ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2000
Seria: INFORMATYKA z. 38 Nr kol. 1451

Adam ŚWITOŃSKI
Politechnika Śląska, Instytut Informatyki

WYBRANE ASPEKTY TWORZENIA APLIKACJI BAZODANOWYCH W JAVIE

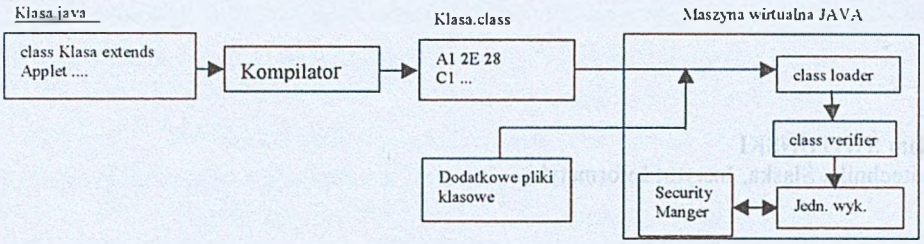
Streszczenie. Artykuł omawia wybrane problemy, z jakimi styka się twórca aplikacji bazodanowych w Javie. Poruszono tutaj następujące kwestie: wybór interfejsu oraz sterowników JDBC, wielowątkową realizację zadań pobierających dane z bazy, wykorzystanie komponentów pakietu Swing do prezentacji danych oraz kwestie bezpieczeństwa w przypadku aplikacji internetowych.

SELECTED ASPECTS OF CREATION JAVA DATABASE APPLICATIONS

Summary. The article describes selected problems which occur during process of creation database applications in Java. There are raised following subjects: choice of JDBC version and drivers, multithreaded architecture of Java programs, useful Swing components for this kind of applications and security issues for internet applications.

1. Wstęp

Rok 1995 to debiut nowego narzędzia programistycznego, jakim jest język Java. Jej twórca, firma Sun Microsystems, skupił się przede wszystkim na osiągnięciu uniwersalności w zakresie platform uruchomieniowych. Aby to osiągnąć, wprowadził pojęcia wirtualnego komputera, tak zwanej Maszyny Wirtualnej Java (ang. Java Virtual Machine – JVM). Najprostsza definicja JVM podaje, że jest to jednostka centralna wraz ze zbiorem niezbędnego oprogramowania, zdolna do wykonywania zdefiniowanego zbioru instrukcji w postaci kodów bajtowych. Uzyskanie ich jest wynikiem kompilacji programu napisanego w języku Java. JVM jest tworzona głównie na drodze programowej, gdzie stanowi ona rozszerzenie bazowego systemu operacyjnego.



Rys.1. Proces tworzenia i uruchamiania programów w Javie
Fig.1. The process of creation and execution Java programs

Sama architektura Maszyny Wirtualnej Java określa kilkustopniowy proces uruchamiania programu, do którego należą między innymi class loader (wgrywa odpowiednie pliki z definicjami klas) oraz class verifier (weryfikuje poprawność tych plików). Oddzielny mechanizm bezpieczeństwa stanowi Security Manager, który faktycznie kontroluje dostęp do wszystkich zasobów komputera.

Obecnie głównym środowiskiem dla aplikacji pisanych w Javie jest heterogeniczna sieć Internet, a dokładnie jej serwis WWW, gdzie obecność appletów przyniosła prawdziwą rewolucję. Jednak również stosowanie Javy dla samodzielnych aplikacji jest już w pełni powszechne.

2. Standard JDBC

Wymogiem koniecznym dla osiągnięcia pełnej funkcjonalności w architekturze aplikacji typu klient-serwer Javy było stworzenie odpowiedniego interfejsu do realizacji połączeń z bazami danych. W pierwszej swojej wersji standard Javy nie przewidywał takiej możliwości, co spowodowało, że wielcy producenci baz danych, jak Oracle, Informix czy Microsoft zaczęli sami opracowywać własne klasy i metody do tego celu. Powstało wiele różnorodnych podejść rozwiązujących ten problem. Dlatego było niemal koniecznością opracowanie uniwersalnego interfejsu, odpowiedniego dla wszystkich baz danych. Sun Microsystems wprowadził taki interfejs pod postacią standardowego pakietu *java.sql*. Określany on jest poprzez standard JDBC (ang. Java Database Connectivity) definiujący zestaw klas oraz interfejsów API, zapewniających jednolity pomost komunikacyjny z relacyjnymi bazami danych, na poziomie języka programowania. Analogicznie jak ODBC również JDBC jest oparty na standardzie X/Open SQL CLI (Call-Level Interface).

Patrząc od strony programisty, każdy dostęp do danych serwera realizowany poprzez interfejs JDBC odbywa się w kilku etapach:

Załadowanie klas sterownika:

Można tego dokonać na dwa sposoby: poprzez jego rejestrację metodą DriverManager.registerDriver lub bezpośrednie wgranie klasy sterownika (w przykładzie wykorzystano oryginalny sterownik thin dla serwera Oracle).

```
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
  } catch(ClassNotFoundException e)
     { JOptionPane.showMessageDialog(parent,e,"JDBC Driver Error",
       JOptionPane.ERROR_MESSAGE);
       System.out.println(e);
     }
```

Nawiązanie połączenia:

Po załadowaniu odpowiedniego sterownika możemy juz przejść do etapu nawiązania połączenia. Dokonuje się tego za pomocą statycznej metody getConnection z klasy DriverManager:

```
try {
  connection=DriverManager.getConnection("jdbc:oracle:thin:@"
    +address+":"+port+":"+sid,user,password);
   statement=connection.createStatement();
 }
catch(SQLException e) {
   JOptionPane.showMessageDialog(parent,e,"Connection Error",
       JOptionPane.ERROR_MESSAGE);
   System.out.println(e);
}
```

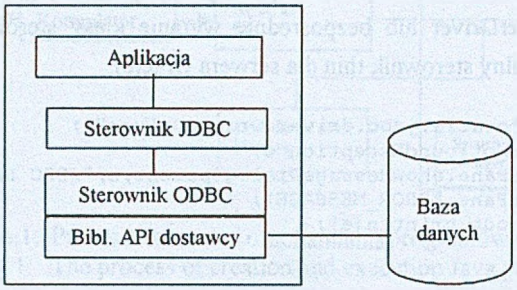
Wykonanie zapytania i pobranie jego wyników:

Etap ten można wykonać wielokrotnie dla wcześniej ustalonego połączenia.

```
try {
 ....
String insertstring="insert into tablica1 values("+zmienna1
  +","+zmienna2+","+zmienna3+",'+"',"+zmienna4 +")";
statement.executeUpdate(insertstring);
String query="select kol1 from tablica2 where kol2='"+zmienna5"+" and
kol2='"+zmienna6;
PreparedStatement ps=connection.prepareStatement(query);
   rs=ps.executeQuery(); //wykonanie zapytania
   if(rs.next()) //pobranie wynikow
    zmienna7=rs.getString("kol1");
 } catch(SQLException e) {
   System.out.println(e);
 }
```

Zgodnie z generalnym założeniem Javy, jakim jest uniwersalność, również interfejs JDBC jest niezależny od platformy bazodanowej. Jednak aby można było wykorzystać jego możliwości, wymagana jest obecność rodzimego sterownika, który musi implementować odpowiednie interfejsy oraz klasy abstrakcyjne pakietu *java.sql* (pierwszy etap). Ze względu na swoją wewnętrzną architekturę obecnie dostępne sterowniki możemy podzielić na cztery kategorie omówione w punktach 2.1-2.4.

2.1. Pomost JDBC/ODBC (typ I)



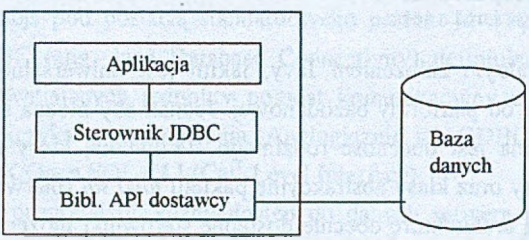
Rys. 2. Pomost JDBC-ODBC
Fig. 2. JDBC-ODBC Bridge

Ten typ dokonuje konwersji odwołań JDBC na ODBC, przekazując je dalej do odpowiedniego sterownika ODBC. Podstawową zaletę tego rozwiązania stanowi fakt, że aplikacje mają dostęp do wszystkich baz danych posiadających sterownik ODBC, który jest standardem bardzo popularnym, a więc wykorzystywanym przez większość systemów bazodanowych. Jednak jest on obciążony znacznymi transformacjami dla dokonywanych operacji. Każde odwołanie najpierw musi przejść przez pomost z JDBC na ODBC, a następnie z ODBC na rodzime API klienta bazy danych. Dodatkowym utrudnieniem jest wymóg zainstalowania odpowiednich *driver'ów* ODBC na stacjach klienckich. Uniemożliwia to praktycznie stosowanie go dla aplikacji działających w środowisku WWW, gdzie nie możemy przyjąć założenia o obecności sterowników ODBC na przyłączonych komputerach.

Implementacja takiego sterownika nie może być całościowo zrobiona w Javie, gdyż sam interfejs ODBC wykorzystuje własności języka C, jak wskaźniki, co wymusza na jego twórcach zastosowanie metod rodzimych dla określonego systemu operacyjnego.

Od wersji JDK 1.2.1 *driver* tego typu jest zintegrowany z maszyną wirtualną, co znacznie ułatwia jego zastosowanie.

2.2. Rodzime API, sterownik częściowo implementowany w Javie (typ II)

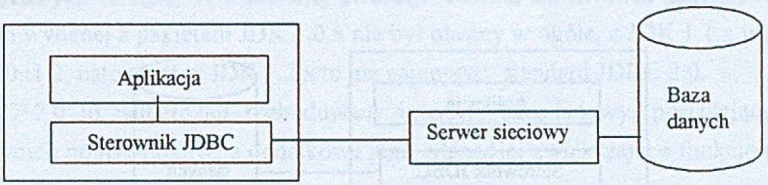


Rys. 3. Rodzime API, sterownik częściowo implmentowany w Javie
Fig. 3. Native API, partly Java driver

Rozwiązanie tego typu posiada dwuwarstwową architekturę, gdyż sterownik JDBC wymaga biblioteki rodzimej, umożliwiającej porozumiewanie się z określonym serwerem bazy danych w charakterystycznym dla niego protokole. Sterowniki te pisane są tylko częściowo w Javie, gdyż muszą korzystać z warstwy języka biblioteki rodzimej, aby móc się odwoływać do jej funkcji, a na dzień dzisiejszy niestety biblioteki rodzime nie są implementowane w Javie. Z tego też powodu praktycznie na każdą platformę sprzętową musi istnieć oddzielna jego implementacja.

Sterownik ten dzięki uproszczeniu architektury, w porównaniu do poprzedniego, poprzez likwidację warstwy ODBC, ma dużo większe możliwości, jeśli chodzi o szybkość działania. Dalej jednak posiada niekorzystną cechę, jaką jest konieczność zainstalowania bibliotek klienckich na stacjach roboczych, co znacznie utrudnia jego zastosowanie w przypadku aplikacji internetowych.

2.3. Protokół sieciowy, sterownik całościowo implementowany w Javie (typ III)



Rys. 4. Protokół sieciowy, sterownik całościowo impl. w Javie
Fig. 4. JDBC-Net, pure Java driver

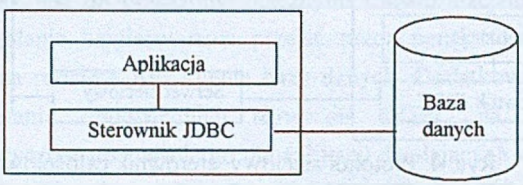
W tym wypadku odwołania JDBC są tłumaczone na niezależny od baz danych protokół sieciowy. Odwołania te jednak nie trafiają bezpośrednio do serwera bazy danych, lecz do tak zwanego serwera sieciowego, zwanego inaczej agentem JDBC, który dopiero dokona ich translacji na wewnętrzny protokół bazodanowy. Sterownik taki wymaga więc odpowiedniego agenta, którego implementacja może nastąpić również w Javie, z wykorzystaniem sterowników typu I, II lub IV, bądź w innych językach z użyciem ODBC, lub bibliotek rodzimych. Naturalnie musi być on uruchomiony na określonej stacji w sieci, gdzie bezpośrednio nastąpi połączenie z poziomu Javy. Dodatkowe parametry mogą wskazywać na docelowy serwer bazy danych lub w przypadku gdy mamy do czynienia z agentem dedykowanym, wartość ta będzie domyślna.

Agent JDBC może spełniać również dodatkowe funkcje, jak ułatwienia w logowaniu, bądź nawet implementować cały proces mapowania użytkowników, co zwiększy bezpieczeństwo systemu (użytkownik nie będzie musiał znać bezpośredniego hasła do bazy danych, lecz tylko do serwera pośredniego), wspierać podręczną pamięć notatnikową przyspieszającą wykonywanie zapytań powtarzających się, stanowić serwer *proxy* w

przypadku zabezpieczeń typu *firewall* (tylko *firewall'e* zamontowane przed serwerem bazy danych) oraz jeszcze kilka innych.

Sama architektura tego *driver'a* niesie za sobą dość dużą złożoność natury komunikacyjnej, gdyż każde odwołanie musi przejść poprzez serwer pośredni. Wiąże się to z faktem znacznego obniżenia jego wydajności w porównaniu z poprzednimi sterownikami. Jednak posiada ona również pewną zaletę: protokół sieciowy może być stosunkowo prosty w porównaniu z rodzimym protokołem serwera bazodanowego, gdyż musi uwzględniać tylko odwołania generowane poprzez klasy pakietu *java.sql*, które generalnie ograniczają się do zapytań SQL'a oraz faz nawiązywania połączenia. Tak uproszona jego struktura implikuje mniejsze rozmiary sterownika oraz w pewnym, niewielkim stopniu poprawia jego efektywność. Może to być zauważalne w przypadku aplikacji wykonującej dużą liczbę krótkich zapytań (w sensie zbioru wynikowego) w sieci o małej przepustowości. Bardzo istotną jego zaletą jest fakt, że jest on całościowo implementowany w Javie.

2.4. Rodzimy protokół, sterownik całościowo implementowany w Javie (typ IV)



Rys. 5. Rodzimy protokół, sterownik całościowo impl. w Javie
Fig. 5. Native protocol, pure Java driver

Tego typu sterowniki dokonują konwersji odwołań JDBC bezpośrednio na rodzimy protokół używany przez określoną bazę danych. Praktycznie mogą one zostać wydane jedynie poprzez producenta określonego serwera, gdyż faktycznie tylko on zna szczegóły techniczne własnego protokołu rodzimego. Są one całościowo implementowane w Javie.

Driver taki, ze względu na najmniejszą złożoność konwersji odwołań, które odbywają się bez pośrednictwa żadnej warstwy przejściowej, ma największe możliwości, jeśli chodzi o wydajność. Posiada jednak, w stosunku do poprzednika, jedną stosunkowo niekorzystną cechę, jaką są jego rozmiary.

Wszystkie typy sterowników mają swoje wady oraz zalety i powinny być stosowane w zależności od wymaganej funkcjonalności. Ogólnie można podać kilka reguł:

- Dla appletów przeznaczonych dla licznej grupy użytkowników Internetu typy I oraz II należy rozważać tylko w wypadku, gdy nie ma sterowników całościowo zaimplementowanych w Javie.

- Rozważając wybór typu III lub IV należy uwzględnić charakter generowanych zapytań oraz przepustowość sieci: Dla zapytań generujących większe zbiory wynikowe oraz gdy wymagamy dużej szybkości działania preferowany powinien być typ IV. W przypadku gdy mamy do czynienia z krótkimi zapytaniami (w sensie zbioru wynikowego) oraz małą przepustowością sieci, bardziej odpowiedni wydaje się być typ III.
- Dla typu III należy rozważyć możliwości agenta JDBC i ewentualnie ocenić korzyści z jego zastosowania.
- W przypadku gdy wymagana jest wysoka wydajność, pod uwagę powinny być brane typy II oraz IV.

3. JDBC 2.0

Java rozwija się niesłychanie dynamicznie i już obecnie posiada kilkanaście wersji nie do końca zgodnych ze sobą. W trakcie jej ewolucji rozwijał się również interfejs JDBC, gdzie dla wersji wydanej z pakietem JDK 1.0.x nie był obecny w ogóle, z JDK 1.1.x występował w wersji 1.0 i1.2, natomiast w JDK 1.2.x to już najnowszy standard JDBC 2.0.

JDBC 2.0 to najbardziej rozbudowany interfejs bazodanowy, posiadający wszystkie cechy swoich poprzedników, a dodatkowe jego własności zwiększające funkcjonalność oraz poprawiające wydajność to:

- Przesuwalne obustronnie kursory zbiorów wynikowych ResultsSet: Możliwe jest przesuwanie się zarówno w przód, jak i tył do określonego wiersza: pierwszy, ostatni, ostatnio zapamiętany, relatywnie o zadaną wartość.
- Modyfikacje wsadowe (ang. batch updates) dla obiektów klas Statement i PreparedStatement: Możliwe jest wykonywanie kilku modyfikacji w jednej fazie, zamiast wysyłania każdej modyfikacji osobno. Zastosowanie tej techniki znacznie zwiększa wydajność aplikacji dla wielokrotnych modyfikacji.
- Programowalne modyfikacje: Po pobraniu danych z bazy do obiektu klasy ResultsSet możemy zmodyfikować wartości odpowiednich pól w rozpatrywanym rekordzie poprzez metody updateXXX, a następnie uaktualnić tę zmianę w bazie. Modyfikacja więc nie musi się koniecznie odbywać na drodze instrukcji SQL'wych, lecz również poprzez odpowiednie metody wywoływane z poziomu języka.
- Ładowanie większej liczby rekordów w jednym etapie, co znacznie przyspiesza sam proces pobrania danych oraz określenie kierunku ładowania wierszy.
- Pobranie wartości w określonej kolumnie w trybie strumienia znakowego.
- Pełna precyzja dla typu BigDecimal.
- Obsługa różnych stref czasowych identyfikowanych poprzez obiekt klasy Calendar.

- Obsługa typów danych standardu SQL3: Do tego celu stworzone zostały specjalne klasy, a do klas już istniejących realizujących proces przetwarzania danych dodano nowe metody. Dodatkowe typy to:
  - Typy do przechowywania dużych obiektów: BLOB – binarny, CLOB – znakowy.
  - Typy obsługujące rodzaje danych fizycznie rezydujące w bazie, wykorzystane między innymi w procesie mapowania: Struct, Ref, SQLData.
  - Typ tablicowy ARRAY przechowujący cały zbiór zwracanych w wyniku zapytania wartości.

Naturalnie dla skorzystania z możliwości nowego standardu konieczny jest odpowiedni sterownik go implementujący. Choć powstały już takie *driver*'y dla ważniejszych systemów, jednak w większości przypadków nie są one ogólnodostępne. Wyjątek stanowi tutaj pomost JDBC/ODBC.

4. Komponenty Swinga

Dla twórców aplikacji istotnym elementem jest funkcjonalność interfejsu użytkownika, która w znacznym stopniu wpływa na szybkość jej tworzenia. Nowe możliwości w tej dziedzinie daje pakiet Swing stanowiący główną część oprogramowania JFC (Java Foundation Classes), rozszerzającego standardowe Core API Javy o zbiór nowych klas. W jego efekcie powstał między innymi cały pakiet *javax.swing*, zawierający nową grupę komponentów interfejsu graficznego, który całkowicie zaimplementowany został w Javie. Zawiera on zarówno komponenty już istniejące w pakiecie AWT, jak i dodatkowe komponenty wyższego poziomu zwiększające funkcjonalność interfejsu użytkownika. Z punktu widzenia aplikacji bazodanowych najistotniejsze elementy będące jego integralną częścią, natomiast nie mające swoich odpowiedników w pakiecie AWT to:

- JTabel: Komponent definiujący dwuwymiarową tabelkę, prezentującą dane różnych typów, jak tekst czy obraz. Posiada wiele cech, które mogą zostać dostosowane do potrzeb aplikacji, jak jej rozkład (interfejsy JTabelModel i JTabelColumnModel), wygląd poszczególnych komórek (TabelCellRenderer), edytor (TabelCellEditor), sposób zaznaczania (ListSelectionModel), nagłówki (TabelHeader) oraz jeszcze parę innych. Naturalnie zaimplementowane są domyślne instancje wszystkich interfejsów, które w zależności od wymagań mogą być zmodyfikowane.
- JScrollPane: Specjalistyczny kontener zarządzający częścią widoczną prezentowanego komponentu identyfikowanego poprzez klasę JComponent. W przypadku gdy jego rozmiary będą większe niż dostępne pole widokowe, w zależności od potrzeb, uaktywnione zostaną suwaki – poziomy i pionowy, umożliwiające dostęp do wszystkich

jego części. Głównie wykorzystywany jest do prezentacji dużej liczby danych, jak tabela z wieloma wierszami.

- JTabbedPane: Komponent umożliwiający przełączanie się pomiędzy grupą wcześniej zdefiniowanych paneli poprzez wybranie określonej zakładki. Poszczególne zakładki są identyfikowane przez indeksy odpowiadające kolejności, w jakiej zostały one dodane. Sposób selekcji zakładek definiuje interfejs SingleSelectionModel, którego domyślną implementację można zmodyfikować. W przypadku aplikacji bazodanowych często bywa pomocny w prezentacji danych ze sobą powiązanych, lecz dotyczących odrębnych logicznie dziedzin (np. pracownik – dane osobiste, staż pracy, nagrody etc.).
- ProgressMonitor, JProgressBar: Okienko dialogowe oraz komponent wewnętrzny pozwalające monitorować postęp wykonania czynności długotrwałych. Wśród parametrów wejściowych podajemy całkowite wartości początkowe i końcowe dla określonej operacji, natomiast w trakcie jej trwania aktualizujemy poprzez odpowiednie metody obecny stan jej wykonania. W efekcie zostanie wyświetlony odpowiedni element (dialog lub komponent wewnętrzny) ukazujący relatywny czas do zakończenia operacji. Klasy te są wykorzystywane podczas ładowania danych z serwera.
- JPasswordField Pole tekstowe umożliwiające edycję pojedynczej linii tekstu z jednoczesnym ukrywaniem swojej zawartości. Głównie stosowane jest do wprowadzania hasła dostępu do serwera.

5. Wielowątkowość

Aplikacja jednowątkowa pozwala w danej chwili czasu na wykonywanie tylko jednej określonej czynności. W przypadku zadań czasochłonnych, lecz nie wymagających dużego zaangażowania procesora, których przykładem może być ładowanie danych z sieci, takie rozwiązanie jest wysoce nieefektywne.

Jednym z aspektów wykorzystania wielowątkowości jest podtrzymanie aktywności interfejsu użytkownika podczas wykonywania czynności długotrwałych. Do takich na pewno możemy zaliczyć pobieranie wyników zapytania z serwera bazy danych znajdującego się na odrębnym komputerze w sieci. Dlatego wszystkie metody, które bezpośrednio operują na bazie danych, powinny zostać wykonane w odrębnym wątku. Problem realizacji zadań w oddzielnych wątkach w Javie należy rozważyć dla dwóch różnych przypadków, w zależności od tego, czy korzystamy z klas pakietu Swing do stworzenia interfejsu graficznego, czy nie.

Gdy interfejs został zaprojektowany wyłącznie za pomocą standardowego pakietu AWT, należy stworzyć nowy wątek poprzez obiekt klasy Thread, implementujący metodę run, a następnie ją wywołać:

```
Thread thread=new Thread(new Runnable() {
  public void   run() {
    try {
     String query=" select koll from tablica1"
     PreparedStatement ps=connection.prepareStatement(query);
     ResultSet rs=ps.executeQuery(); //wykonanie zapytania
     for(int i=0;rs.next();i++) //pobranie wynikow
      KOL1[i]=rs.getString("kol1");
    } catch(SQLException e) {
     System.out.println(e);
    }
  }
  presnetdata(); //prezentacja danych
 }
)
thread.run();
```

W przypadku komponentów Swing problem się już nieco komplikuje. Teraz chodzi już nie tylko o aktywność funkcjonalną, lecz przede wszystkim wizualną. Jest to spowodowane faktem, że są one całościowo kontrolowane poprzez maszynę wirtualną, bez wykorzystania komponentów bazowego systemu operacyjnego. Dlatego wstrzymanie wątku głównego z obsługą podstawowych zdarzeń sprawi między innymi przerwanie procesu odświeżania, a to spowoduje, że komponenty przestaną być aktywne nawet w sensie wizualnym.

Tworząc aplikację opartą na interfejsie Swinga, należy pamiętać o fundamentalnej zasadzie, że wszystkie operacje, które mogą mieć wpływ na komponenty lub korzystają z metod podających ich aktualny stan, powinny zostać wykonane w głównym wątku obsługi zdarzeń. Jest to spowodowane specyficzną budową wewnętrzną klas pakietu Swing. Wyjątek stanowi tutaj jedynie nieliczna grupa metod, które są określane jako thread save. Jednak takie zaprojektowanie wątku, aby nie korzystał on z zabronionych metod, na ogół nie jest możliwe lub przynajmniej jest kłopotliwe. Dlatego naprzeciw wychodzą nam tutaj dwie statyczne metody klasy SwingUtilities, umożliwiające wykonanie ciągu operacji określonego przez argument w wątku głównym :

- **invokeLater**: metoda zwraca sterowanie bez oczekiwania na zakończenie wykonania.
- **invokeWait**: oczekuje na zakończenie, przed zwróceniem sterowania.

Tak więc w przypadku wyżej zaprezentowanego przykładu zamiast linii

```
presnetdata(); //prezentacja danych
```

powinien znaleźć się następujący fragment:

```
invokeLater(new Runnable() {
  public void run() {
   presnetdata(); //prezentacja danych
  }
 }
)
```

SwingWorker:

Sun w odpowiedzi na liczne prośby użytkowników opracował specjalną klasę o nazwie SwingWorker, która jest pomocna przy tworzeniu oraz obsłudze nowych wątków, dla aplikacji korzystających z interfejsu Swing. Obecnie nie jest ona częścią integralną pakietu, jednak jest ogólnodostępna w Internecie i stanowi jego naturalne rozszerzenie. Schematycznie jej działanie możemy przedstawić następująco: w pierwszym etapie wykonuje zbiór czynności nie powiązanych bezpośrednio z komponentami graficznymi (np. dokonuje obliczeń, ładuje plik z sieci etc.), a na zakończenie poprzez wywołanie metody *invokeLater*, ciąg operacji w wątku głównym, które mogą już operować na elementach interfejsu graficznego. Klasa ta jest abstrakcyjna, gdyż nie implementuje metod *construct* oraz *finished*, odpowiedzialnych za wykonanie dwóch wyżej wymienionych etapów.

Przykład tym razem będzie się prezentował następująco:

```
final SwingWorker worker=new SwingWorker() {
 public Object construct() {
  try {
   String query=" select koll from tablica1"
   PreparedStatement ps=connection.prepareStatement(query);
   ResultSet rs=ps.executeQuery(); //wykonanie zapytania
    for(int i=0;rs.next();i++) //pobranie wynikow
     KOL1[i]=rs.getString("koll");
   } catch(SQLException e) {
    System.out.println(e);
   }
  }
  public void finished() { //operacje na komponentach GUI
   presentdata();
  }
}
```

6. Bezpieczeństwo aplikacji internetowych

Kwestie związane z bezpieczeństwem dla aplikacji bazodanowych Javy działających w Internecie należy rozpatrzyć w dwóch różnych punktach: ograniczenia, jakie wprowadza Java w tym obszarze oraz dodatkowe zagrożenia w stosunku do alternatywnych skryptów CGI.

Jednym z mechanizmów chroniących komputery podłączone do Internetu przed niepożądanym dostępem do jego zasobów przez applety Javy jest Security Manager. To on faktycznie decyduje o tym, co może i pod jakimi warunkami, a czego nie może w ogóle zrobić program uruchomiony na platformie maszyny wirtualnej. I tak, cały system wejścia/wyjścia jest kontrolowany przez JVM, która zanim zdecyduje się na wykonanie określonej operacji, najpierw zwróci się z pytaniem do Security Managera o pozwolenie. Szczególnym tego przypadkiem jest ingerencja w możliwość otwierania nowych gniazdek dla połączeń sieciowych, gdzie standardowo Security Manager zezwala appletom tylko na połączenia z adresami, z których pochodzą. Dodatkowo zabroniony jest odczyt i zapis do wszystkich zasobów lokalnych, jak system plików czy drukarka.

Osobny problem stanowią metody rodzime w aplikacjach Javy. Faktycznie żadna wykonywana przez nie operacja nie jest kontrolowana poprzez zaimplementowany system bezpieczeństwa, dlatego rzeczą konieczną było zabronienie ich używania w appletach. Ich stosowanie również nie ma sensu z punktu widzenia niezależności co do bazowych systemów operacyjnych, która stanowi podstawę dla aplikacji działających w Internecie.

Istnieje oczywiście możliwość modyfikacji parametów Security Managera w celu złagodzenia, ewentualnie zaostrzenia restrykcji, jednak musi ona zostać dokonana bezpośrednio poprzez użytkownika, a nie może natomiast przez applet. Stąd wynika jedno zasadnicze ograniczenie: applety dla standardowych ustawień przeglądarki będą mogły się połączyć jedynie z serwerami baz danych fizycznie rezydującymi na tym samym komputerze, z którego pochodzą. Mniejszą niedogodnością wydają się być problemy związane z wydrukami ewentualnych raportów, brakiem możliwości ich zapisu do plików, czy stosowaniem metod rodzimych.

Porównując kwestie bezpieczeństwa bazodanowych appletów Javy oraz alternatywnych dla nich skryptów CGI, należy zwrócić uwagę na dwie dość istotne różnice: Skrypt CGI wykonuje się na serwerze HTTP, tam dokonując bezpośrednio połączeń z bazą danych, a następnie generuje i zwraca stronę WWW. Takie podejście pozwala ukryć przed użytkownikiem parametry identyfikujące dostęp do bazy (np. hasło), które mogą być przechowywane poprzez serwer HTTP. Co za tym idzie, użytkownik będzie miał dostęp tylko do usług oferowanych poprzez skrypt CGI, do którego dostęp może być również restrykcyjny, nie natomiast bezpośrednio do bazy. Inny problem stanowią zabezpieczenia typu *firewall*, gdzie dla większości takich systemów zabrania się komunikacji poprzez

niestandardowe porty, natomiast prawie zawsze otwarte są porty obsługi WWW. Restrykcje mogą również nastąpić w wyższej warstwie, gdzie dostęp do zasobów sieci będzie kontrolowany przez serwery *proxy*. Jednak również tutaj nie należy spodziewać się zdefiniowanych *proxy* do baz danych, w przeciwieństwie do niemalże zawsze obecnych HTTP *proxy*. W efekcie tych ograniczeń działanie prawidłowo napisanych appletów będzie uzależnione od stosowanego systemu bezpieczeństwa ich użytkowników.

7. Podsumowanie

Java to obecnie najbardziej dynamicznie rozwijający się język programowania, mający swoje platformy uruchomieniowe praktycznie dla wszystkich rodzajów powszechnie stosowanych komputerów. Co za tym idzie jest to język uniwersalny, a stworzone za jego pomocą programy można bez żadnych modyfikacji uruchamiać na wielu platformach sprzętowych.

Początkowo jej zastosowanie ograniczało się do prostych appletów ożywiających witryny WWW. Jednak z czasem, wraz z rozwojem języka, zaczęły powstawać coraz poważniejsze aplikacje, przeznaczone nie tylko dla potrzeb Internetu. Wśród nich bardzo ważną oraz liczną grupę stanowią aplikacje bazodanowe. Podczas procesu ich tworzenia programista napotyka na szereg problemów, których optymalne rozwiązanie wymaga poznania wszystkich alternatyw. Najczęściej pojawiające się problemy to:

- **Wybór standardu oraz sterowników JDBC:** Zakładając dostęp do serwera bazy danych poprzez interfejs JDBC, który faktycznie stanowi jedyną alternatywę, należy wybrać odpowiedni sterownik go implementujący. Każdy sterownik przypisany jest do jednej z czterech kategorii, która to w zasadniczy sposób określa jego własności. W przypadku gdy możliwości standardu JDBC w wersji 1.0 są niewystarczające, można rozważyć zastosowanie nowszego standardu JDBC 2.0. Należy jednak pamiętać, że wymaga on zarówno nowszych sterowników, jak i również odpowiedniej maszyny wirtualnej.
- **Wybór pakietu do stworzenia interfejsu graficznego użytkownika:** Ważnym elementem każdej aplikacji jest interfejs graficzny użytkownika. Java, w tym obszarze, daje dwie alternatywy: interfejs stworzony za pomocą pakietu AWT lub z użyciem pakietu Swing. Można również stworzyć interfejs z użyciem obydwu, jednak z punktu widzenia jednolitości aplikacji na pewno nie jest to rozwiązanie korzystne. Swing charakteryzuje się większą funkcjonalnością swoich komponentów, natomiast istotną różnicą w porównaniu do AWT jest fakt, że nie korzysta z komponentów bazowego systemu operacyjnego, gdyż całkowicie został zaimplementowany w Javie.

- **Wielowątkowość:** Aplikacje bazodanowe, aby pobrać dane z bazy, muszą łączyć się z serwerami. Gdy proces taki jest długotrwały, konieczne jest przeniesienie go do odrębnego wątku, aby uniknąć niekorzystnego efektu zawieszania się aplikacji na czas jego trwania. Wielowątkowa architektura aplikacji Javy jest zależna od tego, czy wykorzystywany jest pakiet Swing do stworzenia interfejsu graficznego, czy został on całościowo zbudowany z komponentów pakietu AWT.
- **Kwestie bezpieczeństwa:** W przypadku aplikacji internetowych, zwanych inaczej appletami, Java nakłada pewne ograniczenia ze względu na zminimalizowanie zagrożeń, mogących się pojawić podczas niepożądanych dostępów do zasobów komputera. Dlatego należy rozważyć te ograniczenia i ocenić, czy nie przeszkodzą one w osiągnięciu pełnej funkcjonalności aplikacji.

LITERATURA

1. Sun Microsystems „JDK™ 1.2.2 Documantation", Sun Microsystems, 1999.
2. „Java Tutorial", Sun Microsystems, http://java.sun.com.
3. „The Java Developers Connection", Sun Microsystems, http://developer.java.sun.com.
4. „Proprieaty Java Database API" – WebLogic Inc. – http://www.weblogic.com.
5. „Java Products and APIs", Sun Microsystems, http://java.sun.com.
6. „Javaworld Press" , Web Publishing Inc . http://www.javaworld.com.
7. Laurence Vanhelsuwe, Ivan Philips, Goang-Tay Hsu, Krishna Sankar, Eric Ries, Tim Rohaly, John Zukowski: „Programujemy w Java tom 1 i 2" – Sybex & Exit, 1997.
8. „Current List of JDBC Supporting Vendors", Sun Microsystems, http://splash.javasoft.com.

Recenzent: Dr inż. Marcin Gorawski

Wpłynęło do Redakcji 8 grudnia 1999 r.

Abstract

This article presents and explains few problems, which usually come forward in the process of creation Java database applications. The first section gives brief description of Java Virtual Machine architecture (fig. 1). The second describes JDBC interface. It shows

the way in which programmer creates database connections and takes data from. It also presents four categories of JDBC drivers. It is shown logical flow of data for each type (fig.2, 3, 4, 5) and advantages and disadvantages, which follow using each of them. Next section shows differences between newest JDBC standard (version 2.0) and previous one (version 1.2). Fourth section presents some useful Swing components, which are part of Java extension project – JFC (Java Foundation Classes), from the point of view of database application. Next chapter shows how to download data from the server in the separate thread. There are considered two cases: when application uses Swing components or not. Sixth section contains security issues of internet applications. It points limitations, which follow Java applets and possible additional security leaks in comparing to alternative way of creation WWW database applications – CGI scripts. The last section is general recapitulation.