeniu obiektu `String` nie można już zmienić jego wartości. Wydaje się, że wiele operacji na obiektach `String` zmienia znaki lub długość łańcucha znaków, ale tak naprawdę zwracają one po prostu nowy obiekt `String`, który kopiuje wymagane znaki z oryginału lub wewnętrznie się do nich odwołuje. Implementacje Javy starają się konsolidować identyczne łańcuchy znaków używane w tej samej klasie we współdzieloną pulę i udostępniać części obiektów `String`, gdzie tylko jest to możliwe.

Pierwotną motywacją dla zastosowania takiego rozwiązania była wydajność. Niemutowalne obiekty `String` mogą oszczędzać pamięć i być optymalizowane pod kątem szybkości przez maszynę wirtualną Javy. Drugą stroną medalu jest to, że programista powinien mieć podstawową wiedzę na temat klasy `String`, aby uniknąć tworzenia nadmiernej liczby jej obiektów w miejscach, w których wydajność ma duże znaczenie. Było to szczególnie ważne w przeszłości, gdy maszyny wirtualne

działały powoli i słabo radziły sobie z obsługiwaniem pamięci. Obecnie używanie łańcuchów znaków zwykle nie stanowi problemu dla ogólnej wydajności prawdziwej aplikacji[1].

Konstruowanie łańcuchów znaków

Literalne łańcuchy znaków zdefiniowane w kodzie źródłowym są deklarowane za pomocą pary podwójnych cudzysłowów i mogą być przypisywane do zmiennej typu `String`:

```
String quote = "Być albo nie być";
```

Java automatycznie konwertuje literał łańcucha znaków na obiekt `String` i przypisuje go do danej zmiennej.

W Javie obiekty `String` śledzą własną długość, więc nie wymagają specjalnych terminatorów. Długość obiektu `String` można pobrać za pomocą metody `length()`. Natomiast za pomocą metody `isEmpty()` można przetestować, czy obiekt `String` nie ma zerowej długości (czy nie jest pusty):

```
int length = quote.length();
boolean empty = quote.isEmpty();
```

Do konkatenacji (łączenia) łańcuchów znaków można wykorzystywać jedyny przeciążony operator w Javie, czyli operator +. Poniższy kod generuje równoważne łańcuchy znaków:

```
String name = "Jan " + "Kowalski";
String name = "Jan ".concat("Kowalski");
```

W plikach źródłowych Javy literały łańcuchów znaków nie mogą (jeszcze[2]) obejmować wielu linii kodu, ale możemy konkatenować te linie, aby uzyskać ten sam efekt[3]:

```
String poem =
    "Brzdęśniało już; ślimonne prztowie\n" +
    "Wyrło i warło się w gulbieży;\n" +
    "Zmimszałe ćwiły borogowie\n" +
    "I rcie grdypały z mrzerzy.\n";
```

Zazwyczaj w kodzie źródłowym nie osadza się dłuższych tekstów. W rozdziale 11. omówimy sposoby pobierania obiektów `String` z plików i adresów URL.

Poza tworzeniem łańcuchów znaków z wyrażeń literalnych obiekty `String` można również konstruować bezpośrednio z tablic znaków:

```
char [] data = new char [] { 'L', 'e', 'm', 'i', 'n', 'g' };
String lemming = new String( data );
```

Można także zbudować obiekt `String` z tablicy bajtów:

```
byte [] data = new byte [] { (byte)97, (byte)98, (byte)99 };
String abc = new String(data, "ISO8859_1");
```

[1] Gdy masz wątpliwości, przeprowadź pomiary! Jeśli Twój kod manipulujący łańcuchami znaków jest czysty i łatwy do zrozumienia, nie poprawiaj go, dopóki ktoś nie udowodni Ci, że działa za wolno. Istnieje prawdopodobieństwo, że taka osoba się myli. I nie daj się zwieść relatywnym porównaniom. Milisekunda trwa 1000 razy dłużej niż mikrosekunda, ale nadal może być nieistotna dla ogólnej wydajności aplikacji.

[2] Java 13 oferuje przegląd działania wieloliniowych literałów znaków: *https://oreil.ly/CIlNB*.

[3] Fragment wiersza *Dziaberliada* w przekładzie Stanisława Barańczaka — *przyp. tłum.*

W takim przypadku drugim argumentem konstruktora obiektu `String` jest nazwa schematu kodowania znaków. Konstruktor klasy `String` używa go do konwersji surowych bajtów w określonym kodowaniu na wewnętrznie używane kodowanie wybrane przez środowisko uruchomieniowe. Jeśli nie określisz schematu kodowania znaków, użyty zostanie domyślny schemat kodowania z Twojego systemu[4].

Z kolei metoda `charAt()` klasy `String` umożliwia uzyskanie dostępu do znaków obiektu `String` w sposób tablicowy:

```
String s = "Newton";
for ( int i = 0; i < s.length(); i++ )
    System.out.println( s.charAt( i ) );
```

Ten kod powoduje wypisanie po kolei znaków danego łańcucha znaków.

Koncepcja obiektu `String` jako łańcucha znaków została również skodyfikowana przez klasę `String` implementującą interfejs `java.lang.CharSequence`, który określa metody `length()` i `charAt()` jako sposób pobierania podzbiorów znaków.

Tekstowe reprezentacje z różnych elementów

Obiekty i typy proste można w Javie przekształcać w domyślną reprezentację tekstową, którą jest obiekt String. W przypadku typów prostych, takich jak liczby, łańcuch znaków powinien być dość oczywisty, natomiast jeśli chodzi o typy obiektowe, pozostaje pod kontrolą samego obiektu. Tekstową reprezentację elementu można uzyskać za pomocą statycznej metody `String.valueOf()`. Różne przeciążone wersje tej metody przyjmują poszczególne typy proste:

```
String one = String.valueOf( 1 );           // Liczba całkowita, "1"
String two = String.valueOf( 2.384f );      // Liczba zmiennoprzecinkowa, "2.384"
String notTrue = String.valueOf( false );  // Wartość logiczna, "false"
```

Wszystkie obiekty w Javie mają metodę `toString()`, która jest dziedziczona po klasie `Object`. Dla wielu obiektów metoda ta zwraca przydatny wynik, wyświetlając ich zawartość. Metoda `toString()` obiektu `java.util.Date` zwraca na przykład datę sformatowaną jako łańcuch znaków. W przypadku obiektów, które nie zapewniają reprezentacji, wynikiem tekstowym jest po prostu unikatowy identyfikator, którego można użyć do debugowania. Wywołanie dla obiektu metody `String.valueOf()` powoduje wywołanie metody `toString()` danego obiektu i zwrócenie wyniku. Jedyna prawdziwa różnica w stosowaniu metody `String.valueOf()` polega na tym, że jeśli przekażesz jej referencję do obiektu o wartości `null`, to zamiast rzucania wyjątku `NullPointerException` zwróci łańcuch znaków *null*:

```
Date date = new Date();
// Równoważne na przykład z "Fri Jul 17 08:08:19 CEST 2020"
String d1 = String.valueOf( date );
String d2 = date.toString();

date = null;
d1 = String.valueOf( date );  // "null"
d2 = date.toString();  // NullPointerException!
```

[4] W przypadku większości platform domyślnym kodowaniem jest UTF-8. Więcej informacji na temat zestawów znaków, zestawów domyślnych i zestawów standardowych obsługiwanych przez Javę można znaleźć w oficjalnej dokumentacji Javadoc (*https://oreil.ly/UarRO*) dla klasy `java.nio.charset.Charset`.

Konkatenacja łańcuchów znaków wykorzystuje wewnętrznie metodę `valueOf()`, więc jeśli za pomocą operatora + „dodasz” do łańcucha znaków obiekt lub typ prosty, otrzymasz `String`:

```
String today = "Dziś jest :" + date;
```

Czasami spotkasz się z użyciem pustego łańcucha znaków i operatora + jako skrótu do uzyskania testowej wartości obiektu, np.:

```
String two = "" + 2.384f;
String today = "" + new Date();
```

Porównywanie łańcuchów znaków

Standardowa metoda `equals()` może porównać łańcuchy znaków pod kątem *równości* — czy zawierają dokładnie te same znaki w tej samej kolejności. Aby sprawdzić równoważność łańcuchów znaków bez uwzględniania wielkości liter, możesz użyć innej metody — `equalsIgnoreCase()`:

```
String one = "FOO";
String two = "foo";

one.equals( two );            // false
one.equalsIgnoreCase( two );  // true
```

Częstym błędem w przypadku początkujących programistów Javy jest porównywanie łańcuchów znaków za pomocą operatora `==`, podczas gdy należałoby skorzystać wtedy z metody `equals()`. Pamiętaj, że w Javie łańcuchy znaków są obiektami, a operator `==` przeprowadza testy pod kątem *tożsamości* (identyczności) obiektów — sprawdza, czy dwa testowane argumenty są tym samym obiektem. W Javie łatwo jest utworzyć dwa łańcuchy znaków, które zawierają te same znaki, ale nie są tym samym obiektem `String`, np.:

```
String foo1 = "foo";
String foo2 = String.valueOf( new char [] { 'f', 'o', 'o' } );

foo1 == foo2         // false!
foo1.equals( foo2 )  // true
```

Ten błąd jest szczególnie niebezpieczny, ponieważ taka metoda często sprawdza się w typowym przypadku, w którym porównujemy literały łańcuchów znaków (czyli łańcuchy znaków zadeklarowane za pomocą podwójnych cudzysłowów bezpośrednio w kodzie). Jest to spowodowane tym, że Java próbuje wydajnie zarządzać łańcuchami znaków przez ich łączenie. W czasie kompilacji Java znajduje wszystkie identyczne łańcuchy znaków w danej klasie i tworzy dla nich tylko jeden obiekt. Jest to bezpieczne, ponieważ łańcuchy znaków są niemutowalne i nie mogą być zmieniane. W ten sam sposób możesz sam łączyć łańcuchy znaków w trakcie wykonywania programu, używając metody `intern()` klasy `String`. Zastosowanie jej do łańcucha znaków powoduje zwrócenie referencji do równoważnego łańcucha znaków, unikatowej dla całej maszyny wirtualnej.

Metoda `compareTo()` porównuje wartość leksykalną dwóch obiektów `String`, określając ich wzajemną pozycję w sortowaniu alfabetycznym. Zwraca liczbę całkowitą, która może być mniejsza niż zero, równa lub większa od zera:

```
String abc = "abc";
String def = "def";
```

```
String num = "123";

if ( abc.compareTo( def ) < 0 )     // true
if ( abc.compareTo( abc ) == 0 )    // true
if ( abc.compareTo( num ) > 0 )     // true
```

Metoda `compareTo()` porównuje obiekty `String` wyłącznie na podstawie pozycji ich znaków w specyfikacji Unicode. Sprawdza się to w przypadku prostych tekstów, ale nie wszystkie języki są obsługiwane równie dobrze. Do bardziej wyrafinowanych porównań można używać klasy `Collator`, którą omówimy za chwilę.

Wyszukiwanie

Klasa `String` udostępnia kilka prostych metod umożliwiających wyszukiwanie określonych fragmentów łańcuchów znaków. Metody `startsWith()` i `endsWith()` porównują łańcuch znaków argumentu odpowiednio z początkiem i końcem obiektu `String`:

```
String url = "http://foo.bar.com/";
if ( url.startsWith("http:") )  // true
```

Metoda `indexOf()` wyszukuje pierwsze wystąpienie określonego znaku lub fragmentu łańcucha znaków i zwraca pozycję początkowego znaku lub wartość `-1`, jeśli podany fragment łańcucha znaków nie zostanie znaleziony:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 'p' );     // 15
int i = abcs.indexOf( "def" );   // 3
int I = abcs.indexOf( "Fang" );  // –1
```

Analogicznie metoda `lastIndexOf()` przeszukuje łańcuch znaków wstecz w celu znalezienia ostatniego wystąpienia określonego znaku lub fragmentu łańcucha znaków.

Metoda `contains()` obsługuje bardzo typowe zadanie sprawdzania, czy w docelowym łańcuchu znaków znajduje się określony fragment łańcucha znaków:

```
String log = "Wystąpiła awaria w sektorze 7.!";
if ( log.contains("awaria") ) pageSomeone();
// Równoważne z tym:
if ( log.indexOf("awaria") != -1 ) ...
```

Do przeprowadzania bardziej złożonego wyszukiwania można użyć interfejsu programistycznego wyrażeń regularnych (Regular Expression API), który pozwala wyszukiwać i analizować złożone wzorce. Wyrażenia regularne omówimy w dalszej części rozdziału.

Podsumowanie metod klasy String

Tabela 8.1 podsumowuje metody dostarczane przez klasę `String`. Umieściliśmy w niej kilka metod, których nie omówiliśmy w tym rozdziale, ale chcemy, żebyś miał świadomość dostępności również innych funkcjonalności tej klasy. Możesz śmiało wypróbować te metody w `jshell` lub przejrzeć dokumentację dostępną w internecie na stronie *https://oreil.ly/lbM1R*.

Tabela 8.1. Metody klasy String

Metoda	Funkcjonalność
`charAt()`	Pobiera konkretny znak z obiektu `String`
`compareTo()`	Porównuje dwa obiekty `String`
`concat()`	Konkatenuje dwa obiekty `String`
`contains()`	Sprawdza, czy dany obiekt `String` zawiera inny obiekt `String`
`copyValueOf()`	Zwraca obiekt `String` odpowiadający określonej tablicy znaków
`endsWith()`	Sprawdza, czy obiekt `String` kończy się określonym przyrostkiem
`equals()`	Porównuje dwa obiekty `String`
`equalsIgnoreCase()`	Porównuje dwa obiekty `String`, ignorując wielkość znaków
`getBytes()`	Kopiuje znaki z obiektu `String` do tablicy bajtów
`getChars()`	Kopiuje znaki z obiektu `String` do tablicy znaków
`hashCode()`	Zwraca kod skrótu dla danego obiektu `String`
`indexOf()`	Wyszukuje pierwsze wystąpienie określonego znaku lub fragmentu łańcucha znaków w danym obiekcie `String`
`intern()`	Pobiera unikatową instancję obiektu `String` z globalnej puli udostępnianych obiektów `String`
`isBlank()`	Zwraca wartość `true`, jeśli dany obiekt `String` ma zerową długość lub zawiera tylko znaki niedrukowalne
`isEmpty()`	Zwraca wartość `true`, jeśli dany obiekt `String` ma zerową długość
`lastIndexOf()`	Wyszukuje ostatnie wystąpienie określonego znaku lub fragmentu łańcucha znaków w danym obiekcie `String`
`length()`	Zwraca długość obiektu `String`
`lines()`	Zwraca strumień linii podzielony znakami końca linii
`matches()`	Określa, czy cały obiekt `String` odpowiada wzorcowi wyrażenia regularnego
`regionMatches()`	Sprawdza, czy dany region jednego obiektu `String` odpowiada określonemu regionowi drugiego obiektu `String`
`repeat()`	Zwraca konkatenację danego obiektu `String` powtórzonego określoną liczbę razy
`replace()`	Zastępuje wszystkie wystąpienia określonego znaku w obiekcie `String` innym znakiem
`replaceAll()`	Zastępuje wszystkie wystąpienia wzorca wyrażenia regularnego innym wzorcem
`replaceFirst()`	Zastępuje pierwsze wystąpienie wzorca wyrażenia regularnego innym wzorcem
`split()`	Dzieli obiekt `String` na tablicę obiektów `String`, wykorzystując jako separator wzorzec wyrażenia regularnego

Tabela 8.1. Metody klasy String (ciąg dalszy)

Metoda	Funkcjonalność
`startsWith()`	Sprawdza, czy dany obiekt `String` rozpoczyna się określonym przedrostkiem
`strip()`	Usuwa początkowe i końcowe znaki niedrukowalne określone za pomocą metody `Character.isWhitespace()` (*https://oreil.ly/NK1Nl*)
`stripLeading()`	Usuwa początkowe znaki niedrukowalne podobnie jak metoda `strip()`
`stripTrailing()`	Usuwa końcowe znaki niedrukowalne podobnie jak metoda `strip()`
`substring()`	Zwraca fragment łańcucha znaków z danego obiektu `String`
`toCharArray()`	Zwraca tablicę znaków z danego obiektu `String`
`toLowerCase()`	Konwertuje obiekt `String` na małe znaki
`toString()`	Zwraca wartość tekstową danego obiektu
`toUpperCase()`	Konwertuje obiekt `String` na wielkie znaki
`trim()`	Usuwa początkowe i końcowe znaki niedrukowalne zdefiniowane jako każdy znak z punktem kodowym mniejszym lub równym 32 (znak spacji)
`valueOf()`	Zwraca tekstową reprezentację danej wartości

Różne elementy z tekstowych reprezentacji

Parsowanie i formatowanie tekstu to obszerny, otwarty temat. W tym rozdziale do tej pory przyglądaliśmy się jedynie prostym operacjom na łańcuchach znaków — tworzeniu łańcuchów znaków, przeszukiwaniu ich oraz przekształcaniu prostych wartości w łańcuchy znaków. Teraz chcielibyśmy przejść do bardziej ustrukturyzowanych form tekstu. Java oferuje bogaty zestaw interfejsów API do parsowania i wypisywania sformatowanych łańcuchów znaków, w tym liczb, dat, godzin i wartości walutowych. Większość z tych tematów omówimy w tym rozdziale, a formatowaniem dat i godzin zajmiemy się konkretnie w podrozdziale „Daty i godziny".

Zaczniemy od parsowania, czyli odczytywania prostych liczb i wartości jako łańcuchów znaków oraz dzielenia długich łańcuchów znaków na tokeny. Następnie przyjrzymy się wyrażeniom regularnym — najpotężniejszemu narzędziu do parsowania tekstu, jakie oferuje Java. Wyrażenia regularne pozwalają definiować własne wzorce o dowolnej złożoności, przeszukiwać je oraz parsować je na podstawie tekstu.

Parsowanie liczb prostych

W Javie liczby, znaki i wartości logiczne nie są obiektami, tylko typami prostymi. Jednak dla każdego typu prostego Java definiuje również *klasę opakowującą*. Pakiet `java.lang` zawiera w szczególności następujące klasy: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` i `Boolean`. Omawialiśmy je już w rozdziale 5. w podrozdziale „Metody" w punkcie „Klasy opakowujące dla typów prostych", ale wracamy do nich teraz, ponieważ klasy te posiadają statyczne metody narzędziowe służące do

parsowania odpowiadających im typów z łańcuchów znaków. Każda z tych klas opakowujących ma statyczną metodę „parsowania", która odczytuje obiekt `String` i zwraca odpowiedni typ prosty, np.:

```
byte b = Byte.parseByte("16");
int n = Integer.parseInt( "42" );
long l = Long.parseLong( "99999999999" );
float f = Float.parseFloat( "4.2" );
double d = Double.parseDouble( "99.99999999" );
boolean b = Boolean.parseBoolean("true");
```

Jako alternatywę `java.util.Scanner` zapewnia pojedynczy interfejs API, który służy nie tylko do parsowania poszczególnych typów prostych z łańcuchów znaków, ale także do odczytywania tych typów ze strumienia tokenów. Poniższy przykład pokazuje, jak używać tego API zamiast klas opakowujących:

```
byte b = new Scanner("16").nextByte();
int n = new Scanner("42").nextInt();
long l = new Scanner("99999999999").nextLong();
float f = new Scanner("4.2").nextFloat();
double d = new Scanner("99.99999999").nextDouble();
boolean b = new Scanner("true").nextBoolean();
```

Tokenizacja tekstu

Typowe zadanie programistyczne polega na parsowaniu łańcuchów znaków do postaci słów, zwanych też tokenami, oddzielonych pewnym zestawem separatorów, takich jak spacje lub przecinki. Pierwszy przykład zawiera słowa oddzielone pojedynczymi spacjami. Drugi prezentuje bardziej realistyczny problem dotyczący pól rozdzielanych przecinkami.

```
Nadszedł czas, aby wszyscy dobrzy ludzie...

Numer kontrolny, Opis,                  Liczba
4231,            Programowanie w Javie, 1000.00
```

Do wykonywania takich zadań Java ma kilka (niestety nakładających się) interfejsów API. Najbardziej wszechstronne i przydatne API to `split()` klasy `String` oraz `Scanner`. Oba wykorzystują wyrażenia regularne, aby umożliwić dzielenie łańcuchów znaków według dowolnych wzorców. Nie omawialiśmy jeszcze wyrażeń regularnych, ale żebyś zobaczył je w działaniu, pokażemy Ci po prostu kilka gotowych sztuczek, a szczegółowo wyjaśnimy te kwestie w dalszej części rozdziału. Wspomnimy również o starszym narzędziu, `java.util.StringTokenizer`, które do dzielenia obiektów `String` używa prostych zestawów znaków. Nie jest to tak wielofunkcyjne narzędzie, ale nie wymaga zrozumienia wyrażeń regularnych.

Metoda `split()` klasy `String` przyjmuje wyrażenie regularne opisujące separator i używa go do podzielenia łańcucha na tablicę obiektów `String`:

```
String text = "Nadszedł czas, aby wszyscy dobrzy ludzie";
String [] words = text.split("\\s");
// words = "Nadszedł", "czas,", "aby", "wszyscy", …

String text = "4231,            Programowanie w Javie, 1000.00";
String [] fields = text.split("\\s*,\\s*");
// fields = "4231", "Programowanie w Javie", "1000.00"
```

W pierwszym przykładzie użyliśmy wyrażenia regularnego \\s, które dopasowuje pojedynczy znak niedrukowalny (spację, tabulator lub powrót karetki). Metoda `split()` zwróciła tablicę sześciu łańcuchów znaków. W drugim przykładzie użyliśmy bardziej skomplikowanego wyrażenia regularnego \\s*,\\s*, które dopasowuje przecinek otoczony dowolną liczbą ciągłych spacji (być może zerową). Zredukowało to nasz tekst do trzech przyjemnych, uporządkowanych pól.

W przypadku nowego API `Scanner` możemy pójść o krok dalej i parsować liczby z naszego drugiego przykładu podczas ich wyodrębniania:

```
String text = "4231,            Programowanie w Javie, 1000.00";
Scanner scanner = new Scanner( text ).useDelimiter("\\s*,\\s*");
int checkNumber = scanner.nextInt(); // 4231
String description = scanner.next(); // "Programowanie w Javie"
float amount = scanner.nextFloat();  // 1000.00
```

W tym kodzie poinstruowaliśmy `Scanner`, aby jako separatora użył naszego wyrażenia regularnego, a następnie wywołaliśmy go wielokrotnie w celu sparsowania każdego pola jako odpowiadającego mu typu. Interfejs `Scanner` jest wygodny, ponieważ może przeprowadzać operacje odczytu nie tylko z obiektów `String`, ale również bezpośrednio ze źródeł strumieniowych (więcej informacji na ten temat znajdziesz w rozdziale 11.), takich jak `InputStream` (strumień wejściowy), `File` (plik) oraz `Channel` (kanał):

```
Scanner fileScanner = new Scanner( new File("arkuszkalkulacyjny.csv") );
fileScanner.useDelimiter( "\\s*,\\s*" );
//...
```

Za pomocą interfejsu `Scanner` można ponadto „zerkać w przód” za pomocą metod `hasNext`, aby sprawdzić, czy będzie kolejny element:

```
while( scanner.hasNextInt() ) {
    int n = scanner.nextInt();
    ...
}
```

Klasa StringTokenizer

Chociaż wspomniana już klasa `StringTokenizer` odchodzi powoli w przeszłość, dobrze jest ją poznać, ponieważ powstała wraz z językiem Java i jest używana w wielu kodach. `StringTokenizer` pozwala określać separator jako zestaw znaków i dopasowywać dowolną liczbę lub kombinację tych znaków rozdzielających tokeny. Poniższy fragment kodu odczytuje słowa z naszego pierwszego przykładu:

```
String text = "Nadszedł czas, aby wszyscy dobrzy ludzie...";
StringTokenizer st = new StringTokenizer( text );

while ( st.hasMoreTokens() ) {
    String word = st.nextToken();
    ...
}
```

Metody `hasMoreTokens()` i `nextToken()` wywołujemy w celu wykonywania pętli przez słowa danego tekstu. Domyślnie klasa StringTokenizer używa jako separatorów standardowych znaków niedrukowalnych: powrotu karetki, nowej linii i tabulatora. W konstruktorze `StringTokenizer` można określić

także własny zestaw znaków separatora. Między tokenami pomijana będzie każda ciągła kombinacja określonych znaków pojawiających się w docelowym łańcuchu znaków:

```
String text = "4231,            Programowanie w Javie, 1000.00";
StringTokenizer st = new StringTokenizer( text, "," );

while ( st.hasMoreTokens() ) {
    String word = st.nextToken();
    // word = "4231", "        Programowanie w Javie", "1000.00"
}
```

Nie jest to tak przejrzysty kod, jak nasz przykład z wyrażeniami regularnymi. W tym przypadku jako separatora użyliśmy przecinka, więc w polu opisu otrzymujemy na początku dodatkowe znaki niedrukowalne. Gdybyśmy do naszego łańcucha znaków separatora dodali spację, `StringTokenizer` podzieliłby nasz opis na trzy słowa: *Programowanie*, *w* i *Javie*, czego byśmy nie chcieli. Rozwiązaniem byłoby tutaj użycie metody `trim()`, aby usunąć początkowe i końcowe spacje w każdym elemencie.

Wyrażenia regularne

Teraz nadszedł czas na krótką przerwę w naszej podróży przez Javę i wkroczenie do krainy **wyrażeń regularnych** (ang. *regular expressions*). Wyrażenie regularne, czyli w skrócie od angielskiej nazwy — *regex*, opisuje wzorzec tekstowy. Wyrażenia regularne są używane w wielu narzędziach — m.in. w pakiecie `java.util.regex`, edytorach tekstu i wielu językach skryptowych — w celu zapewnienia zaawansowanego wyszukiwania tekstu i wszechstronnych możliwości manipulacji łańcuchami znaków.

Jeżeli jesteś już zaznajomiony z wyrażeniami regularnymi i sposobem ich używania w innych językach, możesz pominąć pierwszy punkt tego podrozdziału. Powinieneś zapoznać się jednak z kolejnym punktem „Interfejs API java.util.regex", w którym omówimy niezbędne klasy Javy. Jeśli jednak doszedłeś do tego etapu swojej podróży po Javie z czystym kontem w tej kwestii i zastanawiasz się, czym właściwie są wyrażenia regularne, otwórz ulubiony napój i się przygotuj. Za chwilę zapoznasz się z najpotężniejszym narzędziem w arsenale manipulacji tekstem oraz tym, co jest w rzeczywistości niewielkim językiem w ramach głównego języka — a wszystko to na kilku stronach.

Notacja wyrażeń regularnych

Wyrażenie regularne opisuje pewien wzorzec w tekście. Przez wzorzec rozumiemy prawie każdy możliwy do wyobrażenia sobie element, jaki można zidentyfikować w tekście na podstawie samych literalnych znaków, bez faktycznego rozumienia ich znaczenia. Do takich elementów należą m.in. słowa, grupy słów, wiersze i akapity, interpunkcja i wielkość liter oraz, w ujęciu bardziej ogólnym, łańcuchy znaków i liczby o określonej strukturze, takie jak numery telefonów, adresy e-mailowe i cytowane frazy. Za pomocą wyrażeń regularnych można np. przeszukać słownik pod kątem wszystkich słów zawierających literę „q", ale bez przylegającej do niej jej koleżanki, litery „u", albo słów zaczynających się od tej samej litery, na którą się kończą. Po zbudowaniu wzorca można użyć prostych narzędzi, aby wyszukać go w tekście lub ustalić, czy można dopasować do niego w tekście określony łańcuch znaków. Wyrażenie regularne może również zostać użyte chociażby

do rozdzielenia dopasowanego tekstu na określone części, które można następnie wykorzystać przykładowo jako elementy do zastępowania tekstu.

Pisz uniwersalnie, czyli „write once, run anywhere — WORA"

Zanim przejdziemy dalej, powinniśmy napisać kilka słów o ogólnej składni wyrażeń regularnych. Na początku tego podrozdziału mimochodem wspomnieliśmy, że wyrażenia regularne można traktować jako samodzielny niewielki język. I rzeczywiście tak jest, ponieważ wyrażenia regularne stanowią prostą formę języka programowania. Jeśli zastanowisz się przez chwilę nad przytoczonymi wcześniej przykładami, prawdopodobnie sam uznasz, że do opisania nawet prostych wzorców, takich jak adresy e-mailowe, które mogą przyjmować różne formy, potrzebne będzie coś w rodzaju języka.

Podręcznik do informatyki z pewnością sklasyfikowałby wyrażenia regularne na samym dole hierarchii języków komputerowych, zarówno pod względem tego, co potrafią opisywać, jak i tego, co można za ich pomocą zrobić. Tak czy inaczej, wyrażenia regularne potrafią być dość wyrafinowane. Podobnie jak w przypadku większości języków programowania, elementy wyrażeń regularnych są proste, ale można z nich budować wzorce o dowolnej złożoności. I właśnie tutaj zaczyna robić się mniej przyjemnie.

Ponieważ wyrażenia regularne operują na łańcuchach znaków, wygodnie byłoby mieć bardzo zwartą notację, którą można by łatwo wcisnąć między inne znaki. Jednak zwięzła notacja może być tajemnicza i trudna do rozszyfrowania, a doświadczenie pokazuje, że o wiele łatwiej jest pisać złożone instrukcje niż później je czytać. To jest właśnie przekleństwo wyrażeń regularnych. Może się okazać, że pewną wieczorową porą, zainspirowany dużą ilością kofeiny, będziesz w stanie napisać pojedynczy wspaniały wzorzec, który uprości resztę programu do jednej linii. Kiedy jednak powrócisz do czytania tej linii następnego dnia, możesz stwierdzić, że są to dla Ciebie jakieś egipskie hieroglify. Ogólnie rzecz biorąc, im prościej, tym lepiej, a jeśli możesz rozłożyć problem na mniejsze elementy i zrobić to bardziej przejrzyście w kilku krokach, być może właśnie tak powinieneś postąpić.

Znaki ucieczki

Skoro zostałeś już odpowiednio ostrzeżony, musimy jeszcze raz powalić Cię na ziemię, zanim pomożemy Ci stanąć z powrotem na nogi. Notacja wyrażeń regularnych może stać się nie tylko trochę pogmatwana, ale bywa również w pewien sposób niejednoznaczna w przypadku zwykłych łańcuchów znaków Javy. Ważną częścią notacji jest tzw. znak ucieczki (ang. *escape character*) w postaci lewego ukośnika, który służy do zmiany interpretacji znaku lub sekwencji znaków po nim następujących. Znak ucieczki, po którym wpiszemy np. literę „d", czyli `\d`, to skrót dopasowujący dowolny znak jednocyfrowy (z zakresu 0 – 9). Nie możesz jednak wpisać po prostu `\d` do łańcucha znaków Javy, ponieważ (jak pewnie pamiętasz) Java używa lewego ukośnika dla własnych znaków specjalnych oraz określania sekwencji znaków Unicode (`\uxxxx`). Na szczęście Java oferuje nam zamiennik: znak ucieczki dla znaku ucieczki, czyli w sumie dwa lewe ukośniki (`\\`), co w efekcie oznacza literalny lewy ukośnik. Zasada jest taka, że jeśli chcesz, aby w wyrażeniu regularnym pojawił się lewy ukośnik, musisz zastosować do niego jeszcze jeden:

```
"\\d"   // Łańcuch znaków Javy, który w efekcie daje pojedynczy znak ukośnika i literę d (/d)
```

Ale żeby było jeszcze zabawniej, z uwagi na to, że notacja wyrażeń regularnych sama używa lewego ukośnika do oznaczania znaków specjalnych, również musi zapewniać analogiczne „okienko ucieczki", czyli umożliwić podwojenie lewego ukośnika, gdy chcemy uzyskać literalny lewy ukośnik. Jeśli chcesz więc określić wyrażenie regularne zawierające pojedynczy literalny lewy ukośnik, wygląda to tak:

```
"\\\\" // Łańcuch znaków Javy daje dwa lewe ukośniki, a wyrażenie regularne jeden
```

Większość „magicznych" znaków operatorów, o których czytasz w tym podpunkcie rozdziału, działa na poprzedzające je znaki, dlatego dla nich też musisz zastosować znak ucieczki, jeśli potrzebujesz ich literalnego znaczenia. Obejmuje to m.in. następujące znaki: `.`, `*`, `+`, `{}` oraz `()`.

Jeżeli chcesz utworzyć jakąś część wyrażenia, która będzie zawierała wiele literalnych znaków, pomocne mogą być specjalne separatory `\Q` i `\E`. Każdy tekst umieszczony między separatorami `\Q` i `\E` jest automatycznie traktowany w taki sposób, jakby do całości zastosowany został znak ucieczki. (Pamiętaj, że w przypadku łańcucha znaków Javy dla pojedynczego lewego ukośnika nadal potrzebny jest drugi lewy ukośnik, ale czterokrotny już nie). Istnieje również statyczna metoda o nazwie `Pattern.quote()`, która robi to samo, zwracając zacytowaną (literalną) wersję dowolnego podanego jej łańcucha znaków.

Oprócz tego naszą jedyną sugestią, która może pomóc Ci zachować zdrowie psychiczne podczas pracy z tymi przykładami, jest utrzymywanie dwóch kopii — linii komentarza pokazującego nagie wyrażenie regularne oraz prawdziwego łańcucha znaków Javy, w którym należy podwajać wszystkie lewe ukośniki. I nie zapominaj o `jshell`! Może to być bardzo wszechstronna „piaskownica" do testowania i poprawiania wzorców.

Znaki i klasy znaków

Przejdźmy teraz do faktycznej składni wyrażeń regularnych. Najprostszą formą wyrażenia regularnego jest zwykły, literalny tekst, który nie ma żadnego specjalnego znaczenia i jest porównywany bezpośrednio (znak do znaku) do danych wejściowych. Może to być pojedynczy znak lub większa liczba znaków. W poniższym łańcuchu znaków wzorzec `"n"` może np. pasować do znaku `n` w słowach `Cena` i `wynosi`:

```
"Cena róży wynosi 1.99 zł."
```

Wzorzec `"róży"` może pasować tylko do literalnego słowa `róży`. Nie jest to jednak zbytnio interesujące. Podnieśmy poprzeczkę, wprowadzając pewne znaki specjalne oraz pojęcie „klas" znaków.

Dowolny znak: .

Znak specjalny w postaci kropki (`.`) dopasowuje dowolny pojedynczy znak. Wzorzec `".ena"` pasuje do słów `Cena`, `wena`, `_ena` (łańcuch znaków `ena` po spacji) lub do dowolnego innego znaku, po którym następuje sekwencja `ena`. Dwie kropki dopasowują dwa dowolne znaki (`ocena`, `scena` itp.) i tak dalej. Operator kropki nie rozróżnia wartości znaków — zwykle zatrzymuje się tylko dla znaku końca linii (i opcjonalnie można poinstruować go, żeby tego nie robił, ale tym zajmiemy się później). Możemy uznać, że operator `.` reprezentuje grupę lub, inaczej mówiąc, klasę wszystkich znaków. Wyrażenia regularne definiują również bardziej interesujące klasy znaków.

Znak niedrukowalny lub znak inny niż niedrukowalny: `\s`, `\S`

Znak specjalny `\s` dopasowuje znak literalnej spacji lub jeden z następujących znaków: `\t` (tabulator), `\r` (powrót karetki), `\n` (nowa linia), `\f` (podział strony) oraz backspace. Odpowiadający mu znak specjalny `\S` robi rzecz przeciwną, dopasowując dowolny znak z wyjątkiem znaków niedrukowalnych.

Znak cyfry lub znak inny niż cyfry: `\d`, `\D`

Znak `\d` dopasowuje dowolną cyfrę (0 – 9). Znak `\D` robi rzecz przeciwną, dopasowując wszystkie znaki oprócz cyfr.

Znak słowa lub znak inny niż słowa: `\w`, `\W`

Znak `\w` dopasowuje dowolny znak „słowa”, w tym wielkie i małe litery, czyli A – Z i a – z, cyfry 0 – 9 oraz znak podkreślenia (_). Znak `\W` dopasowuje wszystko oprócz tych znaków.

Niestandardowe klasy znaków

Własne klasy znaków można definiować za pomocą notacji `[...]`. Poniższa klasa dopasowuje na przykład dowolny ze znaków a, b, c, x, y lub z:

```
[abcxyz]
```

Specjalnej notacji zakresu `x-y` można używać jako skrótu dla znaków alfanumerycznych. Poniższy przykład definiuje klasę znaków zawierającą wszystkie wielkie i małe litery:

```
[A-Za-z]
```

Umieszczenie znaku karetki (^) jako pierwszego znaku po otwierającym nawiasie kwadratowym odwraca znaczenie klasy znaków. Klasa z poniższego przykładu dopasowuje dowolny znak oprócz wielkich liter z zakresu A – F:

```
[^A-F]   // G, H, I, …, a, b, c, … itd.
```

Zagnieżdżenie klas znaków powoduje po prostu ich dodanie:

```
[A-F[G-Z]\w]   // A – Z oraz znaki niedrukowalne
```

Notacja `&&` koniunkcji logicznej (AND) może być używana w celu dopasowywania znaków z części wspólnej zdefiniowanych klas znaków:

```
[a-p&&[l-z]]   // l, m, n, o, p
[A-Z&&[^P]]    // Od A do Z oprócz P
```

Znaczniki pozycji

Wzorzec `"[Oo] różo"` (łącznie z wielką lub małą literą *o*) znajduje w poniższej frazie trzy dopasowania:

```
"O różo o różo o różo"
```

Znaki pozycji pozwalają wyznaczyć względną lokalizację dopasowania. Najważniejsze z nich to ^ oraz $, które dopasowują odpowiednio do początku i końca linii:

```
^[Oo] różo   // Dopasowuje „O różo” i „o różo” na początku linii
[Oo] różo$   // Dopasowuje „O różo” i „o różo” na końcu linii
```

Mówiąc ściślej, znaki ^ i $ dopasowują odpowiednio do początku i końca danych wejściowych, którymi często jest pojedyncza linia. Jeśli pracujesz z wieloma liniami tekstu i chcesz dopasowywać początki i końce linii w jednym długim łańcuchu znaków, możesz za pomocą odpowiedniej flagi włączyć tryb „wieloliniowy” z flagą, jak opisano w dalszej części tego punktu rozdziału.

Znaczniki pozycji `\b` i `\B` dopasowują odpowiednio granicę słowa lub granicę znaków innych niż słowo. Poniższy wzorzec dopasowuje na przykład słowa `róża` i `różaniec`, ale `stróża` już nie:

```
\bróża
```

Iteracje (wielokrotność wystąpień)

Proste dopasowywanie ustalonych wzorców znaków nie zaprowadziłoby nas daleko. Dlatego przyjrzymy się teraz operatorom, które zliczają wystąpienia określonego znaku (lub w bardziej ogólnym przypadku — wzorca, jak zobaczysz w następnym punkcie tego podrozdziału).

Operator Any (dowolna liczba iteracji): znak gwiazdki (*)

Umieszczenie gwiazdki (*) po znaku lub klasie znaków oznacza dopuszczenie dowolnej liczby wystąpień danego typu znaków — innymi słowy może to być zero lub więcej powtórzeń. Poniższy wzorzec dopasowuje na przykład cyfrę z dowolną liczbą zer na początku (być może bez żadnego):

```
0*\d   // Dopasowuje cyfrę z dowolną liczbą zer na początku
```

Operator Some (co najmniej jedna iteracja): znak plusa (+)

Znak plusa (+) oznacza co najmniej jedną iterację i jest równoważny wyrażeniu `XX*` (wzorcowi, po którym następuje operator gwiazdki). Poniższy wzorzec dopasowuje na przykład liczbę z przynajmniej jedną cyfrą i opcjonalnymi zerami na początku:

```
0*\d+   // Dopasowuje liczbę z co najmniej jedną liczbą i opcjonalnymi zerami na początku
```

Dopasowywanie zer na początku wyrażenia może wydawać się zbędne, ponieważ zero jest cyfrą, więc i tak jest dopasowywane przez fragment `\d+` tego wyrażenia. Później pokażemy Ci jednak, jak możesz porozdzielać łańcuch znaków za pomocą wyrażenia regularnego i uzyskać tylko te fragmenty, których potrzebujesz. W tym przypadku mógłbyś chcieć usunąć początkowe zera i zachować tylko cyfry.

Operator Optional (zero iteracji lub jedna): znak zapytania (?)

Operator znaku zapytania (?) dopuszcza dokładnie zero iteracji lub jedną iterację. Poniższy wzorzec dopasowuje na przykład datę ważności karty kredytowej, która może ewentualnie zawierać pośrodku ukośnik:

```
\d\d/?\d\d   // Dopasowuje cztery cyfry z opcjonalnym ukośnikiem pośrodku
```

Operator Range (przedział obustronnie domknięty od x do y iteracji): `{x,y}`

Operator zakresu `{x,y}` jest najbardziej powszechnym operatorem iteracji. Określa dokładny zakres do dopasowania, który przyjmuje dwa rozdzielone przecinkiem argumenty: dolną granicę

i górną granicę. Poniższe wyrażenie regularne dopasowuje każde słowo, które ma od pięciu do siedmiu znaków włącznie:

```
\b\w{5,7}\b   // Dopasowuje każde słowo zawierające co najmniej 5 i najwyżej 7 znaków
```

Co najmniej x iteracji (y jest nieskończone): `{x,}`

Jeśli pominiemy górną granicę przedziału, pozostawiając sam przecinek przed zamykającym nawiasem klamrowym, górna granica stanie się nieskończona. W ten sposób można określić minimum wystąpień bez żadnego maksimum.

Alternatywa

Operator w postaci pionowej kreski (`|`) oznacza logiczną operację alternatywy (OR), zwaną także alternacją lub wyborem. Operator `|` nie działa na poszczególnych znakach. Jest stosowany do wszystkiego, co znajduje się po obu jego stronach. Dzieli wyrażenie na dwie części, chyba że jest użyty w grupowaniu w nawiasach. Nieco naiwne podejście do parsowania dat może wyglądać np. tak:

```
\w+, \w+ \d+ \d+|\d\d/\d\d/\d\d   // Wzorzec 1. lub wzorzec 2.
```

W tym wyrażeniu lewy wzorzec pasuje np. do formatu daty *Poniedziałek, 20 lipca 2020*, a prawy do formatu *07/20/2020*.

Poniższego wyrażenia regularnego można użyć do dopasowywania adresów e-mailowych z jedną z trzech domen (*net*, *edu* i *gov*):

```
\w+@[\w.]*\.(net|edu|gov)   // Adres e-mailowy kończący się na .net, .edu lub .gov
```

Opcje specjalne

Istnieje kilka specjalnych opcji, które wpływają na sposób przeprowadzania dopasowywania przez silnik wyrażeń regularnych. Te opcje można zastosować na dwa sposoby:

- Flagę lub kilka flag można przekazać na etapie wywoływania metody `Pattern.compile()` (omówimy to w następnym punkcie rozdziału).
- Do wyrażenia regularnego można dołączyć specjalny blok kodu.

Tutaj pokażemy to drugie podejście. Polega ono na umieszczeniu flagi lub flag w specjalnym bloku (`?x`), gdzie *x* jest flagą opcji, którą chcemy włączyć. Zasadniczo robi się to na początku wyrażenia regularnego. Flagi można także wyłączać poprzez dodanie w bloku znaku minusa (`?-x`), co umożliwia stosowanie flag do wybierania fragmentów wzorca.

Dostępne są następujące flagi:

Ignorowanie wielkości liter (flaga Case-insensitive): `(?i)`

Ta flaga instruuje silnik wyrażeń regularnych, aby podczas dopasowywania ignorował wielkość liter, np.:

```
(?i)yahoo   // Dopasowuje Yahoo, yahoo, yahOO itp.
```

Dopasowywanie wszystkiego (flaga Dot all): `(?s)`

Flaga `(?s)` włącza tryb „kropkuj wszystko", pozwalając znakowi kropki dopasować wszystko, w tym również znaki końca linii. Jest to przydatne, jeśli dopasowuje się wzorce obejmujące wiele linii. Litera s pochodzi od angielskiego określenia *single-line mode*, oznaczającego „tryb jednoliniowy". Jest to nieco myląca nazwa zaczerpnięta z języka Perl.

Tryb wieloliniowy (flaga Multiline): `(?m)`

Domyślnie znaki specjalne ^ i $ tak naprawdę nie dopasowują tych początków i końców linii, które są definiowane przez kombinacje znaków powrotu karetki lub nowej linii. Zamiast tego dopasowują początek lub koniec całego tekstu wejściowego. W wielu przypadkach „jedna linia" jest równoznaczna z całymi danymi wejściowymi. Jeżeli mamy do przetworzenia duży blok tekstu, często z jakichś innych powodów dzielimy go na osobne linie — wtedy sprawdzenie dowolnego wiersza za pomocą wyrażenia regularnego jest proste, a operatory ^ i $ zachowują się zgodnie z oczekiwaniami. Jeśli jednak chcemy użyć wyrażenia regularnego dla całego wejściowego łańcucha znaków zawierającego wiele linii (oddzielonych tymi kombinacjami znaków powrotu karetki lub nowej linii), możemy włączyć tryb wieloliniowy za pomocą flagi `(?m)`. Ta flaga powoduje, że znaki specjalne ^ i $ dopasowują początki i końce poszczególnych linii w bloku tekstu, a także początek i koniec całego bloku. Oznacza to w szczególności miejsce przed pierwszym znakiem, miejsce po ostatnim znaku oraz miejsca tuż przed terminatorami linii wewnątrz łańcucha znaków i zaraz po nich.

Linie uniksowe(flaga Unix lines): `(?d)`

Flaga `(?d)` ogranicza definicję terminatora linii dla znaków specjalnych ^, $ i `.` tylko do nowej linii w stylu uniksowym (`\n`). Domyślnie dozwolona jest również kombinacja znaków powrotu karetki i nowej linii (`\r\n`).

Interfejs API java.util.regex

Omówiliśmy już teorię konstruowania wyrażeń regularnych, więc najtrudniejsze za nami. Pozostało tylko zbadanie interfejsu API Javy służącego do stosowania tych wyrażeń.

Klasa Pattern

Jak już wspomnieliśmy, wzorce wyrażeń regularnych, które piszemy jako łańcuchy znaków, są w rzeczywistości niewielkimi programami opisującymi sposób dopasowywania tekstu. Podczas wykonywania kodu pakiet wyrażeń regularnych Javy kompiluje te małe programy do postaci, która umożliwia ich uruchomienie dla wskazanego tekstu docelowego. Kilka prostych metod pomocniczych przyjmuje bezpośrednio łańcuchy znaków, które mają być używane jako wzorce. Natomiast w ujęciu bardziej ogólnym Java pozwala bezpośrednio kompilować wzorce i hermetyzować je w instancji obiektu klasy `Pattern` (wzorzec). Jest to najbardziej wydajny sposób obsługi wzorców, które są używane więcej niż raz, ponieważ eliminuje niepotrzebną ponowną kompilację łańcuchów znaków. Aby skompilować wzór, używamy statycznej metody `Pattern.compile()`:

```
Pattern urlPattern = Pattern.compile("\\w+://[\\w/]*");
```

Gdy mamy już obiekt `Pattern`, możemy poinstruować go, żeby utworzył obiekt klasy `Matcher` (klasy dopasowującej), który wiąże wzorzec z docelowym łańcuchem znaków:

```
Matcher matcher = urlPattern.matcher( myText );
```

`Matcher` przeprowadza dopasowywania. Zajmiemy się tym za chwilę, ale zanim to zrobimy, wspomnimy tylko o jednej metodzie pomocniczej klasy `Pattern`. Jest to statyczna metoda `Pattern.matches()`, która przyjmuje po prostu dwa łańcuchy znaków — wyrażenie regularne oraz docelowy łańcuch znaków — i przeprowadza na tej podstawie dopasowanie. Jest to bardzo wygodne, jeśli trzeba w aplikacji wykonać jednorazowy szybki test, np.:

```
Boolean match = Pattern.matches( "\\d+\\.\\d+f?", myText );
```

Ta linia kodu testuje, czy łańcuch znaków `myText` zawiera liczbę zmiennoprzecinkową w stylu Javy, taką jak chociażby `42.0f`. Zwróć uwagę, że aby dany łańcuch znaków można było uznać za dopasowany, musi on pasować całkowicie. Jeżeli chcesz sprawdzić, czy jakiś dłuższy łańcuch znaków zawiera pewien niewielki wzorzec, ale nie obchodzi Cię reszta tego łańcucha znaków, musisz użyć klasy `Matcher`, jak opisano w następnym podpunkcie.

Wypróbujmy kolejny (uproszczony) wzorzec, który moglibyśmy zastosować w naszej grze, kiedy zaczniemy umożliwiać wielu graczom konkurowanie ze sobą. Sporo systemów logowania jako identyfikatora użytkownika używa adresu e-mailowego. Takie systemy oczywiście nie są idealne, ale dla naszych potrzeb adres e-mailowy sprawdzi się doskonale. Chcielibyśmy poprosić użytkownika, aby wprowadził swój adres e-mailowy, ale zanim go użyjemy, musimy się upewnić, że wygląda poprawnie. Wyrażenie regularne może być szybkim sposobem na przeprowadzenie takiej walidacji[5].

Podobnie jak w przypadku pisania algorytmów w celu rozwiązania problemów programistycznych, zaprojektowanie wyrażenia regularnego wymaga rozłożenia problemu wzorca dopasowującego na mniejsze elementy. Jeśli chodzi o adresy e-mailowe, od razu nasuwa się kilka wzorców. Najbardziej oczywistym jest znak `@` pośrodku każdego adresu. Naiwny (ale lepsze to niż nic!) wzorzec oparty na tym fakcie można zbudować w następujący sposób:

```
String sample = "moje.imię@jakaś.domena";
Boolean validEmail = Pattern.matches(".*@.*", sample);
```

Ten wzorzec jest jednak zbyt permisywny. Z pewnością rozpozna prawidłowe adresy e-mailowe, ale rozpozna również wiele niepoprawnych, takich jak `"zły.adres@"` czy `"@również.zły"`, lub nawet `"@@"`. (Przetestuj to w `jshell` i może zmajstruj jeszcze kilka własnych złych przykładów!). Jak możemy ulepszyć te wzorce dopasowywania? Jedną z szybkich poprawek jest użycie modyfikatora + zamiast *. Taki zrefaktoryzowany wzorzec będzie wymagać teraz co najmniej po jednym znaku po każdej stronie znaku `@`. Wiemy jednak, że adresy e-mailowe mają jeszcze kilka innych charakterystycznych cech. Lewa „połowa” adresu (ta część z nazwą) nie może na przykład zawierać znaku `@`. To samo dotyczy również części domenowej adresu. Do tej następnej aktualizacji możemy użyć niestandardowej klasy znaków:

[5] Walidacja adresów e-mailowych okazuje się znacznie trudniejsza niż to, co możemy w tej chwili wykonać i przedstawić na łamach tego rozdziału. Wyrażenia regularne mogą obsłużyć większość prawidłowych adresów, ale jeśli przeprowadzasz walidację dla komercyjnej lub profesjonalnej aplikacji, powinieneś poszukać raczej jakiejś zewnętrznej biblioteki, na przykład takiej jak ta oferowana przez projekt Apache Commons (*https://oreil.ly/JEjEk*).

```
String sample = "moje.imię@jakaś.domena";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+", sample);
```

Ten wzorzec jest lepszy, ale nadal dopuszcza kilka nieprawidłowych adresów, takich jak `"wciąż@zły"`, ponieważ domeny zawierają co najmniej jedną nazwę, po której następuje kropka (`.`), a następnie domena najwyższego poziomu (ang. *Top Level Domain* — TLD), co wygląda na przykład tak: *helion.pl*. Może więc taki wzorzec:

```
String sample = "moje.imię@jakaś.domena";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.(com|org)", sample);
```

To rozwiązuje nasz problem z adresem, takim jak `"wciąż@zły"`, ale poszliśmy trochę za daleko w drugą stronę. Istnieje bardzo wiele domen najwyższego poziomu — zbyt wiele, aby można było sensownie wypisać na liście, nawet jeśli zignorujemy problem z utrzymywaniem tej listy w miarę dodawania nowych domen najwyższego poziomu[6]. Cofnijmy się więc trochę. Zachowamy kropkę w części domenowej, ale usuniemy konkretną domenę najwyższego poziomu i po prostu zaakceptujemy zwykły ciąg liter:

```
String sample = "moje.imię@jakaś.domena";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.[a-z]+", sample);
```

O wiele lepiej. Możemy dodać ostatnią poprawkę, aby nie przejmować się wielkością liter, ponieważ w adresach e-mailowych nie są one uwzględniane. Wystarczy dołączyć odpowiednią flagę:

```
String sample = "moje.imię@jakaś.domena";
Boolean validEmail = Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", sample);
```

Nie jest to z pewnością idealny walidator adresów e-mailowych, ale zdecydowanie jest to dobry początek i wystarczy dla naszego prostego systemu logowania, gdy dodamy już obsługę sieci. Jeśli chcesz pomajstrować przy tym wzorcu walidacji i rozwinąć go lub ulepszyć, pamiętaj, że w `jshell` możesz „ponownie używać" wpisanych linii poleceń za pomocą przycisków strzałek na klawiaturze. Aby odzyskać poprzednio wpisaną linię, użyj strzałki w górę. Ogólnie rzecz biorąc, za pomocą strzałki w górę i strzałki w dół możesz nawigować po wszystkich swoich wpisanych liniach. W obrębie linii możesz używać strzałek w lewo i w prawo, aby edytować polecenie. Następnie wystarczy nacisnąć przycisk *Enter*, żeby uruchomić zmodyfikowane polecenie — przed naciśnięciem przycisku *Enter* nie trzeba przesuwać kursora na koniec linii.

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "dobry@jakaś.domena")
$1 ==> true

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "dobry@helion.pl")
$2 ==> true

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "helion.pl")
$3 ==> false

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "zły@helion@pl")
$4 ==> false
```

[6] Oczywiście, jeśli masz takie życzenie i luźne kilka(set) tysięcy dolarów, możesz ubiegać się o własną niestandardową globalną domenę najwyższego poziomu (*https://oreil.ly/lMRnm*).

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "ja@helion.PL")
$5 ==> true

jshell> Pattern.matches("[^@]+@[^@]+\\.[a-z]+", "ja@helion.PL")
$6 ==> false
```

W powyższych przykładach wpisaliśmy linię `Pattern.matches(...)` w całości tylko raz. Potem dla kolejnych pięciu linii wystarczyło tylko naciskać strzałkę w górę, edytować polecenie i przyciskać *Enter*. Czy widzisz, dlaczego ostatni test dopasowania zwrócił `false`?

Klasa Matcher

`Matcher` przeprowadza powiązanie wzorca z łańcuchem znaków i zapewnia narzędzia, które pozwalają wykonywać przeszukiwanie, iterowanie oraz testowanie dopasowań wzorca na danym łańcuchu znaków. Klasa `Matcher` jest „stanowa”. Metoda `find()`przy każdym wywołaniu próbuje na przykład znaleźć następne dopasowanie. Ale wywołując metodę `reset()`, możesz wyczyścić obiekt `Matcher` i zacząć od nowa.

Jeśli jesteś zainteresowany „jednym wielkim dopasowaniem” — czyli oczekujesz, że Twój łańcuch znaków będzie pasował do wzorca lub nie — możesz użyć metody `matches()` lub `lookingAt()`. Te dwie metody odpowiadają mniej więcej odpowiednio metodom `equals()` i `startsWith()` klasy `String`. Metoda `matches()` sprawdza, czy dany łańcuch znaków w całości pasuje do wzorca (bez pozostawiania luzem żadnych znaków z łańcucha) i zwraca wartość `true` lub `false`. Metoda `lookingAt()` działa podobnie, z tym wyjątkiem, że sprawdza jedynie, czy dany łańcuch znaków zaczyna się zgodnie z wzorcem i nie przejmuje się dopasowaniem do wzorca reszty znaków z łańcucha.

Generalnie powinniśmy mieć możliwość przeszukiwania danego łańcucha znaków i znajdowania dopasowań. W tym celu można użyć metody `find()`. Każde wywołanie `find()` zwraca wartość `true` lub `false` dla kolejnego dopasowania wzorca i wewnętrznie odnotowuje pozycję pasującego tekstu. Pozycję początkowego i końcowego znaku można uzyskać za pomocą metod `start()` i `end()` klasy `Matcher` albo po prostu można pobrać dopasowany tekst za pomocą metody `group()`. Oto przykład:

```
import java.util.regex.*;

String text="A horse is a horse, of course of course...";
String pattern="horse|course";

Matcher matcher = Pattern.compile( pattern ).matcher( text );
while ( matcher.find() )
    System.out.println(
        "Dopasowano: '"+matcher.group()+"' w pozycji "+matcher.start() );
```

Powyższy fragment kodu wypisuje początkową lokalizację słów `horse` i `course` (w sumie cztery dopasowania):

```
Dopasowano: 'horse' w pozycji 2
Dopasowano: 'horse' w pozycji 13
Dopasowano: 'course' w pozycji 23
Dopasowano: 'course' w pozycji 33
```

Metoda służąca do pobierania dopasowanego tekstu została nazwana `group()` (grupa), ponieważ odnosi się do przechwytywania grupy o numerze zero (całego dopasowania). Można także pobierać tekst innych numerowanych grup przechwytywania, podając metodzie `group()` argument w postaci liczby całkowitej. Do określania, ile jest grup przechwytywania, służy metoda `groupCount()`:

```
for (int i=1; i < matcher.groupCount(); i++)
System.out.println( matcher.group(i) );
```

Dzielenie i tokenizacja łańcuchów znaków

Bardzo często wymagane jest parsowanie łańcucha znaków do postaci kilku pól przy użyciu jakiegoś separatora, np. przecinka. Jest to tak powszechny problem, że klasa `String` zawiera specjalnie przeznaczoną do tego celu metodę. Jest to metoda `split()`, która przyjmuje wyrażenie regularne i zwraca tablicę fragmentów danego łańcucha znaków podzielonego zgodnie z określonym wzorcem. Rozważmy następujący przykład łańcucha znaków i wywołań metody `split()`:

```
String text = "Foo, bar ,   blah";
String[] badFields = text.split(",");
String[] goodFields = text.split( "\\s*,\\s*" );
```

Pierwsze wywołanie `split()` zwraca tablicę obiektów `String`, ale naiwne użycie `","` do rozdzielenia łańcucha znaków oznacza, że spacje z naszej zmiennej `text` pozostają przyczepione do interesujących nas znaków. Najpierw zgodnie z oczekiwaniami otrzymujemy `Foo` jako pojedyncze słowo, ale potem dostajemy `bar<space>`, a na koniec `<space><space><space>blah`. Fuj! Drugie wywołanie metody `split()` również daje tablicę obiektów `String`, ale tym razem zawierającą oczekiwane obiekty `Foo`, `bar` (bez spacji na końcu) i `blah` (bez spacji na początku).

Jeśli zamierzasz używać w swoim kodzie takiej operacji więcej niż kilka razy, powinieneś raczej skompilować taki wzorzec i używać jego metody `split()`, która jest identyczna z wersją tej metody z klasy `String`. Metoda `split()` klasy `String` jest równoważna z następującym kodem:

```
Pattern.compile(pattern).split(string);
```

Jak zauważyliśmy wcześniej, wyrażenia regularne to o wiele szerszy temat niż konkretne funkcjonalności regex zapewniane przez Javę. Zalecamy, żebyś wrócił jeszcze do tego fragmentu rozdziału, w którym omawialiśmy klasę `Pattern`, i użył narzędzia `jshell`, aby pobawić się wyrażeniami regularnymi i poćwiczyć dzielenie łańcuchów znaków. W tej kwestii bardzo ważna jest praktyka.

Narzędzia matematyczne

Język Java ma wbudowaną bezpośrednią obsługę arytmetyki liczb całkowitych i zmiennoprzecinkowych. Bardziej złożone operacje matematyczne są obsługiwane przez klasę `java.lang.Math`. Jak już zapewne wiesz, klasy opakowujące dla prostych typów danych umożliwiają traktowanie ich jako obiektów. Klasy opakowujące zawierają również kilka metod dla przeprowadzania podstawowych konwersji.

Najpierw kilka słów na temat wbudowanej w Javie arytmetyki. Błędy w operacjach arytmetycznych na liczbach całkowitych są w Javie obsługiwane poprzez rzucanie wyjątku `ArithmeticException`:

```
int zero = 0;

try {
    int i = 72 / zero;
} catch ( ArithmeticException e ) {
    // Dzielenie przez zero
}
```

Aby wygenerować błąd w tym przykładzie, utworzyliśmy pośrednią zmienną o nazwie `zero`. Kompilator jest dość sprytny i złapałby nas, gdybyśmy bezczelnie próbowali wykonać dzielenie przez literalne zero.

Natomiast zmiennoprzecinkowe wyrażenia arytmetyczne nie rzucają wyjątków. Zamiast tego przyjmują specjalne wartości wyjścia poza zakres pokazane w tabeli 8.2.

Tabela 8.2. Specjalne wartości zmiennoprzecinkowe

Wartość	Reprezentacja matematyczna
`POSITIVE_INFINITY`	`1.0/0.0`
`NEGATIVE_INFINITY`	`-1.0/0.0`
NaN	`0.0/0.0`

Poniższy przykład generuje wynik nieskończoności:

```
double zero = 0.0;
double d = 1.0/zero;
if ( d == Double.POSITIVE_INFINITY )
    System.out.println( "Dzielenie przez zero" );
```

Wynik dzielenia zera przez zero wskazuje specjalna wartość NaN (ang. *Not a Number*), czyli tzw. nie-liczba. Ta wartość cechuje się specjalną matematyczną relacją polegającą na tym, że nie jest równa sama sobie (`NaN != NaN` ewaluuje do wartości `true`). Do testowania pod kątem liczby NaN używa się metod `Float.isNaN()` lub `Double.isNaN()`.

Klasa java.lang.Math

Klasa `java.lang.Math` to matematyczna biblioteka Javy. Zawiera zestaw statycznych metod obejmujących wszystkie typowe operacje matematyczne, takie jak `sin()`, `cos()` i `sqrt()`. Klasa `Math` nie jest zbyt obiektowa (nie można utworzyć instancji klasy `Math`). Jest ona po prostu pomocniczym elementem do przechowywania statycznych metod, które przypominają bardziej globalne funkcje. Jak dowiedziałeś się w rozdziale 5., aby zaimportować nazwy takich statycznych metod i stałych bezpośrednio do zakresu określonej klasy i używać ich za pomocą prostych, niekwalifikowanych nazw, można użyć funkcji importu statycznego.

Tabela 8.3 podsumowuje metody klasy java.lang.Math.

Tabela 8.3. *Metody klasy java.lang.Math*

Metoda	Typ argumentu	Funkcjonalność
`Math.abs(a)`	`int, long, float, double`	Wartość bezwzględna
`Math.acos(a)`	`double`	Arcus cosinus
`Math.asin(a)`	`double`	Arcus sinus
`Math.atan(a)`	`double`	Arcus tangens
`Math.atan2(a,b)`	`double`	Część kątowa transformacji współrzędnych układu prostokątnego na układ biegunowy
`Math.ceil(a)`	`double`	Najmniejsza liczba całkowita większa niż lub równa *a*
`Math.cbrt(a)`	`double`	Pierwiastek sześcienny z liczby *a*
`Math.cos(a)`	`double`	Cosinus
`Math.cosh(a)`	`double`	Cosinus hiperboliczny
`Math.exp(a)`	`double`	Stała `Math.E` do potęgi *a*
`Math.floor(a)`	`double`	Największa liczba całkowita mniejsza niż lub równa *a*
`Math.hypot(a, b)`	`double`	Precyzyjne obliczanie `sqrt()` z $a^2 + b^2$
`Math.log(a)`	`double`	Logarytm naturalny liczby *a*
`Math.log10(a)`	`double`	Logarytm dziesiętny liczby *a*
`Math.max(a, b)`	`int, long, float, double`	Wartość *a* lub *b* bliższa stałej `Long.MAX_VALUE`
`Math.min(a, b)`	`int, long, float, double`	Wartość *a* lub *b* bliższa stałej `Long.MIN_VALUE`
`Math.pow(a, b)`	`double`	Liczba *a* do potęgi *b*
`Math.random()`	Brak	Generator liczb losowych
`Math.rint(a)`	`double`	Konwertuje wartość podwójną na wartość całkowitą w formacie podwójnym
`Math.round(a)`	`float, double`	Zaokrągla do pełnej liczby
`Math.signum(a)`	`double, float`	Pobiera dla danej liczby wartość signum: `1.0`, `-1.0` lub `0`
`Math.sin(a)`	`double`	Sinus
`Math.sinh(a)`	`double`	Sinus hiperboliczny
`Math.sqrt(a)`	`double`	Pierwiastek kwadratowy
`Math.tan(a)`	`double`	Tangens
`Math.tanh(a)`	`double`	Tangens hiperboliczny
`Math.toDegrees(a)`	`double`	Konwertuje radiany na stopnie
`Math.toRadians(a)`	`double`	Konwertuje stopnie na radiany

Metody `log()`, `pow()` i `sqrt()` mogą podczas wykonywania programu rzucić wyjątek `ArithmeticException`. Metody `abs()`, `max()` i `min()` są przeciążone dla wszystkich wartości skalarnych: `int`, `long`, `float` lub `double`, i zwracają odpowiedni typ. Wersje metody `Math.round()` przyjmują typ `float` albo `double` i zwracają odpowiednio `int` lub `long`. Reszta metod działa na wartościach `double` i taki typ również zwraca:

```
double irrational = Math.sqrt( 2.0 );  // 1.414...
int bigger = Math.max( 3, 4 );          // 4
long one = Math.round( 1.125798 );      // 1
```

Aby podkreślić wygodę opcji statycznego importu, możemy wypróbować te proste funkcje w `jshell`:

```
jshell> import static java.lang.Math.*

jshell> double irrational = sqrt(2.0)
irrational ==> 1.4142135623730951

jshell> int bigger = max(3,4)
bigger ==> 4

jshell> long one = round(1.125798)
one ==> 1
```

Klasa `Math` zawiera również dwie wartości `static final double`, `E` oraz `PI`:

```
double circumference = diameter * Math.PI;
```

Klasa Math w akcji

Zdarzyło się nam już korzystać z klasy `Math` i jej statycznych metod w rozdziale 5. w podrozdziale „Klasy” w punkcie „Uzyskiwanie dostępu do pól i metod”. Możemy użyć jej ponownie, aby uatrakcyjnić nieco naszą grę poprzez losowe rozmieszczanie drzew. Metoda `Math.random()` zwraca losową wartość `double` większą lub równą 0 i mniejszą niż 1. Wystarczy dodać trochę operacji arytmetycznych oraz zaokrąglania lub redukowania dokładności i można używać uzyskiwanych wartości do tworzenia losowych liczb w dowolnym zakresie. Do przekształcania takiej wartości w żądany zakres używa się następującego wzoru:

```
int randomValue = min + (int)(Math.random() * (max - min));
```

Wypróbuj to! Spróbuj w `jshell` pogenerować losowe liczby czterocyfrowe. Wartość minimalną (`min`) i maksymalną (`max`) możesz ustawić odpowiednio na 1000 i 10000 w taki sposób:

```
jshell> int min = 1000
min ==> 1000

jshell> int max = 10000
max ==> 10000

jshell> int fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9603

jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9178
```

```
jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 3789
```

Do rozmieszczania naszych drzew na planszy będziemy potrzebować dwóch liczb losowych dla współrzędnych `x` i `y`. Aby ustawić zakres, który utrzyma drzewa na ekranie, trzeba pomyśleć o marginesie wokół brzegów ekranu. Jeden ze sposób na zrobienie tego dla współrzędnej `x` może wyglądać następująco:

```
private int goodX() {
    // Co najmniej połowa szerokości drzewa plus kilka pikseli
    int leftMargin = Field.TREE_WIDTH_IN_PIXELS / 2 + 5;
    // Teraz szukamy losowej liczby pomiędzy lewym i prawym marginesem
    int rightMargin = FIELD_WIDTH - leftMargin;

    // I zwracamy losową liczbę zaczynając od lewego marginesu
    return leftMargin + (int)(Math.random() * (rightMargin - leftMargin));
}
```

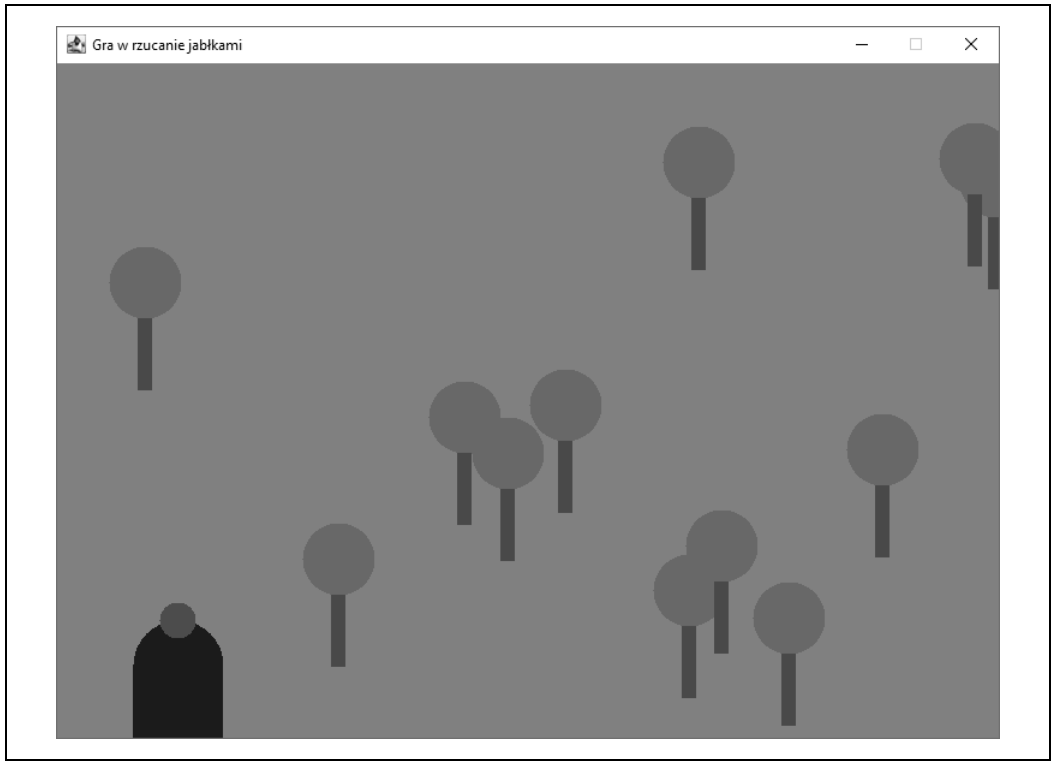
Jeżeli skonfigurujesz podobną metodę dla wyznaczania wartości współrzędnej `y`, powinieneś zobaczyć obraz podobny do tego, który pokazano na rysunku 8.1. Możesz nawet pójść jeszcze dalej i skorzystać z omówionej w rozdziale 5. metody `isTouching()`, aby uniknąć rozmieszczania drzew stykających się z fizykiem. Oto nasza ulepszona pętla konfiguracji drzew:

```
for (int i = field.trees.size(); i < Field.MAX_TREES; i++) {
    Tree t = new Tree();
    t.setPosition(goodX(), goodY());
    // Drzewa mogą być blisko siebie i nachodzić na siebie,
    // ale nie powinny stykać się z fizykami
    while(player1.isTouching(t)) {
        // To drzewo się styka, więc spróbujmy jeszcze raz
        t.setPosition(goodX(), goodY());
        System.err.println("Zmiana pozycji stykającego się drzewa...");
    }
    field.addTree(t);
}
```

Spróbuj zamknąć grę i uruchomić ją ponownie. Przy każdym uruchomieniu aplikacji powinieneś zobaczyć drzewa rozmieszczone w innych miejscach.

Duże i dokładne liczby

Jeśli typy `long` i `double` nie są dla Ciebie wystarczająco duże lub dokładne, pakiet `java.math` zapewnia dwie klasy, `BigInteger` i `BigDecimal`, które obsługują liczby o dowolnej dokładności. Te bogate w funkcjonalności klasy posiadają grupę metod przeznaczonych do wykonywania obliczeń o dowolnej dokładności i precyzyjnego kontrolowania zaokrąglania reszt. W poniższym przykładzie używamy klasy `BigDecimal`, aby dodać dwie bardzo duże liczby, a następnie wygenerować wynik ze stoma miejscami po przecinku:



Rysunek 8.1. Losowo rozmieszczone drzewa

```
long l1 = 9223372036854775807L;  // Long.MAX_VALUE
long l2 = 9223372036854775807L;
System.out.println( l1 + l2 );  // -2 ! Nie za dobrze

try {
    BigDecimal bd1 = new BigDecimal( "9223372036854775807" );
    BigDecimal bd2 = new BigDecimal( 9223372036854775807L );

    System.out.println( bd1.add( bd2 ) );  // 18446744073709551614

    BigDecimal numerator = new BigDecimal(1);
    BigDecimal denominator = new BigDecimal(3);
    BigDecimal fraction =
        numerator.divide( denominator, 100, BigDecimal.ROUND_UP );
    // 100 miejsc po przecinku = 0.333333 ... 3334
}
catch (NumberFormatException nfe) { }
catch (ArithmeticException ae) { }
```

Klasa `BigInteger` jest kluczowa, jeśli chcesz dla zabawy implementować algorytmy kryptograficzne lub naukowe. Natomiast klasę `BigDecimal` można z kolei znaleźć w aplikacjach mających do czynienia z danymi walutowymi i finansowymi. Poza tymi zastosowaniami prawdopodobnie nie będziesz potrzebować tych klas.

Daty i godziny

Bez odpowiednich narzędzi praca z datami i godzinami może być uciążliwa. Przed wydaniem Javy 8 dostępne były trzy klasy, które obsługiwały większość zadań. Klasa `java.util.Date` hermetyzowała określony punkt w czasie. Klasa `java.util.GregorianCalendar`, która rozszerza abstrakcyjną klasę `java.util.Calendar`, przekładała punkt w czasie na pola kalendarza, takie jak miesiąc, dzień i rok. Ostatnia z nich, klasa `java.text.DateFormat`, potrafiła generować i parsować tekstowe reprezentacje dat i godzin w wielu językach.

Chociaż klasy `Date` i `Calendar` obejmowały wiele przypadków użycia, zapewniały niewystarczającą szczegółowość, poza tym brakowało im kilku innych funkcjonalności. Spowodowało to powstanie kilku zewnętrznych bibliotek, które miały na celu ułatwienie programistom pracy z datami, godzinami i czasem trwania. Java 8 zapewniła bardzo potrzebne w tym zakresie ulepszenia dzięki dodaniu pakietu `java.time`. Przyjrzymy się temu nowemu pakietowi, ale nadal będziesz napotykać w kodach bardzo wiele przypadków użycia klas `Date` i `Calendar`, więc warto wiedzieć o ich istnieniu. Jak zawsze nieocenionym źródłem informacji o tych częściach API Javy, których nie będziemy tutaj omawiać, jest dokumentacja udostępniona w internecie (*https://oreil.ly/Behlk*).

Lokalne daty i godziny

Klasa `java.time.LocalDate` reprezentuje datę dla danego regionu bez informacji o czasie. Przypomnij sobie np. jakiś dzień wolny od pracy, taki jak 3 maja 2020 r. Analogicznie klasa `java.time.LocalTime` reprezentuje czas bez żadnych informacji o dacie. Być może Twój budzik dzwoni codziennie o 7:15. Klasa `java.time.LocalDateTime` przechowuje zarówno wartości daty, jak i godziny, na przykład dla umówionej wizyty u okulisty, abyś mógł dalej bez problemów czytać książki o Javie. Wszystkie te klasy oferują statyczne metody do tworzenia nowych instancji: przy użyciu odpowiednich wartości liczbowych za pomocą metody `of()` lub poprzez parsowanie łańcuchów znaków za pomocą metody `parse()`. Uruchommy `jshell` i spróbujmy utworzyć kilka przykładów.

```
jshell> import java.time.*

jshell> LocalDate.of(2019,5,4)
$2 ==> 2019-05-04

jshell> LocalDate.parse("2019-05-04")
$3 ==> 2019-05-04

jshell> LocalTime.of(7,15)
$4 ==> 07:15

jshell> LocalTime.parse("07:15")
$5 ==> 07:15

jshell> LocalDateTime.of(2019,5,4,7,0)
$6 ==> 2019-05-04T07:00
jshell> LocalDateTime.parse("2019-05-04T07:15")
$7 ==> 2019-05-04T07:15
```

Inną świetną statyczną metodą służącą do tworzenia tych obiektów jest metoda `now()`, która jak można się spodziewać, podaje bieżącą datę lub godzinę albo datę i godzinę:

```
jshell> LocalTime.now()
$8 ==> 15:57:24.052935

jshell> LocalDate.now()
$9 ==> 2019-12-12

jshell> LocalDateTime.now()
$10 ==> 2019-12-12T15:57:37.909038
```

Świetnie! Po zaimportowaniu pakietu `java.time` można tworzyć instancje wszystkich klas `Local...` dla określonych chwil lub dla „teraz". Prawdopodobnie zauważyłeś, że obiekty utworzone za pomocą metody `now()` zawierają sekundy i nanosekundy. W razie potrzeby możesz podawać te wartości metodom `of()` i `parse()`. Nie ma w tym nic ekscytującego, ale kiedy będziesz już mieć te obiekty, będziesz mógł wiele z nimi zrobić. Czytaj dalej!

Porównywanie oraz zmiana dat i godzin

Jedną z wielkich zalet korzystania z klas pakietu `java.time` jest spójny zestaw metod do porównywania i zmiany dat i godzin. Wiele aplikacji czatowych pokazuje na przykład, „jak dawno" została wysłana wiadomość. Podpakiet `java.time.temporal` ma właśnie to, czego potrzebujemy: interfejs `ChronoUnit`. Zawiera on kilka jednostek daty i godziny, takich jak `MONTHS` (miesiące), `DAYS` (dni), `HOURS` (godziny), `MINUTES` (minuty) itd. Tych jednostek można używać do obliczania różnic czasowych. Przy użyciu metody `between()` możemy obliczyć chociażby, ile czasu zajmuje nam w `jshell` utworzenie dwóch przykładowych dat i godzin:

```
jshell> LocalDateTime first = LocalDateTime.now()
first ==> 2019-12-12T16:03:21.875196

jshell> LocalDateTime second = LocalDateTime.now()
second ==> 2019-12-12T16:03:33.175675

jshell> import java.time.temporal.*

jshell> ChronoUnit.SECONDS.between(first, second)
$12 ==> 11
```

Jak widać wpisanie linii tworzącej naszą drugą zmienną `second` zajęło około 11 sekund. Powinieneś zapoznać się z dokumentacją interfejsu `ChronoUnit` (*https://oreil.ly/BhCr2*), gdzie znajdziesz pełną listę dostępnych jednostek, ale wystarczy powiedzieć, że otrzymujesz pełny zakres od nanosekund do tysiącleci.

Jednostki te mogą również pomóc w manipulowaniu datami i godzinami za pomocą metod `plus()` i `minus()`. Można na przykład ustawić przypomnienie na za tydzień od dzisiaj:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-12

jshell> LocalDate reminder = today.plus(1, ChronoUnit.WEEKS)
reminder ==> 2019-12-19
```

Nieźle! Ten przykład przypomnienia (`reminder`) przywołuje jednak kolejną manipulację, którą być może trzeba będzie przeprowadzać od czasu do czasu. Możemy zażyczyć sobie przypomnienia 19 grud-

nia o konkretnej godzinie. Konwersję między datami lub godzinami a datami z godzinami można z łatwością przeprowadzić za pomocą metod odpowiednio `atDate()` i `atTime()`:

```
jshell> LocalDateTime betterReminder = reminder.atTime(LocalTime.of(9,0))
betterReminder ==> 2019-12-19T09:00
```

Teraz otrzymamy to przypomnienie o 9:00. Co sie jednak stanie, jeżeli ustawimy to przypomnienie w Atlancie, a następnie polecimy do San Francisco? Kiedy włączy się alarm? Nie musimy się tym przejmować, ponieważ jak wskazuje nazwa, typ `LocalDateTime` jest lokalny! Więc kiedy uruchomimy program, fragment `T09:00` będzie nadal określał 9:00, gdziekolwiek będziemy. Jeśli jednak będziemy mieli do czynienia z czymś takim jak współdzielony kalendarz i harmonogram spotkań, nie będziemy mogli zignorować różnych stref czasowych. Na szczęście w pakiecie `java.time` uwzględniono również tę kwestię.

Strefy czasowe

Autorzy nowego pakietu `java.time` zachęcają oczywiście do korzystania z lokalnych odmian klas godziny i daty w takich sytuacjach, które tego wymagają. Dodanie obsługi stref czasowych oznacza jednak zwiększenie złożoności aplikacji, a jak wiadomo, powinno się tego unikać, jeśli tylko jest to możliwe. W wielu scenariuszach obsługa stref czasowych jest tak czy inaczej nieuchronna. Ze „strefowymi" datami i godzinami można pracować przy użyciu klas `ZonedDateTime` i `OffsetDateTime`. Wariant strefowy rozpoznaje nazwane strefy czasowe i uwzględnia takie kwestie, jak zmiany czasu zimowego na letni. Wariant offsetowy jest stałym, prostym przesunięciem liczbowym względem czasu uniwersalnego (UTC Greenwich).

Większość dat i godzin skierowanych do użytkowników opiera się na strefach nazwanych, więc przyjrzyjmy się tworzeniu strefowej daty z godziną. Aby dodać strefę, używamy klasy `ZoneId`, która do tworzenia nowych instancji ma powszechną statyczną metodę `of()`. Aby uzyskać wartość strefową, można podać strefę regionu jako obiekt `String`:

```
jshell> LocalDateTime piLocal = LocalDateTime.parse("2019-03-14T01:59")
piLocal ==> 2019-03-14T01:59

jshell> ZonedDateTime piCentral =
piLocal.atZone(ZoneId.of("America/Chicago"))
piCentral ==> 2019-03-14T01:59-05:00[America/Chicago]
```

A teraz, używając nieco rozwlekle, ale trafnie nazwanej metody `withZoneSameInstant()` (z tą samą instancją strefy), możesz np. upewnić się, że Twoi znajomi z Paryża będą mogli połączyć się z Tobą w odpowiednim momencie:

```
jshell> ZonedDateTime piAlaMode =
piCentral.withZoneSameInstant(ZoneId.of("Europe/Paris"))
piAlaMode ==> 2019-03-14T07:59+01:00[Europe/Paris]
```

Jeśli masz jeszcze innych znajomych, którzy nie są dogodnie zlokalizowani w dużym regionie metropolitalnym, ale chcesz, aby również dołączyli do rozmowy, możesz użyć metody `systemDefault()` klasy `ZoneId`, co spowoduje, że ich strefa czasowa zostanie wybrana programowo:

```
jshell> ZonedDateTime piOther =
piCentral.withZoneSameInstant(ZoneId.systemDefault())
piOther ==> 2019-03-14T02:59-04:00[America/New_York]
```

W naszym przypadku narzędzie `jshell` zostało uruchomione na laptopie w standardowej wschodniej strefie czasowej (nie w okresie czasu letniego) w Stanach Zjednoczonych, a wartość `piOther` okazała się zgodna z oczekiwaniami. Identyfikator strefy metody `systemDefault()` to bardzo przydatny sposób szybkiego dostosowywania dat i godzin z innej strefy, aby pasowały do tego, co wskazuje zegar i kalendarz użytkownika Twojej aplikacji. W aplikacjach komercyjnych można ewentualnie umożliwić użytkownikowi wybranie preferowanej strefy czasowej, ale zwykle dobrym pomysłem jest właśnie użycie metody `systemDefault()`.

Parsowanie i formatowanie dat i godzin

Przy tworzeniu i wyświetlaniu naszych lokalnych i strefowych dat i godzin za pomocą łańcuchów znaków korzystamy z domyślnych formatów, które są zgodne z wartościami ISO i generalnie działają wszędzie tam, gdzie musimy przyjmować lub wyświetlać daty i godziny. Każdy programista wie jednak, że „generalnie" nie znaczy „zawsze". Na szczęście zarówno do parsowania danych wejściowych, jak i formatowania danych wyjściowych można użyć klasy narzędziowej `java.time.format.DateTimeFormatter`.

Podstawowym zadaniem klasy `DateTimeFormatter` jest budowanie łańcucha znaków dla formatu (ang. *format string*), który zarządza zarówno parsowaniam, jak i formatowaniem. Format tworzy się za pomocą elementów wymienionych w tabeli 8.4. Podaliśmy w niej tylko część dostępnych opcji, ale powinny one pomóc Ci obsłużyć większość przypadków użycia związanych z datą i godziną, które napotkasz na swojej drodze. Zwróć uwagę, że przy stosowaniu wymienionych w tabeli 8.4 znaków wielkość liter ma znaczenie!

Tabela 8.4. Popularne elementy klasy DateTimeFormatter

Znak	Opis	Przykład
`y`	Rok ery	`2004; 04`
`M`	Miesiąc roku	`7; 07`
`L`	Miesiąc roku	`Jul; July; J`
`d`	Dzień miesiąca	`10`
`E`	Dzień tygodnia	`Tue; Tuesday; T`
`a`	Przed południem i po południu	`PM`
`h`	Godzina zegarowa przed południem i po południu (1 – 12)	`12`
`K`	Godzina przed południem i po południu	`0`
`k`	Godzina zegarowa dnia (1 – 24)	`24`
`H`	Godzina dnia (0 – 23)	`0`
`m`	Minuta godziny	`30`
`s`	Sekunda minuty	`55`
`S`	Ułamek sekundy	`033954`
`z`	Nazwa strefy czasowej	Pacific Standard Time; PST
`Z`	Przesunięcie strefy czasowej	+0000; –0800; –08:00

Aby utworzyć na przykład typowy format daty stosowany w Stanach Zjednoczonych, można użyć znaków `M`, `d` oraz `y`. Formater buduje się za pomocą statycznej metody `ofPattern()`. Gotowego formatera można użyć (lub używać wielokrotnie) z metodą `parse()` dowolnej klasy daty lub godziny:

```
jshell> import java.time.format.DateTimeFormatter

jshell> DateTimeFormatter shortUS = DateTimeFormatter.ofPattern("MM/dd/yy")
shortUS ==> Value(MonthOfYe ... (YearOfEra,2,2,2000-01-01)

jshell> LocalDate valentines = LocalDate.parse("02/14/19", shortUS)
valentines ==> 2019-02-14

jshell> LocalDate piDay = LocalDate.parse("03/14/19", shortUS)
piDay ==> 2019-03-14
```

Jak wspomnieliśmy wcześniej, formater działa w obu kierunkach. Aby wygenerować tekstową reprezentację daty lub godziny, wystarczy użyć metody `format()` utworzonego formatera:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-14

jshell> shortUS.format(today)
$30 ==> "12/14/19

jshell> shortUS.format(piDay)
$31 ==> "03/14/19
```

Oczywiście formatery działają zarówno dla godzin, jak i dat z godzinami!

```
jshell> DateTimeFormatter military = DateTimeFormatter.ofPattern("HHmm")
military ==> Value(HourOfDay,2)Value(MinuteOfHour,2)

jshell> LocalTime sunset = LocalTime.parse("2020", military)
sunset ==> 20:20

jshell> DateTimeFormatter basic = DateTimeFormatter.ofPattern("h:mm a")
basic ==> Value(ClockHourOfAmPm)':
'Value(MinuteOfHour,2)' 'Text(AmPmOfDay,SHORT)

jshell> basic.format(sunset)
$42 ==> "8:20 PM"

jshell> DateTimeFormatter appointment =
DateTimeFormatter.ofPattern("h:mm a MM/dd/yy z")
appointment ==>
Value(ClockHourOfAmPm)':' ...
0-01-01)' 'ZoneText(SHORT)

jshell> ZonedDateTime dentist =
ZonedDateTime.parse("10:30 AM 11/01/19 EST", appointment)
dentist ==> 2019-11-01T10:30-04:00[America/New_York]

jshell> ZonedDateTime nowEST = ZonedDateTime.now()
nowEST ==> 2019-12-14T09:55:58.493006-05:00[America/New_York]

jshell> appointment.format(nowEST)
$47 ==> "9:55 AM 12/14/19 EST"
```

Zwróć uwagę, że pod koniec powyższego przykładu identyfikator strefy czasowej (znak z) umieściliśmy w części `ZonedDateTime` — prawdopodobnie nie tam, gdzie się tego spodziewałeś! Chcieliśmy w ten sposób zilustrować szeroką funkcjonalność tych formatów. Można zaprojektować format uwzględniający bardzo szeroki zakres stylów danych wejściowych lub wyjściowych. Jako bezpośredni przykład tego, gdzie `DateTimeFormatter` może pomóc Ci zachować zdrowie psychiczne, na myśl przychodzą starsze systemy przechowywania danych i źle zaprojektowane formularze internetowe.

Błędy parsowania

Nawet z całą tą szeroką funkcjonalnością parsowania na wyciągnięcie ręki czasem coś może pójść nie tak. I niestety wyświetlane wyjątki są często zbyt niejasne, aby były od ręki przydatne. Rozważmy następującą próbę parsowania czasu z godzinami, minutami i sekundami:

```
jshell> DateTimeFormatter withSeconds =
DateTimeFormatter.ofPattern("hh:mm:ss")
withSeconds ==>
Value(ClockHourOfAmPm,2)':' ...
Value(SecondOfMinute,2)
jshell> LocalTime.parse("03:14:15", withSeconds)
|  Exception java.time.format.DateTimeParseException:
|  Text '03:14:15' could not be parsed: Unable to obtain
|  LocalTime from TemporalAccessor: {MinuteOfHour=14, MilliOfSecond=0,
|  SecondOfMinute=15, NanoOfSecond=0, HourOfAmPm=3,
|  MicroOfSecond=0},ISO of type java.time.format.Parsed
|        at DateTimeFormatter.createError (DateTimeFormatter.java:2020)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1955)
|        at LocalTime.parse (LocalTime.java:463)
|        at (#33:1)
|  Caused by: java.time.DateTimeException:
   Unable to obtain LocalTime from ...
|        at LocalTime.from (LocalTime.java:431)
|        at Parsed.query (Parsed.java:235)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1951)
|        ...
```

Rety! Wyjątek `DateTimeParseException` jest rzucany za każdym razem, gdy nie można sparsować wejściowego łańcucha znaków. Jest również rzucany w przypadkach takich jak nasz powyższy przykład. Pola z łańcucha znaków zostały sparsowane prawidłowo, ale nie dostarczyły wystarczającej ilości informacji do utworzenia obiektu `LocalTime`. Być może nie jest to oczywiste, ale nasz czas „3:14:15” może wskazywać zarówno na popołudnie, jak i na bardzo wczesny poranek. Winowajcą okazuje się wybranie przez nas dla godzin wzorca `hh`. Powinniśmy wybrać wzorzec godzinowy z jednoznaczną skalą 24-godzinną lub dodać bezpośredni element AM/PM (przed południem i po południu):

```
jshell> DateTimeFormatter valid1 = DateTimeFormatter.ofPattern("hh:mm:ss a")
valid1 ==> Value(ClockHourOfAmPm,
2)':'Value(MinuteOfHour,2)' ... 2)' 'Text(AmPmOfDay,SHORT)

jshell> DateTimeFormatter valid2 = DateTimeFormatter.ofPattern("HH:mm:ss")
valid2 ==> Value(HourOfDay,2)':'Value(MinuteOfHour,2)':'
Value(SecondOfMinute,2)
```

```
jshell> LocalTime piDay1 = LocalTime.parse("03:14:15 PM", valid1)
piDay1 ==> 15:14:15

jshell> LocalTime piDay2 = LocalTime.parse("03:14:15", valid2)
piDay2 ==> 03:14:15
```

Jeśli więc kiedykolwiek otrzymasz wyjątek `DateTimeParseException`, a dane wejściowe będą wyglądać na prawidłowe dopasowanie do formatu, sprawdź dokładnie, czy sam format zawiera wszystkie elementy niezbędne do utworzenia daty lub godziny. Ostatnia uwaga na temat tych wyjątków: być może do parsowania lat powinieneś użyć trudniejszego do skojarzenia znaku `u`.

Z klasą `DateTimeFormatter` związanych jest jeszcze naprawdę *bardzo* wiele innych szczegółowych informacji. W tym przypadku zapoznanie się z dokumentacją dostępną na stronie internetowej (*https://oreil.ly/rhosl*) jest o wiele bardziej warte zachodu niż w przypadku większości innych klas narzędziowych.

Znaczniki czasu

Kolejną popularną koncepcją czasu i daty uwzględnioną w pakiecie `java.time` jest pojęcie znacznika czasu (ang. *timestamp*). W każdej sytuacji, w której wymagane jest śledzenie przepływu informacji, będziesz potrzebować rekordu określającego dokładny moment jej wygenerowania lub zmodyfikowania. Z pewnością nadal będziesz spotykał użycie do przechowywania tych chwil w czasie klasy `java.util.Date`, ale klasa `java.time.Instant` przenosi wszystkie informacje potrzebne do utworzenia znacznika czasu, a ponadto oferuje wszystkie inne zalety pozostałych klas z pakietu `java.time`:

```
jshell> Instant time1 = Instant.now()
time1 ==> 2019-12-14T15:38:29.033954Z

jshell> Instant time2 = Instant.now()
time2 ==> 2019-12-14T15:38:46.095633Z

jshell> time1.isAfter(time2)
$54 ==> false

jshell> time1.plus(3, ChronoUnit.DAYS)
$55 ==> 2019-12-17T15:38:29.033954Z
```

Jeśli w Twojej pracy pojawiają się daty lub godziny, pakiet `java.time` będzie mile widzianym dodatkiem do Javy. Masz teraz dojrzały, dobrze zaprojektowany zestaw narzędzi do radzenia sobie z tymi danymi — nie potrzebujesz żadnych zewnętrznych bibliotek!

Inne przydatne narzędzia

Przyjrzeliśmy się niektórym elementom konstrukcyjnym Javy, w tym łańcuchom znaków i liczbom, a także jednej z najpopularniejszych kombinacji tych łańcuchów znaków i liczb — datom, w tym klasom `LocalDate` i `LocalTime`. Mamy nadzieję, że omówienie tej gamy narzędzi dało Ci wyobrażenie, w jaki sposób Java działa z wieloma prostymi lub typowymi elementami, które możesz napotkać przy rozwiązywaniu rzeczywistych problemów. Gorąco zachęcamy Cię do przeczytania

dokumentacji dla pakietów `java.util`, `java.text` i `java.time`, ponieważ znajdziesz tam więcej narzędzi, które mogą okazać się przydatne. Możesz na przykład przyjrzeć się klasie `java.util.Random`, służącej do generowania losowych współrzędnych drzew, które widziałeś na rysunku 8.1. Należy również zwrócić uwagę na to, że czasami prace związane z „usługami" są w rzeczywistości złożone i wymagają starannej dbałości o szczegóły. W internecie często można znaleźć przykłady kodu lub nawet pełne biblioteki napisane przez innych programistów, które mogą w jakiś sposób przyspieszyć Twoją pracę.

Następnym naszym krokiem będzie rozpoczęcie korzystania z tych bardziej podstawowych koncepcji i rozwijanie ich. Popularność Javy wciąż nie maleje, ponieważ oprócz obsługi podstawowych elementów zawiera również wsparcie dla bardziej zaawansowanych technik. Jedną z tych zaawansowanych technik, które odegrały istotną rolę w początkowym sukcesie Javy, są funkcjonalności związane z „wątkami", wprowadzane w życie w tym języku od wczesnej fazy jego rozwoju. Wątki zapewniają programistom lepszy dostęp do nowoczesnych, wielofunkcyjnych systemów, utrzymując wydajność aplikacji na określonym poziomie nawet podczas wykonywania wielu złożonych zadań. Zobaczmy więc, jak można wykorzystać obsługę tej sygnatury.

Java: niezawodny kod, aplikacja, która działa!

Twórcy Javy od początku historii tego języka śmiało wprowadzali kolejne awangardowe innowacje, a pisane w niej aplikacje miały swój udział w napędzaniu internetowego postępu. Obecnie Java jest uważana za najpopularniejszy język programowania na świecie, a miliony deweloperów wciąż tworzą za jej pomocą oprogramowanie dla niemal każdego urządzenia wyposażonego w procesor. Java jest wyjątkowo wszechstronnym narzędziem: pozwala napisać zarówno prostą aplikację mobilną, jak i złożony system internetowy. Pozostaje przy tym stosunkowo prosta w nauce — co sprawia, że jest idealnym językiem dla początkujących, którzy mają ambicję dojścia do profesjonalnego poziomu.

Ta książka jest praktycznym przewodnikiem dla każdego, kto chce zdobyć doświadczenie w tworzeniu rzeczywistych aplikacji w Javie. To również znakomity kurs programowania obiektowego dla początkujących, umożliwiający gruntowne zrozumienie podstaw języka Java i jego interfejsów API. Wyczerpująco opisano tu biblioteki klas, techniki programowania oraz idiomy. Nie zabrakło zaawansowanych zagadnień, takich jak wyrażenia lambda czy serwlety. W tym przejrzanym i zaktualizowanym wydaniu ujęto zmiany wprowadzone zarówno w wersji 11 Javy, jak i w przeglądowych wersjach 12, 13 i 14. Przedstawiono więc takie nowości jak interferencja typów w typach sparametryzowanych, ulepszenia w obsłudze wyjątków czy nowe środowisko testowe jshell.

W książce między innymi:

- przygotowanie środowiska pracy i konfiguracja przydatnych narzędzi
- typy, instrukcje, wyrażenia oraz obiekty w Javie
- obsługa wątków i pakiet współbieżności Javy
- błędy i wyjątki
- interfejs API wyrażeń regularnych

Marc Loy jest programistą i szkoleniowcem. Specjalizuje się w projektowaniu doświadczeń użytkownika i tworzeniu aplikacji mobilnych.

Patrick Niemeyer jest niezależnym konsultantem. Pisze książki techniczne dotyczące sieci i aplikacji rozproszonych.

Daniel Leuck jest dyrektorem generalnym spółki Ikayzo, firmy z oddziałami w Tokio i Honolulu, zajmującej się interaktywnym projektowaniem oprogramowania między innymi dla Sony, Oracle, Nomury i PIMCO.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7128-6

9 788328 371286

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł