helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

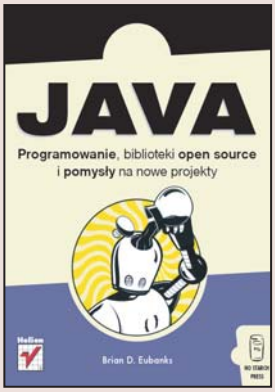
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Java. Programowanie, biblioteki open-source i pomysły na nowe projekty



Autor: Brian Eubanks
Tłumaczenie: Grzegorz Borkowski
ISBN: 83-246-0624-6
Tytuł oryginału: Wicked Cool Java: Code Bits, Open-Source Libraries, and Project Ideas
Format: B5, stron: 248

Odkryj nieznane możliwości Javy

- Sieci semantyczne i neuronowe
- Przetwarzanie grafiki i multimediów
- Obliczenia naukowe

Java, mimo stosunkowo krótkiej obecności na rynku, stała się jednym z najpopularniejszych języków programowania. Codziennie korzystają z niej setki tysięcy programistów z całego świata. Największe korporacje świata za jej pomocą budują systemy informatyczne przetwarzające potężne porcje danych. Aplikacje bazodanowe, serwlety i aplety to najbardziej znane zastosowania Javy, jednak nie jedyne. W sieci dostępna jest ogromna ilość bibliotek tworzonych przez pasjonatów, którzy wykorzystują Javę do odmiennych celów, takich jak przetwarzanie grafiki, modelowanie sieci neuronowych, przeprowadzanie złożonych obliczeń i wielu innych zadań.

Dzięki książce „Java. Programowanie, biblioteki open-source i pomysły na nowe projekty" poznasz mniej znane zastosowania Javy. Dowiesz się, jak za pomocą bibliotek dostępnych na licencji open-source tworzyć ciekawe projekty i pisać nietypowe aplikacje. Nauczysz się przetwarzać pliki XML i HTML, obrabiać i generować grafikę a także wyświetlać pliki multimedialne. Przeczytasz o sieciach semantycznych i neuronowych, odczytywaniu kanałów RSS i sterowaniu urządzeniami podłączonymi do komputera.

- Nieznane funkcje standardowego API Javy
- Przetwarzanie łańcuchów tekstowych
- Analiza plików XML i HTML
- Stosowanie RDF w projektach
- Czytanie kanałów RSS
- Obliczenia o dowolnej precyzji
- Realizacja algorytmów genetycznych
- Symulowanie sieci neuronowych
- Generowanie plików SVG
- Współpraca z interfejsem MIDI

Jeśli lubisz eksperymentować z językami programowania, ta książka będzie dla Ciebie doskonałym źródłem inspiracji

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

Spis treści

PODZIĘKOWANIA 9

WSTĘP 11

1
STANDARDOWE API JAVY 15
Użycie nowej wersji pętli for 16
Wykorzystanie konstrukcji enum 18
Mapy bez rzutowania w dół 21
Pisanie metod z parametrami generycznymi 22
Metody ze zmienną liczbą parametrów 25
Asercje w Javie 27
Użycie System.nanoTime 29
Uśpienie wątku na czas krótszy od milisekundy 30
Klasy anonimowe 31
Porównania == != .equals 33
Podsumowanie 35

2
NARZĘDZIA DO PRACY Z ŁAŃCUCHAMI TEKSTOWYMI 37
Użycie wyrażeń regularnych do wyszukiwania tekstów 38
Użycie metody String.split 40
Wyszukiwanie fragmentów w łańcuchach tekstowych 41
Użycie grup w wyrażeniach regularnych 42
Wykonywanie zamiany tekstów za pomocą wyrażeń regularnych 44
Przetwarzanie z użyciem klasy Scanner 47
Analiza skomplikowanej składni przy użyciu klasy Scanner 49
Generowanie przypadkowego tekstu 51
Drukowanie zawartości tablic w Javie 1.5 52
Kodowanie i dekodowanie danych binarnych 54

Formatowanie tekstów za pomocą MessageFormat ... 57
Powrót funkcji printf — formatowanie tekstów z klasą Formatter ... 58
Podsumowanie ... 59

3
PRZETWARZANIE XML I HTML ... 61
Szybkie wprowadzenie do XML ... 62
Użycie WebRowSet do utworzenia dokumentu XML ... 63
Zapamiętywanie zależności między elementami w SAX ... 64
Bezpośrednie wywoływanie zdarzeń obiektu ContentHandler ... 69
Filtrowanie zdarzeń interfejsu ContentHandler ... 71
Czytanie dokumentów XML z wykorzystaniem DOM4J ... 74
Użycie XPath do łatwego pobierania danych ... 76
Niewidoczne tagi, czyli filtrowanie dokumentu przed załadowaniem do DOM4J ... 80
Generowanie kodu analizatorów za pomocą JavaCC ... 83
Konwersja innych gramatyk na XML ... 87
Wykorzystanie techniki screen scraping do stron HTML ... 93
Wyszukiwanie z Lucene ... 95
Podsumowanie ... 97

4
SIEĆ SEMANTYCZNA ... 99
Krótkie wprowadzenie do N3 i Jena ... 101
Tworzenie słowników RDF na własne potrzeby ... 103
Użycie hierarchii RDF w Jena ... 106
Dołączanie Dublin Core do dokumentów HTML ... 108
Zapytania w Jena RDQL ... 109
Lojban, RDF i projekt Jorne ... 111
RSS i Informa ... 113
Czytanie źródeł RSS ... 115
Odpytywanie i aktualizacja kanałów RSS ... 116
Filtrowanie danych RSS ... 117
Podsumowanie ... 119

5
ZASTOSOWANIA W NAUKACH ŚCISŁYCH I MATEMATYCZNO-PRZYRODNICZYCH ... 121
Tworzenie i zastosowanie funktorów ... 122
Użycie funktorów złożonych ... 125
Bity dużego kalibru — BitVector z biblioteki Colt ... 126
Tworzenie tablic prawdy za pomocą BitMatrix ... 128
Dwa terafurlongi w dwa tygodnie — wielkości fizyczne z JScience ... 130
Krnąbrne ułamki — arytmetyka dowolnej precyzji ... 133
Funkcje algebraiczne w JScience ... 135
Łączenie tablic prawdy za pomocą portów ... 136
Łączenie za pomocą JGraphT ... 139

Łączenie ogólnych jednostek obliczeniowych 141
Budowanie sieci neuronowych z Joone 144
Użycie JGAP do algorytmów genetycznych 146
Tworzenie inteligentnych agentów przy użyciu Jade 149
Język angielski z JWorkNet 153
Podsumowanie 155

6
PRZETWARZANIE GRAFIKI I WIZUALIZACJA DANYCH 157
Definiowanie graficznego interfejsu aplikacji Javy w XML 158
Wizualizacja danych w SVG 160
Wyświetlanie obrazów SVG 163
Konwersja JGraphT do JGraphView 164
Użycie map atrybutów w JGraph 166
Tworzenie wykresów z JFreeChart 167
Tworzenie raportów w Javie 169
Prosta dwuwymiarowa wizualizacja danych 171
Użycie transformacji afinicznych w Java 2D 174
Budowanie aplikacji graficznych z funkcją „zoom” na pomocą Piccolo 176
Podsumowanie 177

7
MULTIMEDIA I SYNCHRONIZACJA WĄTKÓW 179
Tworzenie muzyki z JFugue 180
Użycie JFugue razem z Java Sound MIDI 181
Wysyłanie zdarzeń do urządzeń wyjściowych MIDI 183
Tworzenie dźwięków w JMusic 184
Użycie szumu i skomplikowanej syntezy w JMusic 186
Niskopoziomowy dostęp do Java Sound 189
Czytanie dźwięku z linii wejściowej 191
Użycie Java Speech do tworzenia mówiących programów 192
Odśmiecacz i Javolution 193
Synchronizacja wątków za pomocą CyclicBarrier 196
Podsumowanie 197

8
ROZRYWKA, INTEGRACJA I POMYSŁY NA NOWE PROJEKTY 199
Użycie Javy do sterowania robotem LEGO 200
Kontrolowanie myszy z użyciem klasy AWT Robot 201
Wybór dat z pomocą JCalendar 202
Użycie klasy HttpClient do obsługi metody POST 203
Symulacja systemu Cell Matrix w Javie 204
Cell Matrix i algorytmy genetyczne 206
Uruchamianie aplikacji z Ant 207
Skrypty BeanShell 208
Tworzenie testów JUnit 210

Użycie JXTA w aplikacjach Peer-to-Peer ... 211
Pakiet narzędziowy Globus oraz sieci rozproszone ... 212
Użycie Jabbera w aplikacjach ... 212
Pisanie w języku asemblera JVM ... 213
Połączenie programowania genetycznego z BCEL ... 214
Kompilowanie innych języków do kodu Javy ... 215
Wizualizacja gramatyki języka Lojban ... 215
Edytor instrumentów muzycznych ... 216
WordNet Explorer ... 216
Automatyczny generator RSS ... 217
Sieci neuronowe w robotach ... 217
Narzędzie zarządzania metadanymi (adnotacjami) Javy 5 ... 218
CVS i kontrola kodu źródłowego ... 218
Wykorzystaj SourceForge do swoich projektów ... 219
Posumowanie ... 219

SŁOWNICZEK ... 221

SKOROWIDZ ... 235

5

Zastosowania w naukach ścisłych i matematyczno-przyrodniczych



Zadziwiający jest fakt, że w wielu kręgach naukowych Fortran wciąż sprawuje niepodzielną władzę (wiem, wiem, też mnie to przeraża). Przyczyną tego stanu rzeczy niekoniecznie jest to, że Fortran jest wspaniałym językiem programowania, lecz raczej fakt, że jego standardowe biblioteki dostarczają ogromny zbiór operacji matematycznych, z których korzysta wielka rzesza istniejących programów. Java istnieje od ponad dekady, działa na większej liczbie dostępnych platform, ma standardowe API o bogatszych możliwościach i pozwala robić rzeczy niemożliwe w Fortranie. Dlaczego więc Java nie jest tak popularna w aplikacjach dla nauk ścisłych? Wynika to być może z faktu, że Java nie posiada dobrych bibliotek

matematycznych. Klasa `java.lang.Math` ma bardzo ograniczone możliwości, ale ponieważ jest to klasa ze standardowej dystrybucji Javy, niektórzy programiści aplikacji naukowych nie trudzą się zbytnio poszukiwaniem dodatkowych bibliotek. Może po prostu uważają, że „masz to, co widzisz" (ang. „what you see is what you get"). Ten obraz jednak się zmienia, gdyż powoli na scenę wkraczają nowe biblioteki — istnieje obecnie wiele projektów typu open-source, w których powstają naprawdę wspaniałe rozwiązania. W niniejszym rozdziale przyjrzymy się niektórym matematycznym i naukowym bibliotekom dostępnym dla Javy. Wśród zagadnień, w które się zagłębimy, znajdą się funktory, tablice prawdy, teoria grafów, jednostki fizyczne, sieci neuronowe, algorytmy genetyczne i sztuczna inteligencja.

Tworzenie i zastosowanie funktorów

JGA

JAVA 5+

Według słownika internetowego Merriam-Webster (dostępnego pod adresem *www.m-w.com*) **funktor** jest to „coś, co wykonuje funkcję lub operację". Z punktu widzenia programisty funktor to funkcja, która może być przekazana jako parametr i użyta jak każda inna zmienna. Wiele języków, jak na przykład C, posiada **wskaźniki na funkcje**. Wskaźniki te przechowują adres pamięci, pod którym ulokowana jest dana funkcja. W takich językach możesz przekazać funkcję do innej funkcji i zastosować ją dla różnych argumentów — na przykład dla każdego elementu kolekcji. Języki takie jak Scheme, Lisp czy Haskell używają czysto funkcyjnego stylu programowania, bardzo wydajnego w pewnych zastosowaniach (w szczególności w dziedzinie sztucznej inteligencji). Java nie posiada funkcji w stylu Lispa czy C, ale możemy zaimplementować taki sposób działania za pomocą interfejsów i klas bazowych.

Generic Algorithms for Java to jedna z implementacji typu open-source obiektów funkcyjnych. Mamy w niej do czynienia z klasami odpowiadającymi funktorom przyjmującym zero, jeden lub dwa argumenty:

`Generator`	Funktor bezargumentowy
`UnaryFunctor`	Funktor jednoargumentowy
`BinaryFunctor`	Funktor dwuargumentowy

Funktory te są zaprojektowane dla ścisłej współpracy z Javą 5, gdyż korzystają z typów generycznych (ang. *generics*) (które zostały omówione w rozdziale 1.). Klasa `Generator` jest w rzeczywistości zdefiniowana jako `Generator<R>`. Posiada ona bezargumentową metodę `gen`, która zwraca obiekt typu `R` określony w konstruktorze klasy. Aby utworzyć funktor bezargumentowy, możemy użyć następującego kodu:

```
import net.sf.jga.fn.Generator;
public class CubeGenerator extends Generator<Double> {
   double current = 0;
   public Double gen() {
      current = current + 1;
      return current * current * current;
   }
}
```

Widać tu wyraźnie sens nazwy `Generator`: napisana przez nas przykładowa klasa generuje sześciany kolejnych liczb naturalnych. Oprócz możliwości tworzenia własnych generatorów biblioteka JGA posiada zbiór gotowych generatorów dla tworzenia wartości różnego typu: losowych, stałych lub wyliczanych. Wartości zwracane przez generatory nie muszą być liczbami.

Klasą reprezentującą funktor jednoargumentowy jest klasa `UnaryFunctor<T, R>`. Definiuje ona metodę `fn`, która przyjmuje parametr typu `T` i zwraca wartość typu `R`. Możemy na przykład utworzyć predykat, pisząc funktor zwracający wartość typu `Boolean`. Stwórzmy klasę typu `UnaryFunctor<Number, Boolean>`, która zwraca wartość `true` dla liczb parzystych:

```
public class EvenNumber extends UnaryFunctor<Number, Boolean> {
   public Boolean fn(Number x) {
      return (x.longValue() % 2) == 0;
   }
}
```

Na razie wygląda to tak, jakbyśmy dokładali sobie pracy bez wyraźnych korzyści. Jednakże zaletą takiego rozwiązania jest możliwość tworzenia nowych metod, które jako parametr przyjmą dowolne funktory. Kolejny listing pokazuje, jak to wygląda w praktyce:

```
public void removeMatches(List<Number> aList, UnaryFunctor<Number,Boolean>
functor) {
   for (Number num : aList) {
      if (functor.fn(num))
         aList.remove(num);
   }
}
```

Metoda ta usuwa wszystkie elementy z listy, dla których funktor typu `Number ->Boolean` zwraca `true` (a dokładnie Boolean.TRUE; w Javie 5 typy te są równoważne). Prześledźmy teraz dwa sposoby napisania kodu usuwającego liczby parzyste z listy:

```
List<Number> numbers = ...    // wypełniamy listę jakimiś liczbami
// pierwszy sposób
for (Number aNumber : numbers) {
   if (aNumber.longValue() % 2 == 0)
      numbers.remove(aNumber);
}
// drugi sposób
removeMatches(numbers, new EvenNumber());
```

Metoda taka jak `removeMatches` może być bardzo użyteczną częścią biblioteki. Pozwala nam ona zastosować kolejno kilka funktorów typu `UnaryFunctor <Number,Boolean>` na danej liście:

```
removeMatches(numbers, lessThan30000);
removeMatches(numbers, greaterThan10000000);
```

Możemy osiągnąć ten sam efekt, używając klasy `Iterables`, która dobrze nadaje się do wykorzystania w pętlach „for each" wprowadzonych w Javie 5. Na stronach JGA znajdziesz bardziej szczegółowe przykłady filtrowanych iteracji. Oto krótki przykład pętli, która jest wykonywana wyłącznie na parzystych elementach danej listy:

```
UnaryFunctor<Number,Boolean> even = new EvenNumber();
List<Number> numbers = ...;    // dowolne wypełnienie listy
for (Number aNumber : Iterables.filter(numbers, even)) {
   System.out.println(aNumber);
}
```

Klasa `Algorithms` obejmuje implementacje niektórych popularnych algorytmów, które wykorzystują funktory. Następny przykład używa dwóch z tych algorytmów: `forEach` stosuje funktor unarny dla każdego elementu z listy, a `removeAll` usuwa wszystkie elementy pasujące do predykatu:

```
import net.sf.jga.util.Algorithms;
List<String> aList = ...;    //wypełniamy listę
// usuwamy wszystkie wartości równe null
UnaryFunctor<Object,Boolean> isNull = new UnaryFunctor<Object,Boolean>() {
   public Boolean fn(Object o) {return o == null;}
};
Algorithms.removeAll(aList, isNull);
// przycinamy łańcuchy tekstowe
UnaryFunctor<String,String> trimmer = new UnaryFunctor<String,String>() {
   public String fn(String s) {return s.trim();}
};
Algorithms.forEach(aList, trimmer);
```

Jeżeli nie chcesz modyfikować oryginalnej listy, możesz utworzyć iterator przeglądający oryginalne wartości i zwracający zmodyfikowaną wartość każdego elementu lub ignorujący wartości `null`. Programowanie funkcjonalne jest bardzo użytecznym narzędziem programisty Javy. W następnej części użyjemy nieco bardziej zaawansowanych cech omawianego API.

Użycie funktorów złożonych

JGA

JAVA 5+

W poprzedniej części użyliśmy funktorów do filtrowania danych wejściowych dla pętli `for`, eliminując tym samym potrzebę filtrowania danych w samej pętli. JGA posiada funkcje pomocnicze służące tworzeniu złożonych funktorów i są one wbudowane automatycznie w każdy unarny i binarny funktor. Klasa `UnaryFunctor` posiada metodę `compose`, która zwraca nowy funktor będący złożeniem z wewnętrznym funktorem. Kiedy widzisz metodę `compose`, traktuj ją jak *funkcję złożoną*: `f.compose(g)` oznacza „funkcję f funkcji g". Innymi słowy, możesz utworzyć funktor `h` z dwóch funktorów `f` i `g`, tak że `h=f(g(x))`, za pomocą następującego kodu:

```
UnaryFunctor f,g;
UnaryFunctor h = f.compose(g);
```

Kiedy wywołamy metodę `fn` funktora `h`, zostanie faktycznie użyte złożenie dwóch funktorów. Klasa `BinaryFunctor` ma podobną metodę `compose` dla składania jej z innymi funktorami unarnymi i binarnymi. Możemy utworzyć w ten sposób łańcuch elementów o dowolnej długości. Możemy również przesłać wyjście z generatora do funkcji złożonej, używając pomocniczej klasy `Generate`. Spróbujmy użyć tej metody do stworzenia złożonego generatora wytwarzającego szereg logarytmów kwadratów każdej co trzydziestej liczby naturalnej, zaczynając od 99. Zacznijmy od napisania generatora zwracającego naturalne liczby ze skokiem równym 30, a potem będziemy dodawać dalszą część bazując już na nim:

```
Generator<Number> every30thFrom99 = new Generator<Number>() {
   long count = 99;
   public Number gen() {
      long result = count;
      count += 30;
      return result;
   }
};
UnaryFunctor<Number,Number> log = new UnaryFunctor<Number,Number>() {
   public Number fn(Number in) {
      double val = in.doubleValue();
      return Math.log(val);
   }
};
```

```
UnaryFunctor<Number,Number> square = new UnaryFunctor <Number,Number>() {
   public Number fn(Number in) {
      double val = in.doubleValue();
      return val*val;
   }
};
Generate<Number,Number> logOfSquareOfEvery30thFrom99 =
   new Generate(log.compose(square), every30thFrom99);
```

Zagnieżdżanie funkcji nie jest ograniczone do liczb. Funkcje złożone mogą działać na obiektach dowolnego typu (na przykład `String`, `Employee`, `Automobile`, `WebRowSet`). Widzimy teraz, dlaczego tworzenie funktorów na możliwie niskim poziomie jest ważne — możemy potem je łączyć i wykorzystywać w innym kontekście. Zanim napiszesz nowy funktor, sprawdź, czy nie dasz rady osiągnąć tego samego efektu, łącząc kilka istniejących. W chwili pisania tej książki omawiane API było wciąż w fazie „beta", więc pewne rzeczy mogą ulec jeszcze zmianie — dla pewności sprawdź więc dokumentację dostępną w internecie.

Bity dużego kalibru — BitVector z biblioteki Colt

COLT

JGA

Co mają wspólnego ze sobą matematyka, pistolety, konie i alkohol? Otóż okazuje się, że oprócz producenta broni, gatunku alkoholu i nazwy młodego konia „colt" jest również nazwą API matematyczno-fizycznego. API to zostało stworzone w tym samym miejscu, gdzie narodziła się Sieć — w CERN, laboratorium cząstek elementarnych w Szwajcarii. Strona CERN-u opisuje Colt jako „wydajne i użyteczne struktury danych oraz algorytmy dla przetwarzania danych, algebry liniowej, tablic wielowymiarowych, statystyk, histogramów, symulacji Monte Carlo, programowania równoległego i współbieżnego zarówno online, jak i offline".

W tej części rozdziału przyjrzymy się jednej z pomocniczych klas z biblioteki Colt, `BitVector`. Będziemy modelować funkcję logiki cyfrowej, tworząc funktor (jak to omawialiśmy wcześniej), który działa na wartościach typu `BitVector`. Jest to trochę inna sytuacja od tej, z którą mieliśmy do czynienia poprzednio, gdy używaliśmy predykatów logicznych. Obecnie będziemy starali się modelować funkcję mającą wiele bitów zarówno na wejściu, jak i na wyjściu. Nasza funkcja ma przyjmować uporządkowany zbiór bitów i wytwarzać inny uporządkowany zbiór bitów. Standardowa dystrybucja Javy dostarcza klasę `BitSet` w celu pracy ze zbiorem bitów. Choć ta klasa może być użyteczna w wielu aplikacjach, ma wiele wad, gdy zostanie użyta do modelowania funkcji logicznych. Po pierwsze, nie zapewnia ona stałego rozmiaru zbioru. Przykładowo, jeśli nasza funkcja miałaby przyjmować pięć bitów na wejściu i zwracać trzy na wyjściu, potrzeba by było przechowywać gdzieś w osobnych zmiennych rozmiary wektorów wejścia-

-wyjścia. Po drugie, `BitSet` nie posiada metody do przeliczania podzbioru bitów na reprezentację w postaci liczby całkowitej. Klasa `BitVector` z biblioteki Colt jest bardziej wydajna i lepiej przystosowana do modelowania funkcji tego typu. Zacznijmy od prostego przykładu demonstrującego niektóre z możliwości tej klasy:

```
BitVector vec1000 = new BitVector(1000);    // rozmiar = 1000 bitów
// początkowo wszystkie bity są ustawiane na false (0)
vec1000.set(378);                           // ustaw bit 378 na true (1)
System.out.println(vec1000.get(378));       // drukuje "true" na standardowym
                                               wyjściu
vec1000.replaceFromToWith(1, 40, true);     // ustawia bity od 1 do 40 na true
// zwróć bity z pozycji od 38 do 50 (włącznie)
BitVector portion = vec1000.partFromTo(38, 50);
//zwróć wartość typu long ciągu bitów 3-10
long longValue = portion.getLongFromTo(3,10);
```

Możemy użyć tych metod do symulacji bramek logicznych układów mikroelektronicznych stanowiących podstawowe cegiełki tworzące komputer. Przykładem takich niskopoziomowych bramek są bramki `AND`, `OR` i `NOT`. Do ogólnej reprezentacji bramki logicznej możemy wykorzystać instancję klasy `BitVector` jako wejście i wyjście funktora. Chociaż Colt posiada własne funktory ogólnego zastosowania, użyjemy tutaj API omówionego wcześniej, aby uniknąć nieporozumień. Funktor dla bramki `AND` mógłby wyglądać tak:

```
public class UnaryBitVectorAndFunction extends
UnaryFunctor<BitVector,BitVector> {
   public BitVector fn(BitVector in) {
      int oneBits = in.cardinality();          // ile bitów jest ustawionych na 1
      int size = in.size();                    // rozmiar wektora
      BitVector outVec = new BitVector(1);     // jednobitowe wyjście
      outVec.put(0, size == oneBits);          // AND = same jedynki
      return outVec;
   }
}
```

W podobny sposób jesteśmy w stanie utworzyć funkcję logiczną dowolnego typu. Zobaczmy jeszcze jeden przykład, tym razem z większą liczbą bitów na wyjściu. Dekoder 1-do-4 (demultiplekser) jest blokiem logicznym z trzema wejściami i czterema wyjściami. Demultiplekser przesyła sygnał z wejścia (D) na jedno z czterech wyjść (Q0, Q1, Q2, Q3). O tym, które jest to wyjście, decydują dwa bity sterujące (S0, S1). Tabela 5.1 pokazuje stan każdego z wyjść dla każdej z możliwych kombinacji stanów wejść. Taką tabelę nazywa się *tablicą prawdy*.

Tabela 5.1. *Tablica prawdy dla demultipleksera dwuwejściowego*

D	S1	S0	Q0	Q1	Q2	Q3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Pierwsze trzy kolumny przedstawiają stany na wejściu, pozostałe — na wyjściu. Selektory S0 i S1 decydują, które wyjście przedstawia stan wejścia, a pozostałe wyjścia są wyzerowane. Stwórzmy teraz model tego układu. Każdemu sygnałowi trzeba przypisać indeks w obiekcie `BitVector`. W przypadku wejść przyjmijmy, że indeks 0 odpowiada wejściu D, 1 — S0 i 2 — S1. Poniższy obiekt `UnaryFunctor` tworzy czterobitowy wektor zawierający wartości sygnałów wyjściowych:

```
public class QuadDemuxer extends UnaryFunctor<BitVector,BitVector> {
   public BitVector fn(BitVector in) {
      // czterobitowy wektor wyjścia, domyślnie wyzerowany
      BitVector outVec = new BitVector(4);
      // pobierz wartość wejścia D
      boolean data = in.get(0);
      if (data) {
         // bity sterujące zwrócone jako zmienna int
         int selector = (int) in.getLongFromTo(1,2);
         outVec.set(selector);
      }
      return outVec;
   }
}
```

W następnym podrozdziale rozszerzymy tę procedurę i utworzymy prostą implementację tablicy prawdy dla funkcji logicznej.

Tworzenie tablic prawdy za pomocą BitMatrix

COLT W poprzednim podrozdziale utworzyliśmy demultiplekser, specyficzny rodzaj funkcji logicznej posiadającej wiele bitów wejściowych i wyjściowych. Aby wyjaśnić jej działanie, użyliśmy tablicy prawdy przedstawiającej wyjścia odpowiadające wszystkim kombinacjom wejść. Tablica prawdy działa tak jak słownik typu

klucz-wartość dla kluczy binarnych. Pewnie zauważyłeś również, że nasza tablica prawdy wygląda jak macierz bitów. Możemy wykorzystać tę obserwację dla napisania uniwersalnego modelu tablicy prawdy. W podrozdziale utworzymy tablicę prawdy, używając klasy `BitMatrix`, dwuwymiarowego kuzyna znanej nam `BitVector`, również z biblioteki Colt.

Wejściowe kombinacje bitów w tabeli są wymienione w kolejności rosnącej: 000, 001, 010, 011… Ponieważ kolejność bitów wejściowych w tablicach prawdy jest zawsze taka sama, ta część tablicy jest zbędna — możemy zająć się wyłącznie wyjściami. Liczba rzędów i kolumn bezpośrednio wynika z liczby wejść i wyjść. W ostatnim przykładzie, mając 3 wejścia i 4 wyjścia, otrzymaliśmy 4 kolumny i 8 rzędów. Liczba kolumn jest równa liczbie wyjść. Liczba rzędów jest równa 2^n, gdzie n oznacza liczbę wejść, ponieważ musimy uwzględnić wszystkie możliwe kombinacje bitów na wejściu. Oczywiście pojedyncza tablica prawdy nie jest dobrym pomysłem dla bardzo dużej liczby wejść. Jednakże złożone systemy mogą być stworzone przez wzajemne powiązanie wielu prostszych komponentów, więc nie jest to poważny problem.

Jeśli tylko pamiętasz kolejność bitów wejściowych, możesz przekonwertować je na liczby całkowite i użyć jako indeksów dla rzędów zawierających bity wyjściowe. Zademonstrujmy to na przykładzie, tworząc macierz bitów typu `BitMatrix` dla tablicy prawdy z tabeli 5.1:

```
int inputSize = 3;
int rows = 1 << inputSize;          // 2**n rzędów, dla n bitów na wejściu
int outputSize = 4;
int columns = outputSize;
// tablica prawdy z wszystkimi bitami ustawionymi na 0
BitMatrix matrix = new BitMatrix(columns, rows);
// ustaw mapowanie wyjść dla przykładu z demultiplekserem
matrix.put(0, 4, true);             // kolumna 0, rząd 4 = true (1)
matrix.put(1, 5, true);
matrix.put(2, 6, true);
matrix.put(3, 7, true);

// podaj bity na wyjściu dla wartości wejściowej 101 (5)
boolean Q0 = matrix.get(0, 5);      // kolumna 0, rząd 5
boolean Q1 = matrix.get(1, 5);      // kolumna 1, rząd 5
boolean Q2 = matrix.get(2, 5);      // kolumna 2, rząd 5
boolean Q3 = matrix.get(3, 5);      // kolumna 3, rząd 5
```

Aby z tablicy prawdy wydobyć parę wartości wejścia-wyjścia, trzeba dokonać konwersji bitów wejściowych na liczbę całkowitą i użyć jej jako indeks rzędu zawierającego bity wyjściowe. Można czytać każdy bit z osobna jako wartość logiczną lub przekonwertować na typ `int` lub `long`, aby użyć wszystkich bitów naraz. Nasza tablica prawdy była łatwa do utworzenia, ponieważ zawiera niewiele jedynek, prawdopodobnie wolałbyś jednak użyć jakiejś ogólnej metody pomocniczej, która ustawia mapowanie wejść na wyjścia w pojedynczym wywołaniu. Następujące dwie metody implementują ogólny sposób modyfikacji i dostępu do tablicy prawdy:

```
public void store(int inputVal, long out) {
   int start = inputVal * outSize;
   int end = start + outSize - 1;   // włącznie
   matrix.toBitVector().putLongFromTo(out, start, end);
}

public long retrieve(int inputVal) {
   int start = inputVal * outSize;
   int end = start + outSize - 1;   // włącznie
   long out = matrix.toBitVector().getLongFromTo(start, end);
   return out;
}
```

Moglibyśmy również napisać wersje tych metod dla `BitVector`, używając kodu z poprzedniej części. Powyższe funkcje wykorzystują dostęp do wewnętrznych danych (`BitVector`) klasy `BitMatrix`. (Napisałem to w ten sposób, ponieważ `BitMatrix` nie posiada wygodnej metody dostępu do całych rzędów). Pamiętaj, że liczba bitów wejściowych powinna być utrzymana na niskim poziomie, ponieważ rozmiar tablicy prawdy rośnie wykładniczo. Nie ma tu sprawdzania wartości wejściowych i niewłaściwe dane spowodują wygenerowanie wyjątku. Bity wyjściowe, które nie są używane, są ignorowane — wynika to ze sposobu, w jaki `BitVector` konwertuje bity na wartości `long`. Pamiętaj też, że wartości wyjściowe i wejściowe są przetwarzane przy założeniu, że najmniej znaczący bit jest bitem zerowym. Wrócimy jeszcze do tablic prawdy, gdy utworzymy połączoną sieć bloków logicznych posiadających swoje własne tablice prawdy.

Dwa terafurlongi w dwa tygodnie — wielkości fizyczne z JScience

JScience

JScience jest kolejnym API typu open-source przeznaczonym do zastosowań w naukach ścisłych i matematyczno-przyrodniczych. Jednym z celów JScience, bardzo wzniosłym, jest utworzenie wspólnego API Javy dla wszystkich tych nauk. Do najciekawszych jego cech należy model jednostek fizycznych (na przykład masy, prędkości, temperatury, odległości). Za pomocą JScience możesz korzystać ze stałych fizycznych takich jak prędkość światła czy stała Plancka, nie przejmując się, jakie jednostki są użyte w nich wewnętrznie. Konwersja między różnymi systemami jest wyjątkowo łatwa, możesz również definiować swoje własne stałe. W tej części pokażemy, jak korzystać z wbudowanych stałych, tworzyć własne jednostki i używać klas opisujących wielkości.

W Stanach Zjednoczonych, poza kręgami naukowymi, większość ludzi używa jednostek spoza układu metrycznego: stopni Fahrenheita, stóp, funtów. To często powoduje różne niejasności, a nawet stało się przyczyną katastrofy jednego bezzałogowego statku kosmicznego! Być może nie potrafimy rozwiązać definitywnie tego problemu, ale na szczęście programując w Javie, możemy definiować

wielkości, nie przejmując się jednostkami, jakie za nimi stoją. W JScience podstawowe wielkości fizyczne mają własne klasy dziedziczące po wspólnej klasie bazowej `Quantity` (Ilość). Oto przykłady kilku z nich, razem z ich jednostkami w układzie SI:

- długość (w metrach),
- czas (w sekundach),
- masa (w kilogramach),
- temperatura (w kelwinach).

Klasy typu `Quantity` znajdują się w pakiecie `javax.quantities`. Klasy te zawierają informacje o wielkościach, które mierzymy, i są dużo bardziej precyzyjne (czyli zapewniają mniejsze prawdopodobieństwo błędów) niż zwykłe wartości typu `double` do reprezentacji wartości numerycznych. W dowolnej aplikacji mierzącej zwykłe wielkości, takie jak długość, możesz użyć jednej ze standardowych klas JScience typu `Quantity`. Każda z tych klas przechowuje powiązaną z nią jednostkę wielkości i dzięki temu może dokonywać automatycznych konwersji. Jeśli piszesz aplikację zajmującą się komputerami, możesz napisać taką klasę do reprezentacji możliwych konfiguracji:

```
public class ComputerConfig {
   double length, width, height;
   double mass;
   int ram, rom, network;
}
```

W tej klasie jednostki fizyczne nie są przyporządkowane do zmiennych, brak też wewnętrznej kontroli typów wielkości fizycznych. Na przykład nic nie stoi na przeszkodzie, aby napisać tak:

```
length = mass;
```

W fizyce włożono ogromny wysiłek w stworzenie teorii unifikacji, ale mimo to nie przypuszczam, żeby posunięto się aż *tak* daleko! Możemy zmienić klasę `ComputerConfig` tak, aby używała ona rzeczywistych wielkości fizycznych:

```
import javax.quantities.*;
public class ComputerConfig {
   Length length, width, height;
   Mass mass;
   DataAmount ram, rom;
   DataRate network;
}
```

Teraz nasz program wie, że mamy tu trzy wielkości określające długość, jedną opisującą masę, dwie określające pojemność pamięci i jedną — prędkość przesyłu danych. Ma to jeszcze inną zaletę: możemy podawać i otrzymywać wartości wyrażone w dowolnych jednostkach, wciąż mając je przechowane w jednostkach układu SI. Zobaczmy, jak to wygląda w praktyce — przypiszmy zmiennej wagę w funtach i pobierzmy ją wyrażoną w kilogramach:

```
Measure<Mass> mass = Measure.valueOf(20, NonSI.POUND);
System.out.println(mass.to(SI.KILOGRAM));
```

Jak się pewnie domyślasz, próba użycia jednostki niezgodnej z typem zmiennej spowoduje zgłoszenie wyjątku. Większość jednostek układu metrycznego znajdziesz w klasie `SI`, a pozostałe w klasie `NonSI`. Jeśli potrzebujesz jednostek, które nie zostały dodane jeszcze do JScience, możesz zawsze stworzyć własne, bazując na tych już istniejących.

Jedna z bardziej tajemniczych jednostek angielskiego układu miar nazywa się *furlong* i równa jest jednej ósmej mili — lub inaczej 220 jardom[1]. Jednostki tej nie znajdziemy w JScience, ale możemy łatwo ją utworzyć. Poniższy kod tworzy taką jednostkę, jak również dodaje dla niej alias, aby mogła być użyta w dowolnym miejscu w aplikacji:

```
Unit<Length> furlong = NonSI.MILE.times(0.125);
UnitFormat.getStandardInstance().alias(furlong, "furlong");
UnitFormat.getStandardInstance().label(furlong, "furlong");
Quantity fiveFurlong = Measure.valueOf("5 furlong");
```

Tworzenie aliasów dla nowych jednostek pozwala używać ich później w opisach wielkości fizycznych. Etykieta (ang. *label*) jest zaś stosowana przy wyświetlaniu wartości. Wielkości i jednostki mogą być mnożone i dzielone w celu otrzymania jednostek takich wielkości fizycznych jak przyspieszenie ($m*s^{-2}$). W kolejnym przykładzie wyprowadzimy nową jednostkę i użyjemy jej do wyświetlenia prędkości światła (c):

```
Measure<Velocity> c = Measure.valueOf(299792458, SI.METER_PER_SECOND);[2]
//słowo "fortnight" oznacza dwa tygodnie (14 dni)
Unit<Duration> fortnight = NonSI.DAY.times(14);
UnitFormat.getStandardInstance().alias(fortnight, "fortnight");
UnitFormat.getStandardInstance().label(fortnight, "fortnight");
Unit<Velocity> furlongperfortnight = (Unit<Velocity>)
furlong.divide(fortnight);
System.out.println(c.to(furlongperfortnight));
```

[1] Około 200 metrów — *przyp. tłum.*

[2] Co prawda w Science istnieje klasa `org.jscience.physics.measures.Constants`, w której zdefiniowano prędkość światła *c*, ale klasa ta w obecnej wersji (3.1.6) niestety nie działa poprawnie.

Nasza nowa jednostka jest jednostką prędkości, gdyż jest definiowana jako długość przez czas. Po uruchomieniu tego programu zobaczymy, że prędkość światła w próżni wynosi 1 802 617 499 785 253 furlongi na dwa tygodnie, czyli trochę mniej niż 2 terafurlongi na dwa tygodnie.

Krnąbrne ułamki — arytmetyka dowolnej precyzji

JScience

Dwa razy dwa równa się 3,9999999998, jeśli wierzyć wynikom obliczeń przeprowadzonych na zmiennych typu `double`. Większość programistów Javy miała okazję używać typów `double` i `float`. Być może miałeś do czynienia również z niektórymi metodami z klasy `java.lang.Math`. Klasa ta należy to najbardziej rdzennych bibliotek Javy i zawiera metody dla operacji na funkcjach, takich jak wykładnicze, logarytmiczne, trygonometryczne. Precyzja typu `double` wystarcza dla większości zastosowań, jednak dla naukowych aplikacji jej 11-bitowa cecha i 52-bitowa mantysa (oparte na typie zmiennoprzecinkowym podwójnej precyzji według IEEE 754) może czasem spowodować drobne błędy i szybko urosnąć do dużych problemów. Jest to szczególnie istotne w obliczeniach iteracyjnych, gdy rezultat poprzedniej iteracji jest daną wejściową kolejnej.

Aby rozwiązać problem błędów zaokrąglenia w zastosowaniach matematycznych, Java posiada dwie klasy dla operacji o dowolnej precyzji: `BigDecimal` i `BigInteger`. `BigInteger` jest świetna do pracy z wyjątkowo dużymi liczbami. (Ależ tak, zdaję sobie sprawę, że nazwa dokładnie na to wskazuje!) Wielkość tych liczb jest ograniczona jedynie przez dostępną pamięć oraz prędkość procesora. Liczba taka jak 9^{700} jest zbyt duża, aby przechowywać ją w zmiennej `double` lub `long`, lecz nadaje się idealnie do zapamiętania w zmiennej typu `BigInteger`. Podobnie `BigDecimal` może precyzyjnie przechować liczby takie jak 0,0123456 789876543212345678901234 56, z którą nie poradzi sobie liczba o mantysie 52-bitowej. Następujący krótki przykład używa klas `BigInteger` oraz `BigDecimal` do reprezentacji tych liczb:

```
BigInteger nine = new BigInteger("9");
BigInteger nineToSevenHunredth = nine.pow(700);
BigDecimal exactNumber = new BigDecimal("0.
012345678987654321234567890123456");
```

`BigDecimal` może reprezentować precyzyjnie dowolną liczbę, która ma skończone rozwinięcie dziesiętne. Jest to spełnione dla ułamków, które w mianowniku posiadają wielokrotności liczb 2 i 5, np. 1/2, 15/4, 127/20, lecz nie jest dla liczb takich jak 1/3 (0,333333…) lub 5/7. Może to powodować kumulację błędów zaokrąglenia w bardziej złożonych obliczeniach. W rzeczywistości wszystkie metody klasy `BigDecimal` związane z dzieleniem wymagają parametru skalującego

określającego liczbę cyfr dziesiętnych, które powinny być zachowane w wyniku. Rozważmy następujący przykład, w którym wyliczany jest ułamek 1/3 z dokładnością do 15 miejsc po przecinku (i zaokrąglany w razie potrzeby):

```
BigDecimal one = new BigDecimal("1");
BigDecimal three = new BigDecimal("3");
BigDecimal third = one.divide(three, 15, BigDecimal.ROUND_HALF_UP);
```

Przy wykonywaniu obliczeń arytmetycznych o dowolnie wysokiej precyzji najlepiej jest przechowywać licznik i mianownik w osobnych zmiennych. Ponieważ w podstawowych bibliotekach Javy nie ma obiektów dla ułamków zwykłych o dowolnej precyzji, być może pomyślałeś o napisaniu własnej klasy. Wielu ludzi stworzyło klasy tego typu:

```
public class HugeFraction {
   private BigInteger numerator, denominator;
   // metody dla operacji na ułamkach
   public HugeFraction divide(HugeFraction other) {
      // policz rezultat
      return result;
   }
}
```

W bibliotece JScience istnieje kilka klas służących do przeprowadzania operacji o dowolnie wysokiej precyzji. Pierwsza z nich to `LargeInteger`, podobna do `BigInteger` ze standardowej dystrybucji Javy. `LargeInteger` jest zoptymalizowana ze względu na prędkość i wydajność w czasie rzeczywistym i implementuje interfejs `Ring` (struktura algebraiczna) używany w teorii liczb i obliczeniach macierzowych. Posiada również związany ze sobą format XML. Klasa `Rational`[3] bazuje na niej w celu umożliwienia reprezentacji ułamków *a/b* o nieskończonej precyzji, dla a i b będących liczbami całkowitymi różnymi od zera. Klasa `Rational` jest niezmienna — wszystkie jej metody zwracają wynik, zamiast zmieniać oryginalny obiekt. Działa to dokładnie tak, jakbyś mógł się spodziewać:

```
Rational oneThird = Rational.valueOf("1/3");
Rational nine87654321 = Rational.valueOf("987654321/1");
Rational msixteen = Rational.valueOf("-16/1");
Rational msixteenOver987654321 = msixteen.divide(nine87654321);
Rational aNumber = oneThird.times (msixteenOver987654321);
```

[3] Ang. *rational number* = liczba wymierna — *przyp. tłum.*

Tak długo, jak pozostaniesz w dziedzinie obiektów `Rational`, nie pojawią się żadne błędy zaokrąglania (przy dodawaniu, odejmowaniu, mnożeniu, dzieleniu, potęgach o wykładnikach całkowitych). Inną klasą, której na pewno nie chcesz przegapić, jest `Real`[4]. Reprezentuje ona liczbę rzeczywistą dowolnie wysokiej precyzji, o zagwarantowanej niepewności. JScience API posiada również klasę `Complex`[5], lecz nie jest ona liczbą dowolnej precyzji, gdyż części rzeczywista i urojona są przechowywane w zmiennych typu `double`. W następnej części przyjrzymy się funkcjom algebraicznym i wielomianowym.

Funkcje algebraiczne w JScience

JScience

We wcześniejszych częściach tego rozdziału pokazaliśmy korzyści płynące z możliwości manipulowania funkcjami, tak jakby były one obiektami, przez przekazywanie ich jako parametrów do metod. W podrozdziale spróbujemy spojrzeć na funkcje z matematycznego punktu widzenia. JScience, Colt, JGA (omawiane wcześniej) — wszystkie z nich zawierają swoje własne implementacje funktorów i każda z nich ma swoje zalety. Wersja z JGA jest bardzo dobra w zastosowaniach ogólnego typu z powodu swojej prostoty i ogólności. Colt zawiera więcej wbudowanych funktorów (patrz klasa `Functions`), lecz nie zapewnia obliczeń o dowolnie wysokiej precyzji. W niniejszym podrozdziale omawiamy JScience, ponieważ API to posiada ogólny zrąb dla operacji algebraicznych i wielomianowych możliwy do zastosowania z dowolnymi obiektami implementującymi interfejs `Ring` (takimi jak `Real` bądź `Rational` lub Twoimi własnymi klasami będącymi implementacją *pierścieni*).

Obiekt typu `Ring` posiada metody mnożące i dodające oraz operację odwrotną dla każdej z tych metod. Klasa `Polynomial` (Wielomian) oznacza „wyrażenie matematyczne zawierające sumę potęg jednej lub wielu zmiennych przemnożonych przez współczynniki" (cytat z dokumentacji JScience). Możesz utworzyć wielomiany, które współpracują z dowolną klasą typu `Ring`. Na nasze potrzeby zdefiniujemy wielomian zmiennych rzeczywistych (`Rational`). Zacznijmy od stałego wielomianu. Klasa `org.jscience.mathematics.functions.Constant` jest podklasą `Polynomial` reprezentującą wielomian stopnia zerowego. Stwórzmy jeden egzemplarz za pomocą metody `valueOf`:

```
Constant<Rational> sixty = Constant.valueOf(Rational.valueOf("60/1"));
```

Zastosujmy teraz tą metodę do utworzenia wielomianu $(7/15)\ x^5 + 9xy + 60$. Musimy na początku utworzyć każdy z elementów i przemnożyć przez odpowiedni współczynnik. Następnie możemy dodać elementy do siebie, aby utworzyć wielomian. Poniższy kod tworzy wielomian, przypisuje wartości x oraz y i na koniec wyświetla wynik:

[4] Ang. *real number* = liczba rzeczywista — *przyp. tłum.*

[5] Ang. *complex number* = liczba zespolona — *przyp. tłum.*

```
Variable.Local<Rational> x = new Variable.Local<Rational>("x");
Variable.Local<Rational> y = new Variable.Local<Rational>("y");
Polynomial<Rational> xpoly = Polynomial.valueOf(Rational.ONE, x);
Polynomial<Rational> ypoly = Polynomial.valueOf(Rational.ONE, y);
Rational nine = Rational.valueOf("9/1");
Rational sixty = Rational.valueOf("60/1");
Rational seven15ths = Rational.valueOf("7/15");

Polynomial seven15X5 = Polynomial.valueOf(seven15ths, Term.valueOf(x, 5));
Polynomial nineXY = (Polynomial)
Constant.valueOf(nine).times(xpoly).times(ypoly);
Polynomial poly = (Polynomial)
seven15X5.plus(nineXY).plus(Constant.valueOf(sixty));
x.set(Rational.valueOf("5/7"));
y.set(Rational.ONE);
System.out.println(poly);
System.out.println(poly.evaluate());
```

Wynikiem działania tego kodu dla x = 5/7, y = 1 jest rezultat:

```
[7/15]x^5 + [9/1]xy + [60/1]
479110/7203
```

Możesz również różniczkować i całkować wielomiany. JScience ma wiele innych pożytecznych cech. Opis szczegółów znajdziesz w dokumentacji do API.

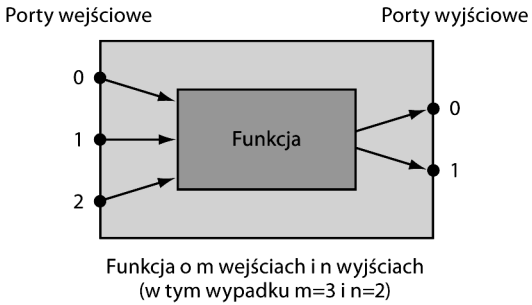
Łączenie tablic prawdy za pomocą portów

COLT

JGA

Na początku dwudziestego wieku filozof Ludwig Wittgenstein napisał „Traktat logiczno-filozoficzny", w którym określił koncepcję **funkcji prawdy** będącej abstrakcją wyższego stopnia w logice zdaniowej. Jego funkcje prawdy działają tak jak znane nam tablice prawdy, lecz łączą one inne zdania zawierające inne funkcje i dają w wyniku jedynie pojedynczą wartość typu logicznego. W naszej tablicy prawdy z podrozdziału „Tworzenie tablic prawdy za pomocą BitMatrix" mieliśmy pewną liczbę wejść i wyjść, przy czym nie łączyliśmy ich z innymi funkcjami. Jeśli budowałbyś symulator logiczny, musiałbyś zapewne połączyć wiele jego części w jeden spójny system. Aby móc tworzyć i łączyć bloki logiczne ze sobą, musimy napisać kod do tego służący.

Możemy połączyć dowolne komponenty na rysunku w jeden schemat, łącząc je liniami, tak jak łączy się elementy elektroniczne na schematach. Nie chodzi tu jednak tylko o połączenie elementów bezpośrednio ze sobą, lecz o dokładne przyporządkowanie wyjść z jednego komponentu do wejść innych komponentów. Innymi słowy, na rysunku 5.1 nie łączymy ze sobą elementów, lecz ich porty.



Rysunek 5.1. Standardowy element funkcyjny

Możesz wyobrazić sobie porty jako nóżki chipów elektronicznych. Łączenie portów pozwala na dokładne określenie, które wyjście jest połączone z którym wejściem. Spróbujmy użyć tego podejścia do utworzenia klasy `Component`, której będzie można użyć w większym systemie.

Nasza klasa `Component` jest uogólnieniem „funkcji z portami", która to może być zaaplikowana do dowolnej funkcji posiadającej zbiór wejść i wyjść. Użyjemy tego podejścia do napisania rozszerzenia dla metod `store` i `retrieve` z tabeli prawdy utworzonej przez nas wcześniej. Dodamy porty wejścia-wyjścia i kilka metod dostępu do klasy. Ponieważ port wejściowy może być podłączony jedynie do portu wyjściowego, utwórzmy na początek osobny interfejs dla każdego typu (oraz klasę implementującą obydwa z nich):

```
public interface Port {
   public Component getParent();
}
public interface InputPort extends Port {
   public void setValue(Object value);
}
public interface OutputPort extends Port {
   public Object getValue();
}
public class PortImpl implements InputPort, OutputPort {
   private Component parent;
   private Object value;
   public PortImpl(Component parent) {
      this.parent = parent;
   }
   public Component getParent() { return parent; }
   public Object getValue() { return value; }
   public void setValue(Object value) { this.value = value; }
}
```

`InputPort` oraz `OutputPort` to interfejsy używane zewnętrznie w stosunku do komponentu. Wysyłanie danych do portu wejściowego wymaga wywołania metody ustawiającej zmienną `value`, a czytanie z portu wyjściowego oznacza wywołanie metody odczytującej wartość zmiennej `value`. Komponent określa rozmiar wejścia i wyjścia oraz definiuje metodę `Object[] -> Object[]`, która oblicza wartość wyjścia dla zadanych wartości na wejściu. Komponent jest zaimplementowany jako interfejs w celu zwiększenia zakresu możliwych zastosowań:

```
public interface Component {
    // zwraca ilość portów wejściowych
    public int getInputSize();
    // zwraca ilość portów wyjściowych
    public int getOutputSize();
    // zwraca port wejściowy o danym numerze identyfikacyjnym
    public InputPort getInputPort(int index);
    // zwraca port wejściowy o danym numerze identyfikacyjnym
    public OutputPort getOutputPort(int index);
    // zasadnicza metoda komponentu wykonująca funkcję outputs = f(intputs)
    public void process();
}
```

W tej chwili mamy już ogólny szkielet dla łączenia komponentów posiadających porty wejścia-wyjścia. Założyłem tutaj, że komponent posiada jedną funkcję wykonującą całą pracę, pobierającą `Object[]` jako parametr i zwracającą również `Object[]`. Jeśli chcesz, możesz uczynić tę funkcję na tyle elastyczną, że będzie ona potrafiła zwracać pojedynczy obiekt na pierwszy port wyjściowy i `null` na pozostałe. Na stronie internetowej książki znajdziesz implementację klasy `Component` wzbogaconą o możliwość użycia typów generycznych Javy 5 (ang. *generics*). Zewnętrzny proces kontroluje komponenty i zarządza połączeniami między nimi — zaimplementujemy go później, korzystając z API grafów. Jednakże wcześniej musimy zmienić nasz obiekt w jednostkę przetwarzającą bity korzystającą z tablicy prawdy.

Możemy napisać jednoargumentowy funktor (podtyp `UnaryFunctor`), który przetwarza tablicę obiektów na bity, stosując pewne proste reguły konwersji. Funkcja powinna być na tyle elastyczna, żeby interpretować każdy obiekt w tablicy jako bit w ten sam sposób (`Boolean`, niezero, nie-`null`). Załóżmy, że metoda `arrayToBits` konwertuje tablicę na bity. Przekonwertowane bity są interpretowane zgodnie z tablicą prawdy i dają bity wyjściowe. Te z kolei są zwracane jako tablica `Boolean[]`, tak aby komponent mógł przesłać rezultat na swoje porty wyjściowe. Załóżmy, że metoda `bitsToArray` potrafi tego dokonać. Poniższy przykład jest uproszczoną wersją faktycznej funkcji, która mogłaby być użyta z naszą klasą `Component`:

```
public class UnaryTruthTableFunction extends
UnaryFunctor<Boolean[],Boolean[]> {
   public Boolean[] fn(Boolean[] in) {
      // zmień obiekty na bity (w jakiś sposób)
      int convertedInput = arrayToBits(in);
      // znajdź wyjście w tabeli prawdy
      // (opis w podrozdziale "Tworzenie tablic prawdy za pomocą BitMatrix")
      long result = retrieve(convertedInput);
      //zmień wyjściowe bity na Boolean[] (w jakiś sposób)
      return bitsToArray(result);
   }
}
```

Dokładniejsze rozwiązanie jest dostępne na stronie internetowej książki. W bieżącej implementacji rezultaty muszą być obliczane w sposób „do przodu" (tzn. tylko w kierunku od wejść do wyjść). Jeśli komponenty są powiązane ze sobą cyklicznie, potrzebny będzie jakiś rodzaj synchronizacji. W kolejnym podrozdziale wprowadzimy API grafów i użyjemy go do połączenia portów między komponentami oraz uruchomienia symulacji.

Łączenie za pomocą JGraphT

JGraphT

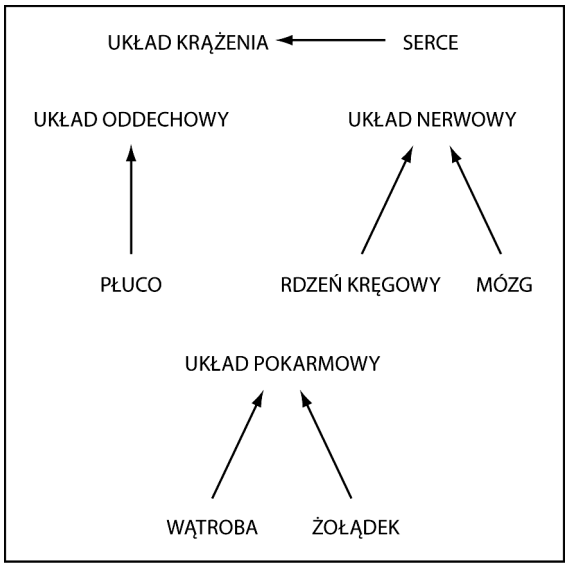
JAVA 5+

W rozdziale 4. odkrywaliśmy „Sieć semantyczną". Standard RDF, który omawialiśmy, modeluje te sieci jako **etykietowane grafy skierowane**. Grafy oznaczają tu sieć węzłów lub wierzchołków. W teorii grafów etykietowany graf skierowany oznacza, że każda **krawędź** między dwoma **wierzchołkami** posiada opis i kierunek. Grafy Jena i RDF użyte przez nas wcześniej są implementacją grafów oznaczonych do specyficznych zastosowań, inne aplikacje mogą jednak potrzebować innych typów grafów do modelowania oraz wykonywania operacji na nich. W tym celu możesz wykorzystać JGraphT, łatwą w użyciu bibliotekę do pracy z grafami różnego rodzaju. Koncentruje się ona na samym modelu grafu, jego połączeń oraz wykonywaniu operacji na nim — a nie na wizualizacji i wyświetlaniu go. W rozdziale 6. omówimy inne API, którego głównym celem jest wizualizacja grafów. Póki co jednak będziemy budować po prostu modele grafów.

Węzły w JGraphT mogą być dowolnymi obiektami Javy. Model grafu opisuje, w jaki sposób obiekty połączone są ze sobą. Używając tego modelu, możesz zajmować się połączeniami między obiektami, niezależnie od samych obiektów. Jest to bardzo podobne do założeń modelu Model-View-Controller (MVC, model-widok-kontroler) użytego w Swingu oraz frameworkach sieciowych. Teoria grafów jest użyteczna w symulacji i analizie wielu skomplikowanych systemów, takich jak sieci komputerowe, obwody cyfrowe, ruch na autostradzie czy przesył danych.

Tworzenie grafu w JGraphT jest proste. Na początku tworzysz instancję wymaganego typu grafu. Następnie wywołujesz metodę `addVertex`, aby dodać nowy obiekt Javy jako wierzchołek. Kiedy obiekt jest już częścią grafu, możesz wywołać

metody służące połączeniu go z innymi wierzchołkami. Poniżej mamy przykład modelu połączeń niektórych organów i systemów w ciele ludzkim. Rysunek 5.2 pokazuje zależności miedzy elementami grafu.



Rysunek 5.2. Organy i systemy w ciele ludzkim

Użyjemy konstrukcji `enum` z Javy 5 (zobacz rozdział 1.) do reprezentacji organów i układów zawartych w grafie. Aby przechowywać zależności między tymi częściami, możemy stworzyć graf nieskierowany:

```
import org.jgrapht.graph.DefaultEdge;
import org.jgrapht.graph.SimpleGraph;

enum Organs  {HEART, LUNG, LIVER, STOMACH, BRAIN, SPINAL_CORD};
enum Systems {CIRCULATORY, DIGESTIVE, NERVOUS, RESPIRATORY};

SimpleGraph<Enum, DefaultEdge> graph =
    new SimpleGraph<Enum, DefaultEdge>(DefaultEdge.class);
graph.addVertex(Organs.HEART);          // serce
graph.addVertex(Organs.LUNG);           // płuco
graph.addVertex(Organs.BRAIN);          // mózg
graph.addVertex(Organs.STOMACH);        // żołądek
graph.addVertex(Organs.LIVER);          // wątroba
graph.addVertex(Organs.SPINAL_CORD);    // rdzeń kręgowy
graph.addVertex(Systems.CIRCULATORY);   // układ krążenia
graph.addVertex(Systems.NERVOUS);       // układ nerwowy
graph.addVertex(Systems.DIGESTIVE);     // układ pokarmowy
graph.addVertex(Systems.RESPIRATORY);   // układ oddechowy
```

```
graph.addEdge(Organs.HEART, Systems.CIRCULATORY);
graph.addEdge(Organs.LUNG, Systems.RESPIRATORY);
graph.addEdge(Organs.BRAIN, Systems.NERVOUS);
graph.addEdge(Organs.SPINAL_CORD, Systems.NERVOUS);
graph.addEdge(Organs.STOMACH, Systems.DIGESTIVE);
graph.addEdge(Organs.LIVER, Systems.DIGESTIVE);
```

Zwróć uwagę, że każdy z wierzchołków musi być dodany do grafu, zanim dołączysz go do krawędzi, w przeciwnym razie otrzymasz wyjątek. Ten kod nie wyświetla niczego, tworzy jedynie wewnętrzną sieć połączeń. Dokładniej tworzy on listę sąsiadów dla każdego obiektu. W tym szczególnym przypadku wszystkie krawędzie są traktowane jednakowo. Aby krawędzie stały się rozróżnialne, możesz użyć krawędzi etykietowanych.

Wywołując odpowiednie metody na grafie, możesz znaleźć sąsiadów danych obiektów. Znajdźmy więc krawędzie dla jakiegoś wierzchołka z naszego przykładu i wyświetlmy, co jest po drugiej stronie danej krawędzi. Węzły połączone bezpośrednio z węzłem `DIGESTIVE` możemy znaleźć za pomocą metody:

```
import org.jgrapht.Graphs;
import org.jgrapht.graph.DefaultEdge;

Set<DefaultEdge> digestiveLinks = graph.edgesOf(Systems.DIGESTIVE);
for (DefaultEdge anEdge : digestiveLinks) {

   Enum opposite = Graphs.getOppositeVertex(graph, anEdge,
   Systems.DIGESTIVE);
   System.out.println(opposite);
}
```

Technika, której tutaj użyliśmy, polega na uzyskaniu listy krawędzi, następnie dla każdej z nich trzeba znaleźć wierzchołek, który nie jest węzłem `DIGESTIVE` (czyli znaleźć przeciwny wierzchołek). `DefaultEdge` (krawędź) to bazowa klasa, po której dziedziczą wszystkie krawędzie i która łączy wierzchołek źródłowy z docelowym. Możesz użyć jednej ze standardowych implementacji dostępnych wraz z biblioteką JGraphT lub napisać własną podklasę, aby spełniała specyficzne zadania.

JGraphT posiada implementacje innych operacji, które można wykonywać na grafach. Więcej informacji znajdziesz w dokumentacji do API. Będziemy korzystać jeszcze z JGraphT w następnej części rozdziału. W rozdziale 6. użyjemy innego API do wizualizacji grafów.

Łączenie ogólnych jednostek obliczeniowych

JGraphT

JAVA 5+

W części poświęconej łączeniu węzłów stworzyliśmy „ogólną jednostkę obliczeniową" z portami wejścia i wyjścia, a w części poświęconej grafom poznaliśmy sposób na łączenie dowolnych obiektów Javy w strukturę grafów. W tej części połączymy wejścia i wyjścia komponentów, używając JGraphT. Tym sposobem będziemy mogli utrzymywać połączenia portów niezależnie od implementacji komponentów. W rzeczywistości nie dbamy tutaj w ogóle o to, co komponenty robią, lub nawet czy przetwarzają one wartości logiczne. Każdy komponent ma metodę `process`, która pobiera dane z portów wejściowych, przetwarza je i zwraca rezultaty na porty wyjściowe. Możemy wywołać metodę `process` dla każdego z komponentów i wysłać wynik na porty wyjściowe. Jeśli użyliśmy grafu skierowanego, możemy przesłać dane z wyjść na wejścia kolejnych stopni poprzez iterację po wszystkich krawędziach. Dla każdej krawędzi wywołujemy metodę `getValue` na wierzchołku źródłowym (`OutputPort`) i ustawiamy wartość wyjściową na wierzchołku docelowym (`InputPort`) za pomocą metody `setValue`. Możemy teraz napisać klasę, która zarządza wierzchołkami używając API grafów.

```
public class MetaComponentSimple {
   private ListenableDirectedGraph<Object, DefaultEdge> graph;

   public MetaComponentSimple() {
      graph = new ListenableDirectedGraph(DefaultEdge.class);
   }

   public void connect(OutputPort out, InputPort in) {
      Component source = out.getParent();
      Component target = in.getParent();
      //dodaj nadrzędne komponenty do grafu
      if (!graph.containsVertex(source)) {
         graph.addVertex(source);
      }
      if (!graph.containsVertex(target)) {
         graph.addVertex(target);
      }
      // dodaj porty do grafu
      if (!graph.containsVertex(in)) {
         graph.addVertex(in);
      }
      if (!graph.containsVertex(out)) {
         graph.addVertex(out);
      }
      // dodaj krawędź od komponentu-źródła do portu wyjściowego
      graph.addEdge(source, out);
      // dodaj krawędź od portu wyjściowego do wejściowego
      graph.addEdge(out, in);
      // dodaj krawędź od portu wejściowego do komponentu-celu
      graph.addEdge(in, target);
   }
```

```
    public void process() {
        processSubComponents();
        propagateSignals();
    }

    private void propagateSignals() {
        for (DefaultEdge edge : graph.edgeSet()) {

            Object source = graph.getEdgeSource(edge);
            Object target = graph.getEdgeTarget(edge);
            if (source instanceof OutputPort) {
                OutputPort out = (OutputPort) source;
                InputPort in = (InputPort) target;
                in.setValue(out.getValue());
            }
        }
    }

    private void processSubComponents() {
        for (Object item : graph.vertexSet()) {
            if (item instanceof Component) {
                ((Component) item).process();
            }
        }
    }
}
```

Aby użyć tej klasy, musisz najpierw ustanowić połączenia pomiędzy portami, wywołując metodę `connect`.

Ta metoda dodaje nadrzędny komponent każdego portu do grafu, tak aby później dany komponent mógł wywołać metodę `process` swoich subkomponentów. Klasa zarządzająca ma własną metodę `process`, która najpierw wywołuje analogiczne metody wszystkich komponentów, a następnie przekazuje wyniki ich działań na wejścia kolejnego stopnia. Wybór tej samej nazwy dla tej metody nie jest przypadkiem. Jeśli chciałbyś zaimplementować inne metody interfejsu `Component`, mógłbyś użyć tej klasy do zbudowania komponentu złożonego z innych komponentów. Na stronie internetowej książki znajdziesz bardziej kompletny przykład. Poniżej mamy kod, który korzysta z naszej nowej klasy do implementacji funkcji `y=AND(OR(a,b),OR(c,d))`.

```
MetaComponentSimple manager = new MetaComponentSimple();
// Załóżmy, że stworzyliśmy komponent – bramkę typu AND,
// używając technik opisanych wcześniej. Posiada ona dwa wejścia.
Component and = createAndGateComponent(2);
OutputPort y = and.getOutputPort(0);
// użyjemy pary bramek OR, każda z nich jest dwuwejściowa
Component or1 = createOrGateComponent(2);
InputPort a = or1.getInputPort(0);
```

```
InputPort b = or1.getInputPort(1);
Component or2 = createOrGateComponent(2);
InputPort c = or2.getInputPort(0);
InputPort d = or2.getInputPort(1);
manager.connect(or1.getOutputPort(0), and.getInputPort(0));
manager.connect(or2.getOutputPort(0), and.getInputPort(1));
// ustaw wartości wejściowe
a.setValue(true);
b.setValue(false);
c.setValue(false);
d.setValue(false);
manager.process();
// potrzebujemy drugiego wywołania, gdyż mamy komponent dwustopniowy
manager.process();
System.out.println(y);    // wynik: false
```

Zauważ, że przetwarzanie sygnału może zajmować więcej niż jeden cykl, zanim osiągnie on porty wyjściowe. Tak też jest dla rzeczywistych obwodów, ponieważ każdy komponent wprowadza opóźnienie, choć jest ono tak krótkie, że go nie zauważamy. Dyskusję zależności czasowych zostawiamy do rozdziału 7. W następnej części będziemy zajmować się innymi sieciami, przypominającymi sieć nerwów w mózgu, zwanymi sieciami neuronowymi.

Budowanie sieci neuronowych z Joone

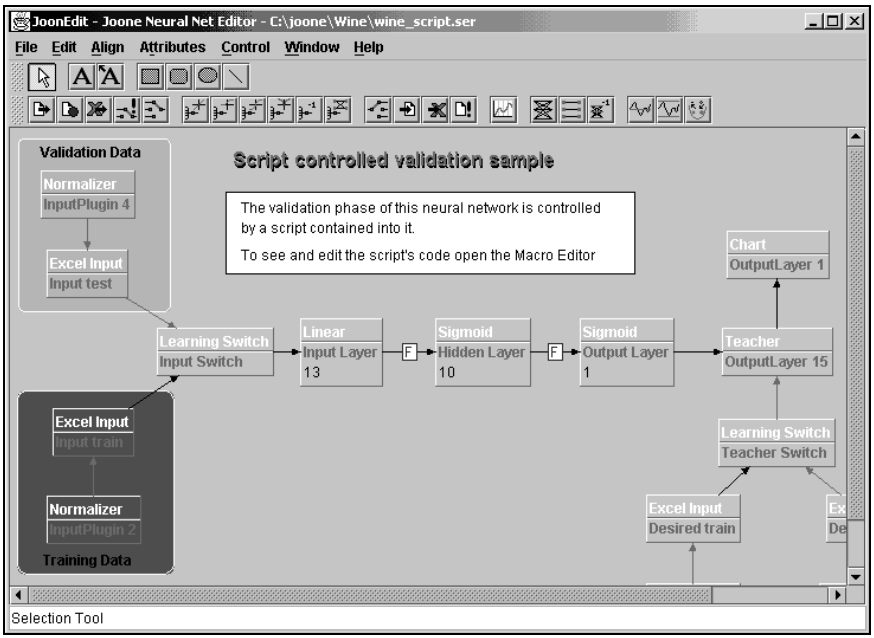
JOONE

Czy zastanawiałeś się kiedyś, jak można zbudować mózg? Cóż, w Javie jest to proste i wcale nie musisz się czuć doktorem Frankensteinem. W terminologii informatycznej **sieci neuronowe** to grupa prostych komórek obliczeniowych ściśle powiązanych ze sobą i tworzących jako całość system do przetwarzania danych. Niektórzy używają terminu *sieci neuronowe* na określenie dowolnych systemów *konekcjonistycznych*. Tutaj jednak będziemy się zajmować systemami o architekturze zbliżonej do tej, którą posiada ludzki mózg. Systemy takie są używane w wielu zadaniach takich jak rozpoznawanie mowy i obrazów oraz uczenie się maszyn.

W sieci neuronowej pojedynczy węzeł nazywa się **neuronem**. Neuron taki odbiera dane wejściowe od sąsiadów, przy czym każdemu połączeniu z sąsiadem (krawędzi) przypisana jest pewna **waga**. Połączenia takie w sieciach neuronowych nazwane są **synapsami**. Waga jest uwzględniana przy pobieraniu danych z synapsy, następnie dane te są przesyłane do kolejnych neuronów docelowych. Sieci neuronowe są użyteczne, gdyż mogą być uczone rozpoznawania pewnych zależności (wzorców) pośród danych. Sieć neuronową można „nauczyć" tego, co ma ona robić.

Joone jest łatwym w użyciu API dla pracy z sieciami neuronowymi w Javie. Posiada edytor graficzny dla tworzenia i uczenia sieci neuronowych. Choć jest również możliwe stworzenie i nauka sieci w sposób programowy, użycie edytora

jest rozwiązaniem prostszym. Gdy już stworzysz w edytorze swoją sieć i zakończysz proces nauki, możesz zagnieździć tę sieć i silnik Joone w swojej aplikacji. Rysunek 5.3 pokazuje edytor graficzny w użyciu oraz jedną z przykładowych sieci Joone.



Rysunek 5.3. Edytor graficzny Joone

Z pomocą edytora możesz tworzyć, uczyć i uruchamiać sieci neuronowe. Eksportując nauczoną sieć do pliku (używając menu *File > Export NeuralNet*), możesz wykorzystać ją później we własnym programie. Edytor Joone tworzy serializowane pliki zawierające sieci, które to pliki możesz następnie wczytać i uruchomić za pomocą silnika Joone, używając poniższego kodu.

```
import org.joone.net.NeuralNetLoader;
import org.joone.net.NeuralNet;
import org.joone.engine.Monitor;
import org.jonne.io.FileOutputSynapse;

NeuralNetLoader netLoader = new NeuralNetLoader("/projects/nn/
mynetwork.snet");
NeuralNet myNet = netLoader.getNeuralNet();
// pobierz warstwę wyjściową sieci
Layer output = myNet.getOutputLayer();
//dodaj wyjściową synapsę (połączenie) do warstwy wyjściowej
FileOutputSynapse myOutput = new FileOutputSynapse();
```

```
// ustaw plik wyjściowy na mynetwork.out
myOutput.setFileName("/project/nn/mynetwork/out");
output.addOutputSynapse(myOutput);
Monitor monitor = myNet.getMonitor();
// wykonamy jeden cykl
monitor.setTotCicles(1);
// ustaw flagę fazy uczenia na 0
monitor.setLearning(false);
// uruchom warstwy sieci
myNet.start();
// uruchom monitor
monitor.Go();
```

Powyższy przykład ładuje serializowaną sieć do instancji klasy `NeuralNet`, używając klasy pomocniczej `NeuralNetLoader`. Następnie tworzymy synapsę `FileOutputSynapse`, która będzie przechwytywać dane wyjściowe generowane przez sieć. Dane wejściowe i wyjściowe są tablicami `double[]`. Klasa `Monitor` zarządza siecią oraz pozwala na uruchamianie i zatrzymanie oraz ustawianie parametrów kontrolujących jej zachowanie. Nie uwzględniłem tutaj kodu obsługi wyjątków, aby pozostawić listing krótkim i prostym oraz aby skupić się na samym działaniu sieci. Jeśli wszystko, co chcesz zrobić, to uruchomienie sieci neuronowej i zapisanie wyników do pliku, nie musisz pisać takiej aplikacji. Istnieje klasa `NeuralNetRunner` obsługiwana z wiersza poleceń, możesz też uruchomić sieć z edytora Joone.

Odnośniki do dokładniejszej dokumentacji Joone znajdziesz na stronie internetowej książki. Dokumentacja ta posiada wiele przykładów, z którymi możesz poeksperymentować: rozpoznawanie obrazów, analizę szeregów czasowych, prognozowanie kursów walut oraz wiele innych. Ponieważ sieci mogą być serializowane, istnieje również framework dla tworzenia rozproszonych sieci neuronowych. Jest on wystarczający do stworzenia „globalnych mózgów" — to mogłoby zapewne uszczęśliwić doktora Frankensteina! Sieci neuronowe są potężną techniką, którą można zastosować w wielu zadaniach związanych z rozpoznawaniem wzorców, a Joone sprawia że tworzenie takich sieci i zagnieżdżanie we własnych aplikacjach jest bardzo łatwe.

Użycie JGAP do algorytmów genetycznych

JGAP

COLT

Istnieją problemy, dla których niełatwo jest znaleźć właściwy algorytm do ich rozwiązania. Jest to szczególnie problematyczne w wypadku procesów, które wymagają częstych zmian algorytmu w celu uwzględnienia zmiennych warunków środowiska programu. W takich przypadkach możesz spróbować napisać program, który samodzielnie wypracowuje rozwiązanie. Jest to tak zwany **algorytm genetyczny** (ang. *Genetic Algorithm*, GA). Ten typ programowania polega na wypróbowaniu wielu rozwiązań i wybraniu najlepszego. W każdym cyklu prób „najlepiej przystosowane" elementy ze zbioru rozwiązań (według wybranej *funkcji oceny* — ang. *fitness function*) są wybrane do „rozmnożenia" i tworzą następne

pokolenie. Następnie nowe pokolenie rozwiązań jest poddawane mutacjom i krzyżowane z innymi, co prowadzi do powstania wielu nowych wersji różniących się od poprzednich. Proces powtarza się dla kolejnych generacji.

Czasami proces ten prowadzi do powstania precyzyjnie wykalibrowanego algorytmu już po kilku iteracjach. Być może nawet trudno będzie zrozumieć, jak dokładnie działa wybrany algorytm — i może się on różnić zdecydowanie od rozwiązania, które wybralibyśmy, pisząc je „ręcznie", ponieważ funkcja oceny wybiera w tym wypadku algorytmy najskuteczniejsze, takie też będą algorytmy wytworzone przez ten proces.

Istnieje wiele bibliotek dla algorytmów genetycznych stworzonych w Javie. Wiele odnośników znajdziesz na stronie internetowej książki. W tej części publikacji przyjrzymy się jednej z nich — JGAP (ang. *Java Genetic Algorithms Package* — pakiet algorytmów genetycznych dla Javy). Zacznijmy od kilku podstawowych pojęć używanych w dziedzinie algorytmów genetycznych. Terminy te są zapożyczone z genetyki, mimo że obiekty używane w tych algorytmach mają raczej niewiele wspólnego z DNA. **Chromosom** (lub **genom**) przedstawia zbiór możliwych podejść do rozwiązania problemu, a **gen** przestawia jednostkę w chromosomie (**allele** są konkretnymi odmianami genu, podobnie jak z zależnością klasa-instancja). Gen może być łańcuchem tekstowym, liczbą, drzewem, programem lub dowolną inną strukturą. Aby używać JGAP, musisz na początku wybrać genom, który odpowiednio reprezentuje przestrzeń twojego problemu. Następnie wybierasz funkcję oceny dla testowania indywidualnych osobników populacji. Na koniec tworzysz obiekt konfiguracji, aby opisać charakterystykę procesu, i możesz już rozpocząć proces rozmnażania osobników.

Użyjemy kodu z naszego przykładu tablicy prawdy z części „Bity dużego kalibru", gdzie stworzyliśmy demultiplekser 1-do-4. Stworzyliśmy wtedy tablicę prawdy dla demultipleksera z dwoma wejściami adresowymi, pojedynczym wejściem danych i czterema wyjściami danych. Wymaga to dokładnie 32 bitów w tablicy prawdy. Możesz potraktować to jako testowanie pojedynczego genu o stałej długości równej 32 bity w naszej symulacji genetycznej. Jak się okazuje, nie działa to najlepiej w naszej aplikacji, ponieważ przestrzeń poszukiwania jest równa całemu zakresowi wartości typu `int`. Możemy zaprojektować to lepiej, używając genu czterobitowego reprezentującego bity wyjściowe i chromosomu o długości 8. Ponieważ cztery bity wyjściowe działają jako całość, okazuje się to być dużo lepszym wyborem dla genu. W wielu testach przeprowadzonych przy 100 000 pokoleń i populacji równej 200 gen 32-bitowy nie wytworzył pojedynczego dopasowania. Jednakże z 4-bitowym genem zwycięzca pojawiał się szybciej niż w dwudziestym pokoleniu.

Biblioteka JGAP posiada implementację interfejsu `Gene` (gen) nazwaną `IntegerGene`. Jest ona łatwa do konwersji na typ `Integer`, więc będzie się ona dobrze integrować z naszą tablicą prawdy. Funkcja oceny w JGAP posiada metodę `evaluate` z parametrem typu `IChromosome`, która zwraca wartość `double`. Wyższa wartość oznacza, że dany osobnik wykonuje zadanie lepiej. Tak wygląda nasza funkcja oceny:

```
public class DemuxFitness extends org.jgap.FitnessFunction {
   private TruthTable tt;

   public DemuxFitness() {
      // to jest nasz cel, do którego chcemy wyewoluować
      tt = new TruthTable(3,4);
      tt.store(4, 1);    // 100 -> 0001
      tt.store(5, 2);    // 101 -> 0010
      tt.store(6, 4);    // 110 -> 0100
      tt.store(7, 8);    // 111 -> 1000
   }

   public int correctBits(int data) {
      BitVector vecValue = new BitVector(new long[] {data}, 32);
      BitVector target = tt.getTruthMatrix().toBitVector();
      // możemy znaleźć liczbę właściwych bitów za pomocą:
      // count(not(xor(target, vecValue)))
      vecValue.xor(target);
      vecValue.not();
      return vecValue.cardinality();
   }

   public double evaluate(IChromosome chrom) {
      int valueTotal = 0;
      for (int i = 7; i>=0; i--) {
         IntegerGene gene = (IntegerGene) chrom.getGene(i);
         Integer value = (Integer) gene.getAllele();
         valueTotal += value;
         valueTotal <<= 4;
      }
      int correct = correctBits(valueTotal);
      // zwracamy kwadrat wartości, aby bardziej nagrodzić dokładne odpowiedzi
      return correct * correct;
   }
}
```

Moglibyśmy zwrócić po prostu liczbę poprawnych bitów w pojedynczym chromosomie. Jest to słuszne, dopóki ocena nie zacznie być niemal zawsze bliska najwyższej, kiedy to zaczyna brakować „zachęty" dla nowych osobników do bycia lepszymi. Zwracanie kwadratu liczby prawidłowych bitów bardziej „wynagradza" osobniki, które uczyniły małą poprawę w ocenie. Stwórzmy teraz obiekt konfiguracji i będziemy mogli uruchomić nasz algorytm genetyczny.

```
Configuration config = new DefaultConfiguration();
// osiem genów
Gene[] genes = new Gene[8];
for (int i = 0; i < 8; i++) {
   // liczba całkowita 4-bitowa (0-15)
   genes[i] = new IntegerGene(config, 0, 15);
}
```

```
Chromosome sample = new Chromosome(config, genes);
config.setSampleChromosome(sample);
DemuxFitness fitTest = new DemuxFitness();
config.setFitnessFunction(fitTest);
config.setPopulationSize(200);
Genotype population = Genotype.randomInitialGenotype(config);
for (int i=0; i<1000; i++) {
   population.evolve();
}
IChromosome fittest = population.getFittestChromosome();
```

Kod ten tworzy chromosom o ośmiu genach, z których każdy może mieć wartości od 0 do 15. Następnie wstawia ten chromosom do obiektu konfiguracji. Dalej ustawia funkcję oceny i rozmiar populacji, tworzy początkową przypadkową populację i uruchamia symulację, a na koniec wybiera najlepiej dopasowany chromosom. W tym przypadku znamy z góry wynik, ale jeśli potrzebowałbyś wartości zwycięskich alleli, mógłbyś wydobyć je z chromosomu. Jeśli chcesz użyć czegoś innego niż `String`, `Integer` czy bit w tworzonych przez siebie genach, możesz stworzyć własną implementację `Gene`.

Algorytmy genetyczne są, jak widać, kolejnym potężnym narzędziem dla programistów Javy, szczególnie do rozwiązywania problemów, w których rozwiązania da się testować i w środowiskach dynamicznych, gdzie może nie istnieć jedno niezmiennie najlepsze rozwiązanie. Rozwiązanie, które tworzy na drodze ewolucji sieć neuronową — za pomocą API Joone oraz JGAP — znajdziesz na stronie internetowej książki.

Tworzenie inteligentnych agentów przy użyciu Jade

JADE

Inteligentny agent (ang. *intelligent agent*) jest to samodzielny program (proces) zdolny do podejmowania decyzji i akcji bez pomocy człowieka. Programiści często używają tego określenia dla procesów działających w większym środowisku tworzonym na potrzeby takich agentów (ang. *agent framework*). Inteligentny agent jest najbardziej użyteczny, gdy jest częścią *systemu wieloagentowego*. W systemach tego typu możesz rozdzielić problem na wiele prostszych części i przypisać agentów do każdej z tych części. Agenty czasami mogą się przenosić między komputerami w ramach swojego środowiska oraz komunikować się między sobą. Inne typy agentów pozostają zawsze na jednym komputerze. Gdy **usługi sieciowe** (ang. *web services*) staną się bardziej powszechne i będą dostarczać większą różnorodność danych, prawdopodobnie staną się głównym źródłem danych i kanałem komunikacyjnym. W wielu systemach agenty działają same z siebie jako małe usługi sieciowe.

Dzięki maszynie wirtualnej zdolnej do pracy na wielu platformach oraz potężnemu modelowi bezpieczeństwa Java jest idealnym rozwiązaniem do tworzenia agentów. Wyobraź sobie na przykład wiele niegraficznych apletów pracujących w rozproszeniu nad rozwiązanie problemu. Istnieje wiele API dla Javy do tworzenia agentów, istnieje nawet konsorcjum nazywające się „Fundacja na rzecz inteligentnych agentów fizycznych" (ang. *Foundation for Intelligent Physical Agents* — FIPA), które stworzyło zbiór standardów do tworzenia frameworków agentowych. My użyjemy API nazwanego Jade, które jest jedną z najpopularniejszych bibliotek zgodnych ze standardami FIPA. Istnieje kilka zupełnie niepowiązanych projektów dla Javy nazywających się Jade, więc upewnij się przy pobieraniu z internetu, że wybrałeś ten właściwy. (Możesz też skorzystać z odnośników na stronie internetowej książki.)

W podrozdziale wprowadzimy bibliotekę Jade przez przykład związany z giełdą papierów wartościowych. Inteligentne agenty sprawdziłyby się świetnie do tzw. **program trading**, czyli automatycznego kupowania i sprzedawania papierów wartościowych w oparciu o zdarzenia, takie jak zmiana cen, podaży, nowe raporty informacyjne. Szczególnie przydatny byłby tutaj system wieloagentowy. Spróbujmy zilustrować tę ideę, pisząc prostego agenta, który kupuje 100 akcji, gdy występuje pewien specyficzny zbiór zdarzeń. Użyjemy pseudokodu do wykonania połączenia z hipotetyczną usługą sieciową, pominiemy też wszelkie zagadnienia związane z prywatnością i bezpieczeństwem. (Sięgnij do dokumentacji Jade, a znajdziesz tam przykłady z bezpiecznymi agentami.) Tak wygląda najprostszy możliwy agent:

```
public class WorthlessAgent extends jade.core.Agent {
   protected void setup() {
   }
}
```

Jade wywołuje metodę `setup`, gdy agent jest pierwszy raz ładowany (podobne działanie ma metoda `init` w apletach i `main` w aplikacjach konsolowych). W tej metodzie powinieneś umieścić kod inicjujący agenta oraz określić jego zachowania. **Zachowania** (ang. *behaviours*) definiują to, co agent robi w czasie swojego życia. Możesz wyobrazić to sobie jako coś podobnego do funktorów. W naszym wypadku moglibyśmy chcieć, aby agent okresowo sprawdzał serwer kursów, aby otrzymać ostatnie ceny danych akcji. Ten agent, „sprawdzacz wartości", mógłby wysyłać wiadomość do agenta „kupca", gdy cena osiąga zadany poziom. Istnieje wiele podklas klasy `Behaviour`, których możemy użyć. Uruchomimy `TickerBehaviour` (zachowanie typu „tykający zegar") do sprawdzania ostatnich cen akcji. Ten typ jest zachowaniem periodycznym, powtarzany jest w stałych odstępach czasu („tyknięciach" zegara). Poniżej mamy kod zachowania sprawdzającego ceny co 300 sekund:

```
public class CheckQuoteBehaviour extends TickerBehaviour {
   public CheckQuoteBehaviour(Agent a) {
      super(a, 300*1000);    // okres w milisekundach
   }

   protected void onTick() {
      // pobierz aktualną cenę
      if (getCurPrice() < 5.0) {
         //wyślij wiadomość zakupu – do zdefiniowania później
      }
      // zatrzymaj po 1000 "tyknięciach"
      if (getTickCount() > 1000) {
         stop();
      }
   }

}
```

Działanie kończy się po wywołaniu metody `stop` — po 1000 „tyknięciach" zegara. Teraz podepniemy tę klasę do agenta.

```
public class QuoteAgent extends jade.core.Agent {
   protected void Setup() {
      addBehaviour(new CheckQuoteBehaviour(this));
   }
}
```

W ten sposób mamy pełnego agenta. Możemy załadować go do serwera agentów bądź to z pomocą graficznej nakładki Jade, bądź też pisząc kod, który wykona to zadanie. Następnie wyślemy wiadomość zakupu akcji do agenta-kupca, zastępując metodę `onTick` zdefiniowaną wcześniej. Gdy agent jest załadowany do systemu, zostaje mu przypisany unikatowy numer identyfikacyjny (AID — ang. *Agent Identifier*). AID może być identyfikatorem lokalnym lub globalnym. Załóżmy, że mamy już agenta-kupca nazwanego `PurchasingAgent` o lokalnym AID „buyer" (dokumentacja Jade wyjaśnia, jak przypisać AID agentowi). Oto poprawiona metoda `onTick`:

```
// wewnątrz QuoteAgent...
   protected void onTick() {
      // pobierz aktualną cenę
      if (getCurPrice() < 5.0) {
      // wyślij wiadomość do agenta-kupca
      jade.core.AID buyingAgent = new jade.core.AID("buyer",
      AID.ISLOCALNAME);
      // prosimy innego agenta o wykonanie pewnej czynności
      ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
      // wyślij wiadomość (typu String)
```

```
      msg.setContent("kup 100 czegokolwiek");
      msg.addReceiver(buyingAgent);
      send(msg);
      // zatrzymaj po 1000 "tyknięciach"
      if (getTickCount() > 1000) {
         stop();
      }
   }
}
```

Kod dla agenta-kupca (czytanie wiadomości i odpowiedź):

```
import jade.lang.acl.ACLMessage;

public class PurchasingAgent extends jade.core.Agent {
   protected void setup() {
      addBehaviour(new jade.core.behaviours.CyclicBehaviour(this) {
         public void action() {
            ACLMessage msg = receive();
            if (msg != null) {
               String data = msg.getContent();
               //wywołaj odpowiednią metodę agenta
               Serializable result = tryToBuy(data);
               // istnieje specjalny skrót do wysyłaniu odpowiedzi
               ACLMessage reply = msg.createReply();
               reply.setPerformative(ACLMessage.INFORM);
               reply.setContentObject(result);
               send(reply);
            }
            block();
         }
      });
   }
}
```

Obiekt klasy `CyclicBehaviour` powtarza swoje działanie w nieskończoność lub dopóki metoda `done` nie zwróci `true`. Metoda `block` spowoduje blokadę instancji klasy, dopóki nie nadejdzie nowa wiadomość. Typ wiadomości `performative` oznacza wiadomość wysyłaną do odbiorcy. Istnieją inne typy zachowań w Jade. Klasa `SequentialBehaviour` zarządza grupą podległych jej zachowań sekwencyjnie. Klasa `ParallelBehaviour` wykonuje grupę zachowań równolegle. Bardziej skomplikowanym zachowanie jest FSMBehaviour (ang. *Finite-State-Machine Behaviour*), która działa jak skończona maszyna stanów. W tej klasie definiujesz zachowania reprezentujące różne stany systemu oraz przejścia pomiędzy poszczególnymi stanami zdeterminowanymi przez wydarzenie kończące poprzedni stan.

Jade posiada wiele innych cech ułatwiających tworzenie rozproszonych systemów wieloagentowych. Agenty mogą działać na serwerach J2EE, w apletach, w małych urządzeniach takich jak telefony komórkowe czy PDA. Jade posiada nawet szkielet ontologiczny dla semantyki agentów, tak aby agenty mogły się komunikować odnośnie swoich zadań i posiadać wspólne ich rozumienie. Przykłady zawarte w dokumentacji Jade pokazują niektóre bardziej zaawansowane zastosowania. Technologie agentowe odegrają prawdopodobnie wielką rolę w przyszłym rozwoju „internetu semantycznego", a umiejętność pracy z już istniejącymi strukturami na pewno pomoże programistom Javy przygotować się do tego.

Język angielski z JWordNet

JWordNet

Dla wielu badaczy **lingwistyka komputerowa** (ang. *computational linguistic*) to okno do świata ludzkiego mózgu. Rozumiejąc, co ludzie mają na myśli, gdy używają słów w określonych znaczeniach, naukowcy mogą poznać, w jaki sposób umysł ludzki działa na głębszym poziomie. Ale spójrzmy prawdzie w oczy — ludzie nie zawsze wypowiadają się w sposób sensowny i zrozumienie znaczenia ich słów może być często trudne nawet dla innych ludzi. Dla obcych z planety Alfa Centauri i dla komputerów jest to zadanie zdecydowanie bardziej skomplikowane. W najlepszych okolicznościach komputer może poznać ogólną składnię i strukturę zdania. Oczywiście zrozumienie prawdziwego *znaczenia* jest dużo trudniejsze. Chcielibyśmy, aby komputery potrafiły reagować na to, co do nich mówimy, a to wymaga wspólnego kontekstu i modelu przyczyny i efektu.

Nie jest istotne, że nie posiadamy tak naprawdę sposobu mierzenia prawdziwości rozumienia. Jeśli komputer mógłby chociaż rozumieć, które znaczenie słów miał na myśli ich autor, i potrafił znaleźć ich relacje do innych pojęć w bazie danych, moglibyśmy stworzyć system rozumienia języka naturalnego. Byłoby bardzo dobrze mieć narzędzie, które potrafiłoby wyszukać znaczenia słów i ich relacje do innych słów. Naukowcy z Princeton University Cognitive Science Laboratory (Laboratorium Nauk Kognitywnych Uniwersytetu Princeton) stworzyli słownik leksykalny dla języka angielskiego. System ten nazywa się WordNet i zawiera tysiące słów pogrupowanych w zbiory synonimów. Każde znaczenie słowa jest traktowane indywidualnie i jest połączone z innymi słowami związanymi z nim. Oprócz synonimów i antonimów relacje słów podzielone są w inne formy o egzotycznie brzmiących nazwach, jak *meronim*, *holonim*, *hipernim*, *hiponim*. Choć może bardziej brzmi to jak nazwy chorób, faktycznie chodzi o określenie zależności typu całość-część oraz nadtyp-podtyp. Jest to bardzo podobne do relacji kompozycji („posiada coś") i dziedziczenia („jest czymś") w językach obiektowych. Tabela 5.2 opisuje typy zależności słów w WordNet.

Tabela 5.2. *Określenia zależności między słowami*

Pojęcie	Definicja
Meronim	Słowo określające część czegoś większego. Przykład: *koło* jest meronimem *roweru*.
Holonim	Słowo określające większą całość w stosunku do jej części. Przykład: *rower* jest holonimem *koła*.
Hipernim	Słowo określające bardziej ogólną klasę czegoś. Przykład: *pojazd* jest hipernimem *roweru*.
Hiponim	Słowo mające bardziej szczegółowe znaczenie. Przykład: *rower* jest hiponimem *pojazdu*.

JWordNet jest interfejsem open-source dla WordNet, który możesz wykorzystać w swoich aplikacjach. To API jest bardzo użyteczne w aplikacjach przetwarzających tekst. Program może na przykład przetwarzać tekst i napotkać słowo *wing* (skrzydło/skrzydełko). Słowo to może mieć kilka znaczeń. Jeśli wcześniejsza część tekstu zawierała słowo *feather* (pióro), możesz użyć JWordNet do wyszukania holonimów słów *feather* i *wing*. Program może znaleźć, że obydwa słowa oznaczają część ptaka i że ptak jest typem zwierzęcia. W tym kontekście możesz odkryć prawidłowe znaczenie słowa *wing* i przejść do dalszego przetwarzania bazując na tej wiedzy.

Poniżej mamy przykład „Hello world" z użyciem JWordNet, w którym sprawdzamy słowo *wing*, aby znaleźć jego holonimy dla każdego znaczenia słowa.

```
configureJWordNet();    // sprawdź w dokumentacji, co obejmuje konfiguracja
DictionaryDatabase dictionary = new FileBackedDictionary();
IndexWord word = dictionary.lookupIndexWord(POS.NOUN, "wing");
System.out.println("Znaczenia 'wing':");
Synset[] senses = word.getSenses();
for (int i=0; i<senses.length; i++) {
   Synset sense = senses[i];
   System.out.println((i+1) + ". " + sense.getGloss());
   Pointer[] holo = sense.getPointers(PointerType.PART_HOLONYM);
   for (int j=0; j<holo.length; j++) {
      Word synsetWord = sense.getWord(0);
      // hasło (lemma) związane jest ze słowem, gdyż obiekt synset zawiera wiele słów
      System.out.println("   -jest częścią-> " + synsetWord.getLemma());
      // wyjaśnienie (gloss) związany jest z obiektem synset
      System.out.println(" = " + sense.getGloss());
   }
}
```

Kod ten na początku sprawdza słowo w bazie danych WordNet, szukając rzeczowników pasujących do słowa *wing*. W rezultacie otrzymuje obiekt typu `Word` (słowo), z którego możesz wydobyć tablicę obiektów `Synset`. `Synset` (synonym set, zbiór synonimów) reprezentuje szczególne znaczenie słowa i wszystkie jego synonimy. Program znajduje wszystkie powiązane holonimy dla każdego

ze znaczeń słowa *wing*. Następnie drukuje wyjaśnienie (ang. *gloss*) każdego z holonimów. Hasło (*lemma*) jest to etykieta (nagłówek) dla danego znaczenia. Przykład zwraca następujące rezultaty[6]:

```
Znaczenia słowa 'wing':
1. ruchomy organ umożliwiający latanie (jeden z pary)
   -jest częścią-> ptak = ciepłokrwisty kręgowiec znoszący jajka ...
   -jest częścią-> nietoperz = nocny ssak przypominajacy mysz
   -jest częścią-> owad = drobny stawonóg oddychający powietrzem
   -jest częścią-> anioł = duchowa istota towarzysząca Bogu
2. jedna z poziomych części samolotu
   -jest częścią-> samolot = maszyna latająca ...
3. skrzydełko drobiu; "on wolał skrzydełka jeść pałeczkami"
   -jest częścią-> drób = jadalne mięso ptactwa ...
4. jednostka wojskowa w wojskach powietrznych
5. boczna część frontu wojsk ...
6. boczna część budynku
   -jest częścią-> budynek = struktura posiadająca dach i ściany ...
```

Użyłem znaku wielokropka, aby skrócić niektóre długie linie tekstu. JWordNet obsługuje słowniki plikowe, pamięciowe i bazodanowe. Istnieje również wersja RDF dla WordNet — ten temat opisywany był w rozdziale 4. JWordNet jest potężnym narzędziem przy połączeniu z technikami opisanymi w rozdziale 2. (dopasowanie tekstu) oraz rozdziale 4. (analiza semantyczna). Odnośnik do danych RDF i inne informacje o WordNet i JWordNet znajdziesz na stronie internetowej książki.

Podsumowanie

W rozdziale przyjrzeliśmy się wielu bibliotekom przeznaczonym do zastosowań w matematyce, fizyce i innych naukach. Według mnie jest to jeden z najpotężniejszych obszarów zastosowań Javy. Mam nadzieję że programiści bibliotek open-source będą dalej tworzyć wspaniałe projekty takie jak te opisane powyżej. W następnych rozdziałach spotkamy się z innymi bibliotekami, które mogą być użyte w naukach ścisłych, gdy będziemy się zajmować grafiką, multimediami i integracją projektów.

[6] Program oczywiście zwraca opisy po angielsku — *przyp. tłum.*