Jerzy Krawiec

Bezpieczne programowanie w Javie - kontrola dostępu = Security Programming in Java Access Control

Edukacja - Technika - Informatyka nr 2(20), 321-329

2017

©Wydawnictwo UR 2017 „Edukacja – Technika – Informatyka” nr 2/20/2017
ISSN 2080-9069 www.eti.rzeszow.pl
ISSN 2450-9221 online DOI: 10.15584/eti.2017.2.42

JERZY KRAWIEC

Bezpieczne programowanie w Javie – kontrola dostępu

Security Programming in Java – Access Control

Doktor inżynier, Politechnika Warszawska, Wydział Inżynierii Produkcji, Instytut Organizacji Systemów Produkcyjnych, Zakład Systemów Informatycznych, Polska

Streszczenie

Przedstawiono praktyczne aspekty bezpiecznego programowania w języku Java. Zbadano znaczenie hermetyzacji jako kluczowego elementu kontroli dostępu do kodu źródłowego Javy. Przeprowadzono badania reakcji maszyny wirtualnej Javy (JVM) w zależności od różnych wariantów zastosowanych specyfikatorów dostępu. Wyniki badań dowodzą, że konstrukcja kodu źródłowego odgrywa zasadniczą rolę w zapewnieniu bezpieczeństwa oprogramowania.

Słowa kluczowe: programowanie**,** bezpieczeństwo, kontrola dostępu, hermetyzacja

Abstract

The article discusses some aspects of safe Java programming. The importance of encapsulation as a key element of access control for Java source code has been examined. JVM has been tested, depending on the variants of the access controllers used. Research shows that the design of Java code plays a vital role in ensuring the software security.

Keywords: programming, security, access control, encapsulation

Wstęp

Programowanie to faza projektowania decydująca o sposobie budowy systemu informatycznego (Górny, Krawiec, 2016). Etap programowania powinien przywiązywać szczególną uwagę do bezpieczeństwa budowanej aplikacji. Efektem projektowania systemu informatycznego może być struktura danych oraz zestaw funkcji w strukturze programu (metodologia strukturalna). W przypadku metodologii obiektowej rezultatem projektowania jest szczegółowy projekt obiektowy w postaci hierarchii klas z dziedziczeniem, zestaw atrybutów i metod charakteryzujących obiekty w ramach klas. Najczęstszymi błędami popełnianymi przez programistów jest umożliwienie swobodnego dostępu do chronionych katalogów przy wysokim prawdopodobieństwie zidentyfikowania tej podatności przez hakerów i jej wykorzystania np. poprzez kradzież danych (Krawiec, 2012). Język pro-

gramowania obiektowego powinien zapewnić mechanizmy, które umożliwią implementację modelu obiektowego. W Javie jednym z tych mechanizmów oprócz dziedziczenia i polimorfizmu jest hermetyzacja (Schildt, 2014). Jest to mechanizm, który umożliwia połączenie kodu źródłowego i danych modyfikowanych przez ten kod oraz tworzy zabezpieczenie przed dostępem zewnętrznym.

Hermetyzacja

Hermetyzacja jest traktowana jako powłoka ochronna systemu chroniąca przed nieautoryzowanym dostępem z poziomu kodu spoza tej powłoki. Dostęp do kodu i danych wewnątrz tej powłoki ochronnej jest determinowany przez interfejsy obiektu. Podstawą hermetyzacji w Javie jest klasa, która decyduje o strukturze i zachowaniu danych i kodu współdzielonych przez zbiór obiektów. Szczegóły implementacji klasy powinny być chronione, gdyż ukrycie jej złożoności jest jej głównym zadaniem. Każda klasa i jej metody mogą być zadeklarowane jako publiczne, prywatne lub chronione. Z poziomu kodu źródłowego danej klasy mogą być dostępne jedynie metody i dane prywatne. To oznacza, że kod spoza klasy nie powinien mieć dostępu bezpośredniego do metod i zmiennej prywatnej danej klasy (Downey, 2012).

Hermetyzacja, oprócz łączenia kodu i danych, dotyczy także bardzo ważnego elementu, jakim jest kontrola dostępu. Za pomocą hermetyzacji można definiować, który program może mieć dostęp do składowych klasy. Aby zapobiec niewłaściwemu wykorzystaniu danych lub ich zmodyfikowaniu, należy je udostępniać za pomocą odpowiedniego interfejsu.

Modyfikatory dostępu

Z punktu widzenia bezpieczeństwa modyfikatory (specyfikatory) dostępu w Javie pełnią istotną rolę. Zarządzają one prawami dostępu do klas i ich składowych. To oznacza, że dostęp do składowych klas (pól i metod) może być zdefiniowany jako: publiczny (*public*), prywatny (*private*), chroniony (*protected*) lub pakietowy (*package*).

Dostęp publiczny oznacza, że dostęp nie jest niczym ograniczony i do tak zadeklarowanej klasy mają go wszystkie inne klasy. To oznacza, że klasa i jej składowe są dostępne do dowolnego innego kodu. W tym kontekście należy zauważyć, że metoda *main()* jest zawsze poprzedzona specyfikatorem *public*, gdyż jest wywoływana przez zewnętrzny kod, czyli system wykonawczy Javy (JVM – Java Virtual Machine).

Klasy i składowe prywatne oznaczają, że dostęp do nich jest możliwy wyłącznie z wnętrza tej klasy. Możliwe jest odczytywanie i zapisywanie jedynie przez metody tej klasy, natomiast dostęp z zewnątrz jest zabroniony (blokada odczytu i zapisu).

Dostęp chroniony oznacza, że składowe klasy są dostępne tylko dla metod tej klasy, klas potomnych oraz klas tego samego pakietu.

Klasy mogą być również pogrupowane w powiązane ze sobą zestawy zwane bibliotekami lub pakietami.

Badania – mechanizm kontroli dostępu

Badania sposobu działania dostępu publicznego i prywatnego przeprowadzono na przykładzie programu do pobierania wartości zmiennych:

```
class Alfa {
  int ab;
  public int bc;
  private int cd;
  void ustaw_c(int i) {
cd = i;
  }
  int pobierz_cd() {
return cd;
  }
}
class Beta {
  public static void main(String jekr[]) {
    Alfa obkt = new Alfa();
    obkt.ab = 12;
    obkt.bc = 28;
obkt.cd = 40; //Błąd!!!
    obkt.ustaw_cd(40);
     System.out.println("ab, bc, cd: "+ obkt.ab+ " " + obkt.bc+ "
" + obkt.pobierz_cd());
  }
}
```

Składowa klasy *Alfa* (zmienna *ab*) nie używa żadnego specyfikatora (specyfikator domyślny), co w tym przypadku oznacza, że dostęp jest publiczny. Zmienna *bc* ma specyfikator *public*, a zmienna *cd* używa specyfikatora *private*. To oznacza, że klasa *Beta* nie ma możliwości bezpośredniego odwołania się do zmiennej *cd*. Dlatego wykonanie linii kodu:

```
obkt.cd = 40;
```

skutkuje zgłoszeniem błędu dostępu w trakcie kompilacji. W związku z tym najczęściej stosuje się takie rozwiązania, które wymuszają dostęp do danych jedynie przez metody. Ponadto metody pomocnicze mogą być deklarowane jako prywatne składowe klasy.

Wprowadzenie dostępu pakietowego (pakietów) chroni przez powstaniem konfliktu nazw (Gosling, Joy, Steele, Bracha, Buckley, 2011). To oznacza, że bez zastosowania tej techniki niemożliwe stałoby się jednoczesne skorzystanie z dwóch klas w programie.

Brak specyfikatora dostępu oznacza, że składowe klasy są dostępne publicznie dla wszystkich innych klas, ale tylko z danego pakietu, czyli że składowe klasy nie są dostępne dla kodu pochodzącego z innych pakietów.

Badania – praktyczny aspekt hermetyzacji

Badania praktycznego aspektu hermetyzacji przeprowadzono na przykładzie stosu do przechowywania liczb całkowitych. Stos to implementacja struktury danych pochodząca z klasy *Stack*. Stos przechowuje dane zgodnie z zasadą LIFO (*Last In First Out*). Dane na stosie są umieszczane za pomocą metody *push()*, a ich zdejmowanie w odwrotnej kolejności odbywa się za pomocą metody *pop()*. Klasa *Delta1* definiuje stos liczb typu *int* przechowujący wartości do 15 liczb. Stos liczb jest przechowywany w tablicy o nazwie *tab*. Tablica jest indeksowana za pomocą zmiennej *st*, która zawsze zawiera indeks wskazujący na wierzchołek stosu. Element umieszczany na stosie jest nazwany *elem*. Konstruktor inicjuje zmienną o wartości –1, co oznacza stos pusty. Za pomocą metody *put()* umieszcza się nowy element na stosie, a metoda *get()* pobiera element ze stosu. Klasa *Sigma1* tworzy dwa stosy liczb całkowitych, umieszcza na nich wartości, a następnie je zdejmuje. Dla stosu pierwszego (*stos1*) liczby zaczynają się od 5, a dla stosu drugiego (*stos2*) liczby zaczynają się od 12. Kod programu przedstawia się następująco:

```
class Delta1 {
  int tab[] = new int[15];
int st;
  Delta1() {
  st = -1;
  }
  void put(int elem) {
    if(st==14)
      System.out.println("Stos jest pełny.");
    else
      tab[++st] = elem;
  }
  int get() {
if(st < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
return 0;
    }
    else
      return tab[st--];
  }
}
class Sigma1 {
  public static void main(String jekr[]) {
    Delta1 stos1 = new Delta1();
    Delta1 stos2 = new Delta1();
    for(int i=5; i<20; i++) stos1.put(i);
    for(int i=12; i<27; i++) stos2.put(i);
       System.out.println("Stos nr 1:");
    for(int i=0; i<15; i++)
       System.out.println(stos1.get());
       System.out.println("Stos nr 2:");
    for(int i=0; i<15; i++)
```

```
      System.out.println(stos2.get());
}
}
```

Taki sposób przechowywania elementów stosu w tablicy umożliwia zmianę elementów tablicy przez kod spoza klasy *Delta1*. Stwarza to niebezpieczeństwo dokonywania niepożądanych modyfikacji elementów stosu. Powstaje potencjalna podatność kodu programu, który powinien być w takiej sytuacji odpowiednio zabezpieczony.

W związku z tym, aby wyeliminować taką podatność kodu programu, należy zadeklarować zmienne *tab* oraz *st* jako *private*, co oznacza, że zmiana elementów stosu oraz ich bezpośrednie odczytanie nie będzie możliwe. Zastosowanie takiego rozwiązania zabezpieczy przed zmianą wartości tablicy przez inny kod programu. W takim przypadku kod programu przedstawia się następująco:

```
class Delta2 {
private int tab[] = new int[15];
  private int st;
  Delta2() {
  st = -1;
  }
  void put(int elem) {
    if(st==14)
System.out.println("Stos jest pełny.");
else
      tab[++st] = elem;
  }
  int get() {
if(st < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
return 0;
    }
    else
      return tab[st--];
  }
}
class Sigma2 {
  public static void main(String jekr[]) {
    Delta2 stos1 = new Delta2();
    Delta2 stos2 = new Delta2();
    for(int i=5; i<20; i++) stos1.put(i);
    for(int i=12; i<27; i++) stos2.put(i);
       System.out.println("Stos nr1:");
    for(int i=0; i<15; i++)
       System.out.println(stos1.get());
       System.out.println("Stos nr 2:");
    for(int i=0; i<15; i++)
       System.out.println(stos2.get());
stos1.st = -2; //Błąd!!!
    stos2.tab[4] = 45; //Błąd!!!
  }
}
```

Próba zmiany wartości zmiennych (*st* i *tab*) w postaci następującego kodu:

```
stos1.st = -2; //Błąd!!!
stos2.tablica[4] = 45; //Błąd!!!
```

skutkuje zablokowaniem dostępu do zmiennych (*st i tab*) spoza klasy *Delta2*, co oznacza zadziałanie zabezpieczenia przed nieautoryzowaną zmianą. Program wyświetli poprawne wyniki, ale wygeneruje komunikat:

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - st has private access in Delta2 at Sigma2.main(Delta2.java:35).

Badania pakietów – nowego wymiaru sterowania dostępem

Java zapewnia wielopoziomowy system ochrony zmiennych. Chociaż sterowanie dostępem w Javie nie jest proste, warto przeanalizować i zrozumieć zasady dotyczące dostępu do składowych klas, aby uniknąć potencjalnych podatności kodu programu.

Klasa może mieć tylko dwa poziomy dostępu: publiczny i domyślny. Różne kombinacje modyfikatorów dostępu i ich konsekwencje rozważymy na przykładzie 5 plików źródłowych: 3 klas z pierwszego pakietu oraz 2 klas z drugiego pakietu. Wszystkie programy wyświetlają jedynie wartości zmiennych.

Pierwsza klasa *Test1* z pakietu *zestaw1* definiuje 4 zmienne typu *byte* o różnym dostępie. Kod pliku ma następującą postać:

```
package zestaw1;
public class Test1 {
    byte zm1 = 15;
    public byte zm2 = 12;
pravite byte zm3 = 24;
    protected int zm4 = 45;
public Test1() {
        System.out.println("Wartość zmiennej zm1 wynosi " + zm1);
        System.out.println("Wartość zmiennej zm2 wynosi " + zm2);
        System.out.println("Wartość zmiennej zm3 wynosi " + zm3);
        System.out.println("Wartość zmiennej zm4 wynosi " + zm4);
    }
}
```

Klasa nie zawiera błędów kompilacji, gdyż bez względu na typ specyfikatora dostępu zmiennych zapisany kod źródłowy dotyczy tej samej klasy.

Druga klasa *Test2* z pakietu *zestaw1* jest potomną klasy *Test1*. Zbadamy możliwość dostępu jej składowych klasy *Test2* do składowych klasy *Test1*. Kod źródłowy przedstawia się następująco:

```
package zestaw1;
class Test2 extends Test1 {
    Test2() {
        System.out.println("Wartość zmiennej zm1 wynosi " + zm1);
        System.out.println("Wartość zmiennej zm2 wynosi " + zm2);
        System.out.println("Wartość zmiennej zm3 wynosi " + zm3);
//Błąd!!!
        System.out.println("Wartość zmiennej zm4 wynosi " + zm4);
    }
}
```

W tym przypadku JVM wygeneruje komunikat o braku dostępu zmiennej *zm3* z klasy *Test2* do zmiennej *zm3* z klasy *Test1*, gdyż zmienna *zm3* w klasie *Test1* ma specyfikator *private*.

Trzecia klasa *Test3* z pakietu *zestaw1* nie jest klasą potomną żadnej klasy i też będzie próbowała dostać się do klasy *Test1*. Przetestujemy jej kod źródłowy pod tym kątem:

```
package zestaw1;
class Test3 {
Test3() {
    Test1 obkt = new Test1();
System.out.println("Wartość zmiennej zm1 wynosi " + obkt.zm1);
        System.out.println("Wartość zmiennej zm2 wynosi " +
obkt.zm2);
        System.out.println("Wartość zmiennej zm3 wynosi " +
obkt.zm3); //Błąd!!!
        System.out.println("Wartość zmiennej zm4 wynosi " +
obkt.zm4);
    }
}
```

W tym przypadku dostęp z klasy *Test3* do *Test1* jest możliwy z wyjątkiem zmiennej *zm3*, która jest zadeklarowana jako *private*, gdyż JVM wygeneruje komunikat o braku dostępu zmiennej *zm3* z klasy *Test3* do zmiennej *zm3* z klasy *Test1*.

Czwarta klasa *Test4* z pakietu *zestaw2* jest klasą potomną klasy *Test1*. Zbadamy reakcję JVM na uruchomienie kodu źródłowego w sytuacji, gdy klasa *Test4* będzie próbowała się dostać do składowych klasy bazowej. Kod źródłowy tej klasy przedstawia się następująco:

```
package zestaw2;
class Test4 extends Test1 {
    Test4() {
        System.out.println("Wartość zmiennej zm1 wynosi " + zm1);
//Błąd!!!
        System.out.println("Wartość zmiennej zm2 wynosi " + zm2);
        System.out.println("Wartość zmiennej zm3 wynosi " + zm3);
//Błąd!!!
        System.out.println("Wartość zmiennej zm4 wynosi " + zm4);
    }
}
```

W tym przypadku JVM wygeneruje komunikat o braku dostępu zmiennych *zm1* i *zm3* z klasy *Test4* do zmiennych *zm1* i *zm3* z klasy *Test1*, gdyż klasa *Test4* znajduje się w innym pakiecie niż klasa *Test1* (dotyczy zmiennej *zm1*), a zmienna *zm3* w klasie *Test1* ma specyfikator *private*.

Piąta klasa *Test5* z pakietu *zestaw2* nie jest klasą potomną żadnej klasy i też będzie próbowała dostać się do klasy *Test1*, a obie klasy znajdują się w różnych pakietach. Kod źródłowy przedstawia się następująco:

```
package zestaw2;
```

```
class Test5 {
    Test5() {
    Zestaw1.Test1 obkt = new zestaw1.Test1();
        System.out.println("Wartość zmiennej zm1 wynosi " +
obkt.zm1); //Błąd!!!
        System.out.println("Wartość zmiennej zm2 wynosi " +
obkt.zm2);
        System.out.println("Wartość zmiennej zm3 wynosi " +
obkt.zm3); //Błąd!!!
        System.out.println("Wartość zmiennej zm4 wynosi " +
obkt.zm4); //Błąd!!!
    }
}
```

W tym przypadku JVM wygeneruje komunikat o braku dostępu zmiennych *zm1*, *zm3 i zm4* z klasy *Test5* do zmiennych klasy *Test1*, gdyż:

– klasa *Test5* znajduje się w innym pakiecie niż klasa *Test1* (dotyczy zmiennej *zm1*),

– zmienna *zm3* w klasie *Test1* ma specyfikator *private*,

– zmienna *zm4* w klasie *Test1* ma specyfikator *protected*, a klasy *Test1* i *Test5* znajdują się w różnych pakietach.

Podsumowanie

Autoryzowany dostęp do danych klasy jest przeważnie zapewniony przez metody, jednak nie zawsze należy korzystać z kontrolowanego dostępu do składowych klasy. Jeżeli z pewnych względów składowe klasy powinny być publiczne, to nie stosuje się takiego zabezpieczenia. Natomiast wszędzie tam, gdzie należy chronić dane, zasadą jest wprowadzenie mechanizmów zabezpieczeń w kodzie źródłowym programu. Bezpieczeństwo oprogramowania wymaga wprowadzenia i przestrzegania zasady: „co nie jest dozwolone, jest zabronione".

Luki (błędy) w kodzie źródłowym są największym problemem związanym z bezpieczeństwem. Bezpieczeństwo oprogramowania jest ściśle związane z bezpieczeństwem systemu informatycznego oraz środowiska, w którym funkcjonuje. Oprogramowanie uznane za bezpieczne musi sprostać odpowiednim wymaganiom będącym odzwierciedleniem atrybutów bezpieczeństwa informacji, czyli zapewnieniem: poufności, dostępności i integralności informacji.

Błędy programistyczne mają krytyczne znaczenie w zapewnieniu bezpieczeństwa systemów informatycznych. Pozornie drobna pomyłka programisty może się okazać poważną luką w systemie, stwarzając olbrzymie zagrożenie dla użytkowników. Najlepszym sposobem na podniesienie poziomu bezpieczeństwa oprogramowania jest unikanie błędów programistycznych poprzez tworzenie stabilnego kodu źródłowego i weryfikowalnego procesu zastosowanych w nim zabezpieczeń przed etapem wdrożenia systemu informatycznego.

Literatura

Downey, A. (2012). *Think Java. How to Think Like a Computer Scientist*. Pobrane z: http://thinkapjava.com (20.04.2017).

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. (2011). *The Java™ Language Specification. Java SE 7 Edition*. ORACLE.

Górny, P., Krawiec, J. (2016). Cyberbezpieczeństwo – podejście systemowe. *Obronność – Zeszyty Naukowe Wydziału Zarządzania i Dowodzenia Akademii Obrony Narodowej*, *2* (18), 75–89.

Krawiec, J. (2012). Zabezpieczanie danych. Część 3. Projektowanie systemów informatycznych. *ITprofessional*, *8*, 60–64.

Schildt, H. (2014). *Java The Complete Reference.* New York: McGraw-Hill Companies, Inc.