helion.pl

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. Wprowadzenie

Tłumaczenie: Rafał Jońca
ISBN: 83-7197-925-8
Tytuł oryginału: Learning Java
Format: B5, stron: 780
Zawiera CD-ROM

Przystępne, a zarazem kompletne wprowadzanie do języka programowania, który zmienił sposób myślenia wielu programistów. W ostatnich latach Java wyprzedziła pod względem popularności inne języki, takie jak C++ i Visual Basic, spełniając większość wymagań stawianych przez twórców aplikacji i stała się najważniejszym językiem nowego pokolenia programistów – w szczególności projektantów aplikacji działających w Internecie.

Książka pozwala na przyswojenie podstaw języka Java. Dzięki niej poznasz sam język, biblioteki klas, techniki programistyczne i idiomy. „Java. Wprowadzenie" zawiera wiele łatwych w zrozumieniu przykładów wraz z kodem źródłowym. Pozwolą Ci one zapoznać się z wieloma cechami, funkcjami i interfejsami programistycznymi Javy.

Załączony CD-ROM poza przykładami omawianymi w książce zawiera także pełną wersję J2SE SDK 1.4, środowisko NetBeans, narzędzie make (Ant) i serwer aplikacji Tomcat z projektu Jakarta oraz BeanShell, prosty, darmowy język skryptowy Javy napisany przez jednego z autorów tej książki, Pata Niemeyera.

Tematy omówione w tej książce dotyczą:

- technik programowania zorientowanego obiektowo,
- interfejsów programistycznych serwletów i aplikacji internetowych,
- asercji języka i obsługi wyjątków,
- pakietu NIO umożliwiającego tworzenie złożonych systemów wejścia-wyjścia,
- programowania wykorzystującego wątki,
- komponentów Swing służących do budowy okienkowego interfejsu użytkownika
- nowych interfejsów programistycznych wersji 1.4: wyrażeń regularnych, właściwości i dzienników zdarzeń,
- JavaBeans i darmowego środowiska NetBeans,
- Java Plug-in, podpisywaniu apletów i Java Web Start,
- XML (omówiono: SAX, DOM, DTD, XSLT)

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa *11*

Rozdział 1. *Jeszcze jeden język?* *17*

Java 17
Maszyna wirtualna 20
Java a inne języki programowania 23
Bezpieczeństwo projektowania 25
Bezpieczeństwo implementacji 30
Aplikacja i bezpieczeństwo na poziomie użytkownika 34
Java i Internet 36
Java jako podstawowy język aplikacji 39
Przeszłość i przyszłość Javy 39

Rozdział 2. *Pierwsza aplikacja* *43*

WitajJava 43
WitajJava2: Następna wersja 54
WitajJava3: Atak przycisków! 62
WitajJava4: Zemsta Netscape'a 70

Rozdział 3. *Narzędzia pracy* *79*

Interpreter Javy 79
Zmienna classpath 81
Pliki zasad 83
Kompilator Javy 87
Pliki JAR (Java Archive) 89

Rozdział 4. *Język Java* *93*
Kodowanie tekstu 93
Komentarze 94
Typy 95
Instrukcje i wyrażenia 101
Wyjątki 108
Asercje 119
Tablice 122

Rozdział 5. *Obiekty w Javie* *129*
Klasy 130
Metody 135
Tworzenie obiektu 142
Usuwanie obiektów 145

Rozdział 6. *Związki między klasami* *149*
Klasy pochodne i dziedziczenie 149
Interfejsy 161
Pakiety i jednostki kompilacji 166
Widoczność zmiennych i metod 169
Tablice a hierarchia klas 171
Klasy wewnętrzne 173

Rozdział 7. *Praca z obiektami i klasami* *183*
Klasa Object 183
Klasa Class 187
Odbicie 189

Rozdział 8. *Wątki* *199*
Wprowadzenie do wątków 200
Wątki w aplecie 207
Synchronizacja 210
Harmonogramy i priorytety 218
Grupy wątków 223
Wydajność wątków 224

Rozdział 9. ***Praca z tekstem*** ***227***

Inne interfejsy programistyczne związane z tekstem 229
Klasa String 230
Przetwarzanie i formatowanie tekstu 238
Internacjonalizacja 240
Pakiet java.text 242
Wyrażenia regularne 246

Rozdział 10. ***Podstawowe narzędzia*** ***261***

Narzędzia matematyczne 261
Daty 266
Czasomierze 268
Zbiory 269
Właściwości 280
Interfejs Preferences 283
Interfejs Logging 286
Obserwatorzy i obserwowani 293

Rozdział 11. ***Funkcje wejścia-wyjścia*** ***295***

Strumienie 295
Pliki 309
Serializacja 318
Kompresja danych 321
Pakiet NIO 324

Rozdział 12. ***Programowanie sieciowe*** ***337***

Gniazda 338
Gniazda datagramów 352
Prosty protokół serializacji obiektów 356
Zdalne wywoływanie metod 360
Skalowalne we-wy z NIO 372

Rozdział 13. ***Programowanie dla Internetu*** ***381***

Adresy URL 381
Klasa URL 382
Procedury obsługi w praktyce 385

Rozdział 14. ***Serwlety i aplikacje internetowe*** **393**
Serwlety — użyteczne narzędzia 394
Aplikacje internetowe 394
Cykl życia serwletu 395
Serwlety stron WWW 396
Serwlet WitajKliencie 397
Odpowiedź serwletu 399
Parametry serwletu 400
Serwlet PokazParametry 401
Zarządzanie sesją użytkownika 403
Interfejs programistyczny ServletContext 409
Pliki WAR i ich rozmieszczenie 410
Ponowne wczytywanie aplikacji internetowych 414
Strony błędów i indeksów 414
Bezpieczeństwo i identyfikacja 416
Filtry serwletów 419
Tworzenie plików WAR w Ant 427

Rozdział 15. ***Swing*** **429**
Komponenty 432
Pojemniki 439
Zdarzenia 445
Podsumowanie zdarzeń 452
Robot AWT! 458
Wielowątkowość w Swing 458

Rozdział 16. ***Używanie komponentów Swing*** **461**
Przyciski i etykiety 461
Opcje i grupy opcji 465
Listy i listy rozwijane 467
Pole z przewijaniem 470
Granice 472
Menu 475
Klasa PopupMenu 478
Klasa JScrollPane 480

Klasa JSplitPane 482
Klasa JTabbedPane 483
Paski przewijania i suwaki 485
Okna dialogowe 487

Rozdział 17. ***Więcej komponentów Swing*** **493**
Komponenty tekstowe 493
Zmiana aktywności 507
Drzewa 508
Tabele 513
Pulpity 520
Własny wygląd i zachowanie 522
Tworzenie własnych komponentów 524

Rozdział 18. ***Klasy zarządzające układem graficznym*** **529**
FlowLayout 531
GridLayout 532
BorderLayout 533
BoxLayout 536
CardLayout 537
GridBagLayout 538
Niestandardowe klasy zarządzające układem 553
Pozycjonowanie bezwzględne 553
SpringLayout 554

Rozdział 19. ***Rysowanie za pomocą 2D API*** **555**
Większy obraz 555
Potok renderingu 557
Krótka podróż po Java 2D 559
Wypełnianie kształtów 566
Obrysy kształtów 567
Używanie czcionek 568
Wyświetlanie obrazów 573
Techniki rysowania 576
Drukowanie 584

Rozdział 20. ***Praca z obrazami i innymi mediami*** **587**
ImageObserver 587
MediaTracker 590
Tworzenie danych obrazu 592
Filtrowanie danych obrazu 603
Prosty dźwięk 607
Java Media Framework 608

Rozdział 21. ***JavaBeans*** **611**
Czym jest Bean? 611
Środowisko programistyczne NetBeans 613
Właściwości i dostosowanie 617
Adaptery i dołączanie zdarzeń 618
Łączenie właściwości 622
Tworzenie Beans 624
Ograniczenia projektowania wizualnego 632
Serializacja kontra generowanie kodu 632
Dostosowanie danych za pomocą BeanInfo 632
Ręczne pisanie kodu z Beans 636
BeanContext i BeanContextServices 641
Java Activation Framework 642
Enterprise JavaBeans 642

Rozdział 22. ***Aplety*** **645**
Polityka apletów 645
Klasa JApplet 646
Znacznik <APPLET> 657
Korzystanie z Java Plug-in 662
Java Web Start 664
Używanie podpisów cyfrowych 665

Rozdział 23. ***XML*** **675**
Tło 675
Podstawy XML 677
SAX 681

DOM 689
Walidacja dokumentów 693
XSL i XSLT 696
Usługi sieciowe 701

Dodatek A ***Procedury obsługi zawartości i protokołu*** ***703***

Dodatek B ***BeanShell — skrypty w Javie*** ***719***

Słowniczek ***725***

Skorowidz ***741***

3

Narzędzia pracy

Istnieje wiele środowisk programistycznych Javy, od tradycyjnych, jak edytor tekstu i narzędzia linii poleceń, po bardzo zaawansowane *IDE* (*Integrated Development Environment*), takie jak Forte for Java Suna, Visual Café firmy WebGain i JBuilder firmy Inprise. Przykłady w tej książce powstawały przy użyciu wersji Solaris i Windows standardowego SDK (*Software Development Kit*), więc zajmiemy się tylko tymi narzędziami. Gdy będziemy mówili o kompilatorze lub interpreterze, będziemy mieli na myśli ich wersje dla linii poleceń, więc książka bazuje na tym, że umiemy już obsłużyć środowisko linii poleceń znane z Unixa lub DOS-a. Jednak podstawowe cechy opisywanego kompilatora i interpretera Javy firmy Sun powinny mieć zastosowanie także w innych środowiskach Javy.

W tym rozdziale opiszemy narzędzia, za pomocą których będziemy kompilować i uruchamiać aplikacje Javy. Ostatnia część rozdziału omawia sposób umieszczania plików klas Javy w archiwach Javy (plikach JAR). W rozdziale 22. zajmiemy się podpisywaniem klas w pliku JAR, dzięki czemu będziemy przyznawać większe prawa zaufanym programom.

Interpreter Javy

Interpreter Javy to oprogramowanie, które implementuje maszynę wirtualną i wykonuje aplikacje Javy. Może to być samodzielna aplikacja, jak program *java* dostarczany z SDK, lub część większej aplikacji, jak przeglądarka. Zwykle interpreter napisany jest w kompilowanym języku najbardziej odpowiednim dla danej platformy. Pozostałe narzędzia, takie jak kompilatory i środowiska programistyczne, są przeważnie implementowane bezpośrednio w Javie, aby uzyskać jak największą uniwersalność. Forte for Java (w bezpłatnej wersji znajduje się także na dostarczonym z książką CD-ROM-ie o nazwie NetBeans) jest przykładem środowiska programistycznego napisanego wyłącznie w Javie.

Interpreter Javy wykonuje wszystkie zadania systemu uruchomieniowego Javy. Wczytuje pliki klas i interpretuje skompilowany kod bajtowy. Weryfikuje skompilowane klasy pochodzące z niepewnych źródeł. W implementacjach, gdzie jest obsługiwana kompilacja

dynamiczna, interpreter służy także jako wyspecjalizowany kompilator, który zamienia kod bajtowy na kod maszynowy odpowiedni dla danej platformy.

W większej części książki utworzymy samodzielne programy Javy, często będziemy też wspominać o apletach. Obydwa rodzaje programów są uruchamiane w interpreterze Javy. Jedyna różnica polega na tym, że samodzielny program zawiera wszystkie części; jest to aplikacja, która może działać niezależnie. Aplet to jakby osadzany moduł. Interpreter nie uruchamia bezpośrednio apletów, ponieważ muszą one stanowić część większej aplikacji. Aby uruchomić aplet, używamy przeglądarki lub programu *appletviewer* dostarczanego z SDK. HotJava — przeglądarka napisana w Javie — i *appletviewer* to samodzielne aplikacje wykonywane bezpośrednio przez interpreter Javy; programy te implementują dodatkową strukturę wymaganą do uruchomienia apletów Javy.

W samodzielnej aplikacji Javy musi istnieć przynajmniej jedna klasa zawierająca metodę `main()`, która ma w sobie instrukcje potrzebne do uruchomienia całej aplikacji. Aby uruchomić aplikację, włączamy interpreter, podając jako parametr nazwę klasy. Możemy także przekazać opcje do interpretera i argument dla aplikacji. Interpreter Javy firmy Sun nosi nazwę *java*.

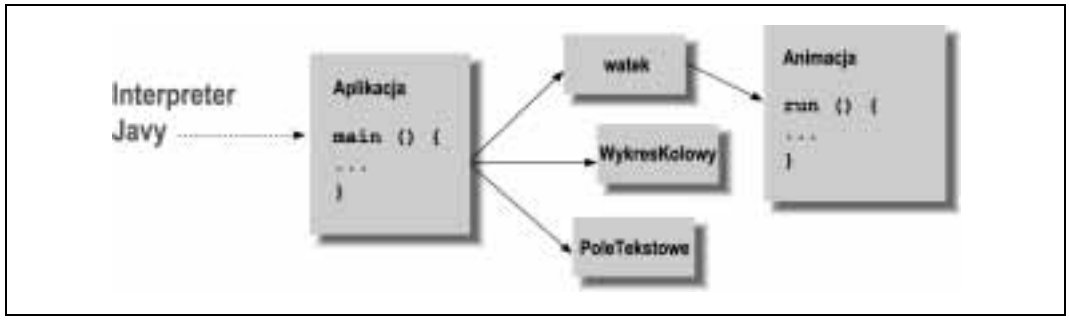
```
% java [opcje interpretera] nazwa_klasy [argumenty programu]
```

Klasa powinna zostać określona pełną nazwą wraz z nazwą pakietu, jeśli jest częścią takowego. Pamiętajmy jednak o tym, aby nie umieszczać rozszerzenia *.class* po nazwie klasy. Oto kilka przykładów.

```
% java zwierzeta.ptaki.DuzyPtak
% java Test
```

Interpreter szuka klas w ścieżkach określonych w zmiennej `classpath`. W następnym podrozdziale dokładniej zajmiemy się tą zmienną. Jest ona zwykle określana przez zmienną środowiska, którą możemy jednak przesłonić opcją `-classpath` linii poleceń.

Po wczytaniu klasy określonej w linii poleceń interpreter wykonuje metodę `main()` tej klasy. Od tego momentu aplikacja może rozpocząć wykonywanie dodatkowych wątków, odnosić się do innych klas, tworzyć interfejsy użytkownika bądź inne struktury (patrz rysunek 3.1).



Rysunek 3.1. Początek działania aplikacji napisanej w Javie

Metoda `main()` musi posiadać poprawną *sygnaturę*. Sygnatura metody to zbiór informacji definiujących metodę. Wlicza się w to nazwę metody, argument, zwracany typ, a także modyfikatory widoczności i typu. Metoda `main()` musi być metodą `public` i `static`, która przyjmuje tablicę obiektów `String` i nie zwraca żadnej wartości (`void`).

```
public static void main( String[] argumenty )
```

Ponieważ `main()` jest metodą publiczną i statyczną, może ją wywołać inna klasa za pomocą nazwy klasy i nazwy metody. Implementacje modyfikatorów widoczności, takich jak `public`, i znaczenie `static` omówimy w rozdziałach od 4. do 6.

Pojedynczy argument metody, tablica obiektów `String`, zawiera argumenty z linii poleceń przekazane do aplikacji. Nazwa, jaką nadamy parametrowi, nie ma znaczenia, liczy się tylko jego typ. W Javie zawartość argumentu to rzeczywista tablica. Nie trzeba przekazywać parametru z liczbą elementów, ponieważ tablica dokładnie wie, ile ich zawiera i udostępnia tę informację.

```
int liczbaargumentow = argumenty.lenght;
```

`argumenty[0]` zawiera pierwszy argument z linii poleceń itd. Łatwo zauważyć, że różni się od argumentów w C i C++, gdzie zerowy argument zawierał nazwę aplikacji. Jeśli jesteśmy przyzwyczajeni do przetwarzania argumentów w C, musimy pamiętać o tej różnicy.

Interpreter Javy działa do czasu, aż metoda `main()` podanej klasy nie zakończy działania, a wszystkie wątki, które zostały załączone, nie skończą się. Specjalne wątki służące jako wątki „demony" są po cichu zabijane, gdy reszta aplikacji zakończy działanie.

Właściwości systemu

Java nie zapewnia bezpośredniego dostępu do zmiennych środowiska systemu operacyjnego będącego gospodarzem. Umożliwia jednak przekazanie dowolnej liczby *właściwości systemu* do aplikacji w trakcie włączania interpretera. Właściwości systemu to po prostu pary nazwa-wartość, które są dostępne przez statyczną metodę `System.getProperty()`. Możemy używać tych właściwości jako alternatywnego sposobu przekazywania argumentów z linii poleceń do aplikacji. Każda wartość jest przekazywana do interpretera za pomocą opcji `-D`, po której następuje *nazwa=wartość*, na przykład:

```
% java -Dulica=sezamkowa -Dscena=aleja zwierzeta.ptaki.DuzyPtak
```

Wartość właściwości `ulica` jest dostępna w następujący sposób:

```
String ulica = System.getProperty("ulica");
```

Zmienna classpath

Pojęcie zmiennej *path* powinno być znane każdemu, kto pracował w systemie DOS lub Unix. Jest to zmienna środowiska, która zapewnia listę miejsc do przeszukania w celu znalezienia pewnego zasobu. Najbardziej znanym przykładem są ścieżki do plików wykonywalnych.

W powłoce systemu Unix zmienna środowiska `PATH` zawiera oddzielone średnikami katalogi, w których należy szukać programu, gdy użytkownik wpisze polecenie. Podobnie działa zmienna środowiska Javy `CLASSPATH`, zawierająca listę miejsc, w których mogą się znajdować pakiety z plikami klas Javy. Zarówno kompilator, jak i interpreter korzysta z tej zmiennej do znajdowania klas w lokalnym systemie.

Elementem ścieżki do klas może być nazwa katalogu lub nazwa *pliku archiwum*. Java obsługuje archiwa plików klas we własnym formacie *JAR* oraz w tradycyjnym już formacie *ZIP*. *JAR* i *ZIP* to w zasadzie ten sam format, tylko archiwa *JAR* zawierają dodatkowe pliki opisujące zawartość archiwum. Pliki *JAR* są tworzone przez narzędzie *jar*; większość narzędzi do tworzenia archiwów *ZIP* jest publicznie dostępna, często nawet za darmo. Format archiwów umożliwia przenoszenie dużych grup klas i ich zasobów w pojedynczym pliku; interpreter Javy automatycznie wydobędzie z archiwum odpowiednie klasy, gdy będą one potrzebne.

Dokładny format i sposób ustawiania zmiennej `CLASSPATH` zmienia się w zależności od systemu. W systemach uniksowych ustawiamy zmienną środowiska `CLASSPATH`, podając oddzielone dwukropkami nazwy katalogów lub nazwy archiwów.

```
CLASSPATH=/home/jan/Java/klasy:/home/marcin/starocie/cos.jar:.
export CLASSPATH
```

Przykład określa trzy ścieżki, w których należy szukać klas: katalog w katalogu macierzystym użytkownika, plik *JAR* w katalogu macierzystym innego użytkownika i aktualny katalog, który zawsze jest określany jako kropka (.). Dołączanie do listy wyszukiwania aktualnego katalogu jest przydatne, gdy zamierzamy majstrować przy klasach, ale ogólnie umieszczanie aktualnego katalogu w jakichkolwiek ścieżkach nie jest dobrym pomysłem.

W systemie Windows zmienną środowiska `CLASSPATH` ustawiamy, podając nazwy katalogów i plików oddzielone średnikiem.

```
set CLASSPATH=D:\uzytkownicy\jan\Java\klasy;.
```

Interpreter Javy i pozostałe narzędzia linii poleceń znajdują podstawowe klasy, które zawiera każda instalacja Javy. Wszystkie klasy z pakietów `java.lang`, `java.io`, `java.net` i `javax.swing` to podstawowe klasy. Nie musimy dołączać ich do ścieżki wyszukiwania; interpreter i inne narzędzia potrafią znaleźć je samodzielnie.

W trakcie wyszukiwania interpreter sprawdza lokalizacje po kolei. Wyszukiwanie łączy ścieżkę położenia z pełną nazwą klasy. Rozważmy na przykład wyszukiwanie klasy `zwierzeta.ptaki.DuzyPtak`. Wyszukiwanie w lokalizacji */usr/lib/java* ze ścieżki klas oznacza wyszukiwanie pliku klasy w */usr/lib/java/zwierzeta/ptaki/DuzyPtak.class*. Wyszukiwanie w pliku *ZIP* lub *JAR* podanym w ścieżce, powiedzmy */home/jan/Java/narzedzia/narzedzia.jar*, oznacza, że interpreter poszuka w tym archiwum komponentu o ścieżce *zwierzeta/ptaki/DuzyPtak.class*.

W interpreterze i kompilatorze Javy (odpowiednio *java* i *javac*) ścieżkę wyszukiwania można też podać w opcji `-classpath`.

```
% javac -classpath /pkg/sdk/lib/klasy.zip:/home/jan/java:. Klasa.java
```

Jeśli nie określimy zmiennych środowiska `CLASSPATH`, domyślnie brany jest pod uwagę aktualny katalog (.); oznacza to, że bez problemów mamy dostęp do plików w aktualnym katalogu. Jeśli określimy własną ścieżkę wyszukiwania i nie dołączymy aktualnego katalogu, pliki te nie będą dostępne.

javap

Użytecznym narzędziem, o którym warto wiedzieć, jest polecenie *javap*. Przy jego użyciu wyświetlamy opis skompilowanej klasy. Nie musimy posiadać kodu źródłowego, a nawet nie musimy wiedzieć, gdzie dokładnie znajduje się klasa, wystarczy, że znajduje się w ścieżce wyszukiwania. Na przykład

```
% javap java.util.Stack
```

wyświetli informacje o klasie `java.util.Stack`:

```
Compiled from Stack.java
public class java.util.Stack extends java.util.Vector {
  public java.util.Stack( );
  public java.lang.Object push(java.lang.Object);
  public synchronized java.lang.Object pop( );
  public synchronized java.lang.Object peek( );
  public boolean empty( );
  public synchronized int search(java.lang.Object);
}
```

Jest to bardzo przydatne, gdy nie mamy dostępu do innej dokumentacji, a także podczas badania problemów związanych z klasami. Warto przyjrzeć się `javap` z opcją `-c`, która powoduje wyświetlenie instrukcji wszystkich metod klasy dla maszyny wirtualnej.

Pliki zasad

Jedną z rzeczywistych nowości w Javie jest bezpieczeństwo wbudowane w sam język. Już w rozdziale 1. pisaliśmy, że maszyna wirtualna Javy potrafi weryfikować pliki klas, a zarządca bezpieczeństwa wprowadza ograniczenia na klasy. We wcześniejszych wersjach Javy implementacja zasad bezpieczeństwa polegała na napisaniu klasy zarządcy bezpieczeństwa i wykorzystaniu go w aplikacji. Dużym postępem była wersja Java 1.2, w której wprowadzono system zabezpieczeń wykorzystujący deklaracje. W tym systemie można pisać *pliki zasad* — tekstowy opis tego, co mogą klasy — które są znacznie prostsze i nie wymagają zmian w kodzie. Pliki te informują zarządcę, co komu wolno, a czego i komu nie wolno.

Za pomocą zasad bezpieczeństwa możemy odpowiedzieć na pytania: „Jak można zapobiec wykradzeniu przez program pobrany z pewnego miejsca w Internecie informacji

z komputera i wysłaniu ich do kogoś innego?", „Jak się uchronić przed złośliwym programem, który będzie próbował usunąć wszystkie pliki z dysku?" Większość platform komputerowych nie odpowiada na te pytania.

Na początku istnienia Javy dość głośno mówiło się o zabezpieczeniach apletów. Aplety działają z ograniczeniami, które uniemożliwiają im wykonywanie niepewnych zadań, takich jak odczyt z i zapis na dysk lub dostęp do dowolnych komputerów w sieci. Za pomocą plików zasad bezpieczeństwa możemy te same ograniczenia nałożyć na aplikacje bez ich modyfikowania. Możemy dokładnie określić, co wolno aplikacjom. Możemy na przykład pozwolić aplikacji na dostęp do dysku, ale tylko do wybranego katalogu, lub udostępnić tylko niektóre adresy sieciowe.

Zrozumienie bezpieczeństwa i jego zasad jest bardzo ważne, więc teraz tym się zajmiemy. W praktyce jednak zapewne nie będziemy korzystać z narzędzi, chyba że tworzymy szkielet do uruchamiania aplikacji z wielu niepewnych źródeł. Przykładem takiego szkieletu może być aplikacja Java Web Start. Instaluje i aktualizuje ona aplikacje pobierane z sieci oraz wprowadza dla nich ograniczenia zdefiniowane przez użytkownika.

Domyślny zarządca bezpieczeństwa

Gdy uruchamiamy aplikację lokalnie, domyślnie nie zostaje zainstalowany żaden zarządca bezpieczeństwa. Możemy włączyć zabezpieczenia przy użyciu opcji interpretera Javy, która spowoduje instalację domyślnego zarządcy bezpieczeństwa. Aby zaprezentować jego sposób działania, przygotowaliśmy niewielki program, który robi coś niepewnego: próbuje się połączyć z pewnym komputerem w Internecie. (Dokładnym opisem programowania sieciowego zajmiemy się w rozdziałach 12. i 13.).

```
//plik: ImperiumZla.java
import java.net.*;

public class ImperiumZla {
  public static void main(String[] argum) throws Exception{
    try {
      Socket s = new Socket("207.46.131.13", 80);
      System.out.println("Połączony!");
    }
    catch (SecurityException e) {
      System.out.println("SecurityException: połączenie nie powiodło się.");
    }
  }
}
```

Jeśli wykonamy ten program w interpreterze Javy, nastąpi połączenie sieciowe.[1]

```
C:\> java ImperiumZla
Połączony!
```

[1] Lub też nie nastąpi, bo przedstawiony przykładowy serwer może nie działać lub my możemy nie być podłączeni do Internetu. W takim przypadku po kilku sekundach program wyłączy się z powodu wyjątku minięcia czasu oczekiwania na połączenie. Oczywiście dowodzi to tezy autorów o dostępie programu do sieci, gdyż to drugi serwer nie odpowiedział — *przyp. tłum.*

Ale ponieważ program jest „zły”, zainstalujmy domyślnego zarządcę bezpieczeństwa.

```
C:\> java -Djava.security.manager ImperiumZla
SecurityException: połączenie nie powiodło się.
```

Już lepiej, ale załóżmy, że ta aplikacja rzeczywiście ma dobry powód, by połączyć się z podanym serwerem. Chcemy pozostawić zarządcę, by czuć się bezpiecznie, ale jednocześnie postanawiamy umożliwić programowi to połączenie.

Narzędzie policytool

Aby umożliwić `ImperiumZla` dostęp do sieci, musimy otworzyć odpowiedni plik zasad, który zawiera potrzebne pozwolenia. Przydatne narzędzie, nazwane *policytool* i dostarczane wraz z SDK, pomaga tworzyć takie pliki. Uruchamiamy je z linii poleceń w następujący sposób:

```
C:\> policytool
```

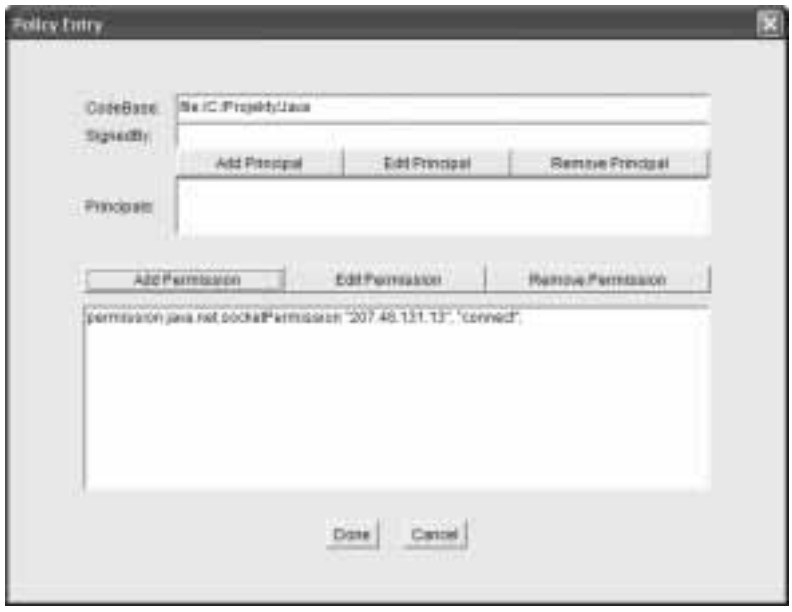
Możemy otrzymać komunikat o błędzie po uruchomieniu programu, ponieważ nie mógł on znaleźć domyślnego pliku zasad. Nie jest to powód do zmartwień, po prostu kliknijmy *OK*, aby pozbyć się komunikatu.

Dodamy teraz pozwolenia dostępu do sieci dla aplikacji `ImperiumZla`. Aplikacja jest identyfikowana za pomocą pochodzenia określanego na podstawie adresu URL. W tym przypadku będzie to adres typu `file:`, który wskazuje na położenie aplikacji `ImperiumZla` na dysku.

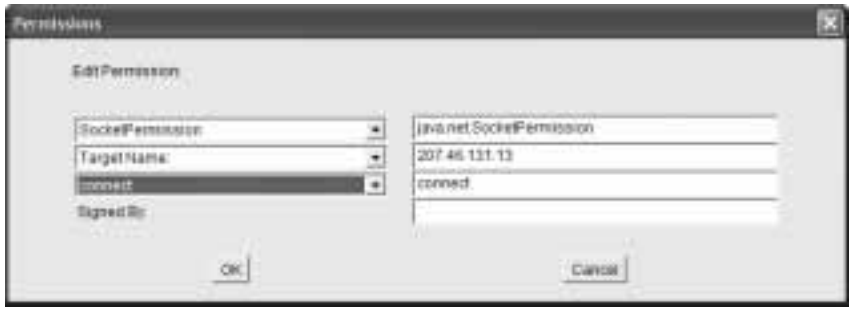
Jeśli uruchomimy program *policytool*, powinniśmy zobaczyć jego główne okno przedstawione na rysunku 3.2. Po kliknięciu przycisku *Add Policy Entry* pojawi się jeszcze jedno okno, podobne do tego z rysunku 3.3 (ale z pustymi polami).

Rysunek 3.2. Okno narzędzia policytool

Najpierw wpiszemy w pole *CodeBase* adres URL katalogu, w którym znajduje się aplikacja `ImperiumZla`. Następnie kliknijmy przycisk *Add Permission*. Pojawi się jeszcze jedno okno przedstawione na rysunku 3.4.



Rysunek 3.3. Dodawanie zasady



Rysunek 3.4. Tworzenie nowego pozwolenia

Wybieramy opcję *SocketPermission* z pierwszej listy rozwijanej. Następnie w drugim polu tekstowym podajemy adres internetowy, z którym łączy się aplikacja. Z trzeciej listy rozwijanej wybieramy opcję *connect*. Klikamy przycisk *OK* i powinniśmy zobaczyć nowe pozwolenie w oknie zasad (rysunek 3.3).

Klikamy przycisk *Done*, aby zakończyć tworzenie zasady. Następnie wybieramy polecenie *Save As* z menu *File* i zapisujemy zasadę w pliku o opisowej nazwie, na przykład *ImperiumZla.policy*. Możemy zamknąć program *policytool*, nie jest już nam potrzebny.

Utworzony przed chwilą plik jest zwykłym plikiem. Po otwarciu wygenerowanego pliku za pomocą edytora tekstowego przekonamy się, że składnia utworzonej zasady jest prosta.

```
grant codeBase "file:/C:/Projekty/Java/" {
  permission java.net.SocketPermission "207.46.131.13", "connect";
};
```

Możemy teraz pominąć program *policytool* i tworzyć nowe pliki zasad za pomocą edytora tekstu, jeśli ten sposób jest bardziej komfortowy.

Wykorzystanie pliku zasad z domyślnym zarządcą bezpieczeństwa

Skoro przebrnęliśmy już przez tworzenie pliku zasad, pora go wykorzystać. Możemy za pomocą opcji z linii poleceń interpretera wskazać domyślnemu zarządcy bezpieczeństwa, z jakiego pliku zasad ma korzystać.

```
C:\> java -Djava.security.manager -Djava.security.policy=ImperiumZla.policy
➥ImperiumZla
Połączony!
```

Teraz aplikacja `ImperiumZla` może wykonywać połączenia sieciowe, ponieważ wyraźnie pozwoliliśmy na to w pliku zasad. Domyślny zarządca stanowi zabezpieczenie w pozostałych kwestiach. Program nadal nie może odczytywać i zapisywać plików w katalogach innych niż ten, w którym się znajduje, nie może także połączyć się z innymi adresami niż wskazany. Teraz chwilę odpoczniemy i będziemy delektować się swoimi prawami dostępu.

W rozdziale 22. jeszcze raz posłużymy się narzędziem *policytool*, gdy będziemy omawiać podpisywane aplety. W tym rozdziale identyfikacja wykorzystuje adres URL, co nie jest najbezpieczniejsze. W sieci można zastosować sztuczki i otrzymać kod, który wydaje się pochodzić z pewnego miejsca, ale w rzeczywistości został podstawiony. Bardziej wiarygodny jest kod podpisany cyfrowo (patrz rozdział 22.).

Kompilator Javy

W tym podrozdziale opiszemy krótko *javac*, kompilator Javy z pakietu SDK. (Jeśli korzystamy z innego środowiska programistycznego, przechodzimy do następnego podrozdziału). Kompilator Javy został napisany w całości w Javie, więc jest dostępny na każdej platformie obsługującej system uruchamiania programów Javy. Kompilator *javac* zamienia kod źródłowy na skompilowaną klasę zawierającą instrukcje dla maszyny wirtualnej. Pliki źródłowe posiadają rozszerzenie *.java*; skompilowane klasy — rozszerzenie *.class*. Każdy plik źródłowy to pojedyncza jednostka kompilacji. W rozdziale 6. przekonamy się, że klasy z danej jednostki kompilacji współdzielą pewne cechy, takie jak instrukcje `package` i `import`.

Kompilator pozwala na jedną publiczną klasę w pliku i wymaga, aby plik miał tę samą nazwę co klasa. Jeśli nazwy nie są identyczne, kompilator zgłasza błąd. Pojedynczy plik może zawierać wiele klas, jeśli tylko jedna z nich jest publiczna. Unikajmy jednak umieszczania wielu klas w jednym pliku. Dołączanie wielu niepublicznych klas do jednego pliku *.java* to prosty sposób związania tych klas z klasą publiczną. Możemy jeszcze rozważyć wykorzystywanie klas wewnętrznych (patrz rozdział 6.).

Teraz przykład. Umieszczamy poniższy kod źródłowy w pliku *DuzyPtak.java*.

```
package zwierzeta.ptaki

public class DuzyPtak extends Ptak {
  ...
}
```

Kompilujemy go teraz za pomocą polecenia:

```
% javac DuzyPtak.java
```

W odróżnieniu od interpretera, który przyjmuje tylko nazwę klasy jako argument, kompilator potrzebuje nazwy pliku. Powyższe polecenie tworzy plik klasy *DuzyPtak.class* w tym samym katalogu co plik źródłowy. Chociaż warto posiadać plik klasy w tym samym katalogu co kod źródłowy w trakcie testowania prostych przykładów, w większości rzeczywistych aplikacji musimy przechowywać pliki klas w odpowiedniej hierarchii.

Możemy skorzystać z opcji `-d` w celu określenia innego katalogu, w którym umieszczamy generowane pliki klas. Podany katalog to korzeń hierarchii klas, więc pliki *.class* są umieszczane w tym katalogu lub jego podkatalogach w zależności od tego, czy klasa znajduje się w pakiecie. (Kompilator tworzy potrzebne podkatalogi, gdy nie istnieją). Możemy na przykład skorzystać z poniższego polecenia, aby utworzyć plik *DuzyPtak.class* w lokalizacji */home/jan/Java/klasy/zwierzeta/ptaki/DuzyPtak.class*.

```
% javac -d /home/jan/Java/klasy DuzyPtak.java
```

W jednym wywołaniu *javac* możemy podać wiele plików *.java*; kompilator utworzy plik klasy dla każdego pliku źródłowego. Nie musimy jednak podawać kodów źródłowych innych klas, do których się odnosimy, o ile klasy te zostały już skompilowane. W trakcie kompilacji Java rozwiązuje wszystkie odniesienia do klas, używając ścieżki wyszukiwania klas. Jeśli nasza klasa odnosi się do innych klas w pakiecie `zwierzeta.ptaki` lub innym, odpowiednie ścieżki powinny być dostępne w zmiennej `CLASSPATH` lub podane jako parametr w trakcie kompilacji.

Kompilator Java jest inteligentniejszy od typowego kompilatora, bowiem zastępuje pewne działania narzędzia *make*. Na przykład *javac* porównuje czasy modyfikacji plików źródłowych oraz klas wszystkich klas, do których się odnosimy i kompiluje je ponownie tylko wtedy, gdy jest to potrzebne. Skompilowana klasa Javy zawiera informacje o tym, skąd pochodził plik źródłowy i jeśli nadal się tam znajduje, *javac* może go ponownie skompilować w razie potrzeby. Jeśli na przykład w poprzednim przykładzie odnosiliśmy się do klasy `zwierzeta.futrzaki.Hodowca`, Java poszuka pliku źródłowego *Hodowsca.java* w pakiecie `zwierzeta.futrzaki` i skompiluje go ponownie, gdy *Hodowca.class* jest starsza niż ten plik.

Jednak domyślnie Java sprawdza tylko te pliki źródłowe, które są wyraźnie podane w innym pliku źródłowym. Wobec tego gdy posiadamy aktualną klasę, która odnosi się do starszej klasy, ta starsza klasa nie zostanie ponownie skompilowana. Możemy wymusić przejście przez cały graf obiektów, używając opcji `-depend`, ale to znacznie

zwiększa czas kompilacji. Ta metoda nie pomoże jednak, gdy chcemy utrzymać aktualne biblioteki i inne zbiory klas, gdy się do nich nie odnosimy. Wtedy powinniśmy rozważyć użycie narzędzia *make*.

Warto wspomnieć o tym, że kompilator *javac* potrafi skompilować aplikację nawet wtedy, gdy istnieją tylko skompilowane wersje klas, do których się odnosimy. Nie potrzebujemy kodu źródłowego dla wszystkich obiektów. Pliki klas Javy zawierają wszystkie informacje o typach danych i sygnaturach metod, które znajdują się w plikach źródłowych, więc kompilowanie z pliku binarnego klasy jest tak samo bezpieczne jak z kodu źródłowego.

Pliki JAR (Java Archive)

Archiwa Javy to jakby aktówki Javy. To standardowa i przenośna metoda spakowania wszystkich części aplikacji Javy do niewielkiego pakunku w celu dystrybucji lub instalacji. Do plików *JAR* możemy upchać wszystko: pliki klas, obiekty serializowane, pliki danych, obrazy, dźwięki itp. W rozdziale 22. dowiemy się także, że plik *JAR* może zawierać jeden lub kilka podpisów cyfrowych, zapewniających niezmienioną postać archiwum. Podpis można zastosować do całego archiwum lub tylko niektórych plików.

System uruchamiania Javy czyta pliki *JAR* i wczytuje z nich klasy. Można więc spakować klasy archiwum i umieścić je w `CLASSPATH`, jak już wcześniej to przedstawiliśmy. Podobnie możemy postąpić z apletami, wymieniając plik JAR w atrybucie `ARCHIVE` znacznika `<APPLET>`. Pliki niebędące klasami (dane, obrazy itp.) także można wydobyć ze ścieżki wyszukiwania za pomocą metody `getResource()` (omówionej w rozdziale 11.). Przy użyciu tej funkcji kod nie musi wcale wiedzieć, czy zasób znajduje się w osobnym pliku czy stanowi część archiwum *JAR*. Gdy tylko dana klasa lub plik znajduje się w pliku *JAR*, archiwum w ścieżce wyszukiwania a w przypadku apletów na serwerze, zawsze możemy się do niego odnieść w standardowy sposób a system uruchamiania zajmie się resztą.

Kompresja plików

Elementy umieszczane w plikach *JAR* są kompresowane za pomocą standardowej kompresji *ZIP*[2]. Kompresja pozwala przyspieszyć pobieranie klas z sieci. Proste testy wykazują, że typowe klasy zmniejszają swój rozmiar o około 40% po kompresji. Pliki tekstowe, jak HTML lub ASCII, zawierające słowa kompresują się jeszcze bardziej, bo do 75% — czyli do jednej czwartej ich oryginalnego rozmiaru. (Jednak z drugiej strony obrazy zapisane w formatach stosujących własną kompresję nie zmniejszają rozmiaru).

Kompresja to nie jedyna zaleta plików *JAR* wykorzystywana podczas transportu w sieci. Umieszczenie wszystkich klas w jednym pliku pozwala pobrać wszystko za pomocą tylko jednej transakcji. Eliminacja narzutu powodowanego przez zgłoszenia HTTP to duża

[2] Pliki JAR są w pełni kompatybilne z archiwami ZIP dla użytkowników systemu Windows. Możemy nawet używać narzędzi, takich jak *pkzip*, do tworzenia i zarządzania prostymi plikami JAR, ale *jar* (narzędzie archiwizacji Javy) potrafi trochę więcej, o czym się wkrótce przekonamy — *przyp. aut.*

oszczędność, ponieważ poszczególne klasy są zwykle niewielkie, a złożone aplety mogą wymagać ich naprawdę wielu. Z drugiej strony jednak zwiększy się czas uruchamiania, gdyż cały plik musi zostać pobrany przed rozpoczęciem wykonywania programu.

Narzędzie jar

Narzędzie *jar* dostarczane wraz z pakietem SDK umożliwia tworzenie i odczytywanie plików *JAR*. Jego interfejs użytkownika jednak nie jest zbyt przyjazny. Jego działanie przypomina uniksowe polecenie *tar*. Jeśli stosowaliśmy to polecenie, poniższe szablony są nam znane.

`jar -cvf` *`plikJar ścieżka`* `[` *`ścieżka`* `] [ ... ]`
Tworzy *`plikJar`* zawierający ścieżkę(i).

`jar -tvf` *`plikJar`* `[` *`ścieżka`* `] [ ... ]`
Wyświetla zawartość *`plikJar`*, opcjonalnie wyświetla tylko podane ścieżki.

`jar -xvf` *`plikJar`* `[` *`ścieżka`* `]`
Wydobywa zawartość z *`plikJar`*, opcjonalnie wydobywa tylko podane katalogi.

W tych poleceniach litery `c`, `t` i `x` informują narzędzie, czy chcemy odpowiednio tworzyć, wyświetlać lub wydobywać zawartość archiwum. Litera `f` oznacza, że następnym parametrem będzie nazwa archiwum, na którym będziemy operować, natomiast `v` informuje narzędzie, że musimy być powiadamiani o wszystkich działaniach programu lub chcemy wiedzieć więcej o archiwum.

Pozostałe elementy linii poleceń (czyli wszystko poza literami przekazującymi do narzędzia, co ma robić i nazwą archiwum) są nazwami elementów archiwum. Jeśli tworzymy archiwum, są w nim umieszczane wszystkie wymienione katalogi i pliki. Kiedy wydobywamy pliki, zostaną pokazane tylko te pliki, które podaliśmy. (Jeśli nie podamy żadnych plików, narzędzie *jar* wydobędzie wszystkie pliki z archiwum).

Przypuśćmy, że skończyliśmy właśnie pracę nad nową grą Spaceblaser. Wszystkie pliki związane z grą znajdują się w katalogach drzewa. Klasy związane z grą umieściliśmy w katalogu *spaceblaster/game*, *spaceblaster/images* zawiera rysunki do gry, a *spaceblaster/docs* — dokumentację gry. Możemy to wszystko umieścić w archiwum *JAR* za pomocą polecenia:

```
% jar cvf spaceblaster.jar spaceblaster
```

Ponieważ poprosiliśmy o dokładne raportowanie, narzędzie wyświetli informacje o tym, czym aktualnie się zajmuje:

```
adding:spaceblaster/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/Game.class (in=8035) (out=3936) (deflated 51%)
adding:spaceblaster/game/Planetoid.class (in=6254) (out=3288) (deflated 47%)
adding:spaceblaster/game/SpaceShip.class (in=2295) (out=1280) (deflated 44%)
adding:spaceblaster/images/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/images/spaceship.gif (in=6174) (out=5936) (deflated 3%)
adding:spaceblaster/images/planetoid.gif (in=23444) (out=23454) (deflated 0%)
adding:spaceblaster/docs/ (in=0) (out=0) (stored 0%)
```

```
adding:spaceblaster/docs/help1.html (in=3592) (out=1545) (deflated 56%)
adding:spaceblaster/docs/help2.html (in=3148) (out=1535) (deflated 51%)
```

Narzędzie *jar* utworzyło plik *spaceblaster.jar* i dodało katalog *spaceblaster* wraz z jego podkatalogami i plikami do archiwum. W trybie dokładnego raportowania program podał zysk osiągnięty przez kompresję każdego z plików.

Możemy rozpakować archiwum za pomocą poniższego polecenia:

```
% jar cvf spaceblaster.jar
```

W podobny sposób wydobędziemy interesujący nas plik lub katalog:

```
% jar cvf spaceblaster.jar nazwapliku
```

Ale normalnie nie musimy rozpakowywać archiwów *JAR*, by skorzystać z ich zawartości. Narzędzia Javy automatycznie radzą sobie z archiwami. Zawartość pliku *JAR* możemy wyświetlić za pomocą polecenia

```
% jar tvf spaceblaster.jar
```

Oto wyjście zawierające listę wszystkich plików wraz z ich rozmiarami i datami utworzenia.

```
    0 Thu May 15 12:18:54 PDT 1997 META-INF/
 1074 Thu May 15 12:18:54 PDT 1997 META-INF/MANIFEST.MF
    0 Thu May 15 12:09:24 PDT 1997 spaceblaster/
    0 Thu May 15 11:59:32 PDT 1997 spaceblaster/game/
 8035 Thu May 15 12:14:08 PDT 1997 spaceblaster/game/Game.class
 6254 Thu May 15 12:15:18 PDT 1997 spaceblaster/game/Planetoid.class
 2295 Thu May 15 12:15:26 PDT 1997 spaceblaster/game/SpaceShip.class
    0 Thu May 15 12:17:00 PDT 1997 spaceblaster/images/
 6174 Thu May 15 12:16:54 PDT 1997 spaceblaster/images/spaceship.gif
23444 Thu May 15 12:16:58 PDT 1997 spaceblaster/images/planetoid.gif
    0 Thu May 15 12:10:02 PDT 1997 spaceblaster/docs/
 3592 Thu May 15 12:10:16 PDT 1997 spaceblaster/docs/help1.html
 3148 Thu May 15 12:10:02 PDT 1997 spaceblaster/docs/help2.html
```

Pliki-wykazy archiwum JAR

Zauważ, że polecenie *jar* automatycznie dodaje katalog o nazwie *META-INF*. Katalog ten przechowuje pliki opisujące zawartość archiwum *JAR*. Znajduje się tam przynajmniej plik *MANIFEST.MF*. Plik ten może przechowywać nazwy plików w archiwum wraz z przypisanymi im atrybutami.

Wykaz to plik tekstowy zawierający zbiór linii w formie *słowokluczowe:wartość*. W Javie 1.2 i późniejszych plik ten jest domyślnie prawie pusty i można w nim znaleźć tylko informacje o wersji pliku *JAR*:

```
Manifest-Version: 1.0
Created-By: 1.2.1 (Sun Microsystems Inc.)
```

W rozdziale 22. omówimy *podpisywane* pliki *JAR*. Gdy podpiszemy plik *JAR*, skrót podpisu cyfrowego dla każdego pliku w archiwum jest umieszczany w pliku-wykazie. Wygląda to następująco:

```
Name: com/oreilly/Test.class
SHA1-Digest: dF2GZt8G11dXY2p4olzzIc5RjP3=
...
```

W przypadku podpisanych plików *JAR* katalog *META-INF* przechowuje pliki podpisów cyfrowych dla elementów archiwum. W Javie 1.1 skróty były zawsze dodawane do plików *JAR*. Ponieważ są one tak naprawdę wymagane tylko dla podpisanych plików, to archiwa w Javie 1.2 lub późniejszych je pomijają.

Możemy dodać własne informacje do opisów wykazu, podając nazwę własnego wykazu w trakcie tworzenia archiwum. Jest to dobre miejsce do przechowywania prostych atrybutów plików w archiwum, na przykład informacji o wersji i autorze.

```
Name: spaceblaster/images/planetoid.gif
Wersja: 42.7
Temperament-grafika: nastrojowy
```

Aby dodać te informacje do wykazu archiwum, musimy umieścić powyższy tekst w pliku o nazwie *wykaz.mf* i użyć następującego polecenia *jar*.

```
% jar -cvmf wykaz.mf spaceblaster.jar spaceblaster
```

Pojawia się dodatkowa opcja `m`, która instruuje narzędzie, że dodatkowe informacje do wykazu powinno odczytać z pliku podanego w linii poleceń. Ale jak narzędzie ma zidentyfikować, który to plik? Ponieważ `m` znajduje się przed `f`, to spodziewa się znaleźć nazwę pliku wykazu przed nazwą archiwum. Jeśli wydaje się to niewygodne, to prawda; ale wystarczy podać pliki w złej kolejności, a *jar* zrobi wszystko na odwrót.

Chociaż atrybuty nie są automatycznie dostępne w kodzie aplikacji, łatwo je odczytać za pomocą klasy `java.util.jar.Manifest`.

Więcej przykładów dodawania informacji do plików *JAR* umieściliśmy w rozdziale 21. Interfejs JavaBeans wykorzystuje wykaz do określenia, które klasy to klocki, przy użyciu atrybutu `Java-Bean`. Informacja ta jest wykorzystywana przez zintegrowane środowiska programistyczne, które wczytują JavaBeans z plików *JAR*.

Tworzenie wykonywalnych plików JAR

Poza atrybutami istnieje kilka specjalnych wartości, które możemy umieścić w pliku wykazu. Jedną z nich jest `Main-Class`, która pozwala określić klasę zawierającą główną metodę `main()` aplikacji znajdującej się w archiwum *JAR*.

```
Main-Class: com.oreilly.Game
```

Jeśli dodaliśmy plik wykazu z taką zawartością do archiwum (używając opcji `m`), możemy uruchomić aplikację bezpośrednio z archiwum

```
% java -jar spaceblaster.jar
```

W systemie Windows i innych graficznych środowiskach wystarczy po prostu kliknąć plik *JAR*, by go uruchomić. Interpreter szuka wartości `Main-Class` w wykazie. Jeśli ją znajdzie, wczytuje podaną klasę i uruchamia jej metodę `main()`.