Reflection API

Reflection API jest zbiorem programowych narzędzi, które reprezentują, bądź też odzwierciedlają, klasy i obiekty w bieżącej wirtualnej maszynie Javy. *Reflection API* są używane głównie do pisania narzędzi do tworzenia oprogramowania, takich jak debuggery, przeglądarki klas, aplikacje do graficznego tworzenia oprogramowania. Dzięki *Reflection API* można:

- określić klasę obiektu
- uzyskać informację o modyfikatorach klasy, jej parametrach, metodach, konstruktorach i klasy ojcowskiej
- odnaleźć, jakimi stałe oraz metody należą do danego interfejsu
- utworzyć instancję klasy, której nazwa nie jest znana do chwili uruchomienia aplikacji
- uzyskać lub też ustawić właściwości (pola) obiektu, nawet jeśli ich nazwy nie są znane do chwili uruchomienia programu.
- wywołać metodę obiektu, nawet jeśli nie była ona znana do chwili uruchomienia programu
- utworzyć nową tablicę, której rozmiar oraz typ elementów nie są znane do chwili wywołania programu, oraz zmodyfikować zawartość tablicy.

Uwaga: nie należy używać *reflection API* gdy inne, bardziej naturalne metody są wystarczające. Na przykład, jeśli przyzwyczaiłeś się do używania wskaźników do funkcji w innym języku, pewnie chciałbyś użyć w ten sam sposób obiekt `Method` z *reflection API*. Nie rób tego! Twój program będzie łatwiejszy do debuggowania i utrzymania, jeśli jednak nie użyjesz obiektu `Method`. Zamiast tego powinieneś zdefiniować interfejs, a następnie zaimplementować go w odpowiedniej klasie.

W terminologii obiektowej mówi się, że klasa posiada parametry, zmienne członkowskie lub pola. Ponieważ klasa `Field` jest częścią *reflection API*, aby się dobrze kojarzyło, częściej będzie używana nazwa "pole".

Sprawdzanie (prześwietlanie) klas

Jeśli implementujesz przeglądarkę klas, to pewnie zainteresowany jesteś uzyskaniem informacji o klasie dostępnych w chwili wykonywania programu. Tymi informacjami mogłyby być na przykład: nazwy pól, metod, konstruktorów. Dobrze byłoby też znać nazwy interfejsów, które implementuje dana klasa. Wszystkie te informacje można uzyskać za pomocą obiektu `Class`, który odzwierciedla daną klasę.

Dla każdej klasy, JRE (Java Runtime Environment) utrzymuje niezmienny obiekt `Class`, zawierający informacje o klasie. Obiekt `Class` reprezentuje, lub odzwierciedla, klasę. Z reflection API można wywoływać metody obiektu `Class`, które zwracają obiekty typu `Constructor, Method, Field`. Posługując się tymi obiektami można uzyskać informację o odpowiadających im konstruktorach, metodach i polach badanej klasy.

Obiekt `Class` reprezentuje interfejs. Można wywołać metody obiektu `Class` aby znaleźć informacje o modyfikatorach interfejsów, metodach, publicznych stałych. Nie wszystkie metody `Class` są odpowiednie, gdy obiekt `Class` odzwierciedla interfejs. Na przykład, nie ma sensu wywoływać `getConstructors` gdy obiekt `Class` reprezentuje interfejs.

Podrozdział Prześwietlanie Interfejsów wyjaśnia, które metody klasy `Class` mogą być użyte do pobierania informacji o interfejsach.

Dostarczanie obiektów Class

Obiekty `Class` uzyskać można na szereg sposobów:

- Jeśli instancja klasy jest dostępna, to można wywołać metodę `Object.getClass`. Metoda ta jest użyteczna, kiedy ma się obiekt, ale nie znana jest jego klasa. Poniższy fragment kodu pokazuje, jak uzyskać obiekt `Class` dla obiektu o nazwie `mystery`:

```
Class c = mystery.getClass();
```

- Jeśli celem jest uzyskanie obiektu `Class` odzwierciedlającego klasę nadrzędną obiektu, który jest już odzwierciedlany przez obiekt typu `Class`, należy skorzystać z metody `getSuperclass`. W poniższym przykładzie `getSuperclass` zwraca obiekt typu `Class` związanych z klasą `TextComponent`, gdyż `TextComponent` jest klasą nadrzędną klasy `TextField`:

```
TextField t = new TextField();
Class c = t.getClass();
Class s = c.getSuperclass();
```

- Jeśli znana jest nazwa klasy podczas kompilacji programu, można uzyskać obiekt `Class` przez dodanie `.class` do nazwy klasy. W przykładzie uzyskano obiekt `Class` reprezentujący klasę `Button`:

```
Class c = java.awt.Button.class;
```

- Jeśli nazwa klasy jest nieznana podczas kompilacji, jednak będzie dostępna w czasie działania programu, można wtedy użyć metody `forName`. W pokazany przykładzie jeśli `String` nazwany `strg` będzie ustawiony na "java.awt.Button", to `forName` zwróci obiekt klasy `Class`, powiązany z klasą `Button`:

```
Class c = Class.forName(strg);
```

Uzyskanie nazwy klasy

Każda klasa w Javie ma nazwę. Kiedy deklarowana jest klasa, nazwa jej jest podawana za słowem kluczowym `class`. Deklaracja klasy `Point` wygląda następująco::

```
public class Point {int x, y;}
```

W czasie działania programu nazwę obiektu `Class` można uzyskać wywołując metodę `getName`. Ciąg `String` zwrócony przez `getName` jest w pełni określoną nazwą klasy.

Poniższy program uzyskuje nazwę klasy danego obiektu. Najpierw, znajduje on odpowiadający obiekt `Class`, a następnie wywołuje metodę `getName` obiektu `Class`.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleName {
```

```
   public static void main(String[] args) {
      Button b = new Button();
      printName(b);
   }

   static void printName(Object o) {
       Class c = o.getClass();
       String s = c.getName();
       System.out.println(s);
   }
}
```

Wynikiem działania programu jest:

```
java.awt.Button
```

Odkrywanie modyfikatorów klasy

Deklaracja klasy może zawierać następujące modyfikatory: `public`, `abstract`, lub `final`. Modyfikatory klasy poprzedzają słowo kluczowe `class` w definicji klasy. W następującym przykładzie modyfikatorami klasy są `public` oraz `final`:

```
public final Coordinate {int x, int y, int z}
```

Aby zidentyfikować modyfikatory klasy w czasie wykonywania programu należy:

1. Wywołać metodę `getModifiers` obiektu `Class` aby otrzymać zbiór modyfikatorów.
2. Sprawdzić modyfikatory, wywołując `isPublic`, `isAbstract` lub `isFinal`.

W poniższym przykładzie wykonano te kroki na przykładzie klasy `String`.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleModifier {

   public static void main(String[] args) {
      String s = new String();
      printModifiers(s);
   }

   public static void printModifiers(Object o) {
      Class c = o.getClass();
      int m = c.getModifiers();
      if (Modifier.isPublic(m))
         System.out.println("public");
      if (Modifier.isAbstract(m))
         System.out.println("abstract");
      if (Modifier.isFinal(m))
         System.out.println("final");
   }
}
```

W wyniku działania na konsoli pojawią się wypisane wszystkie modyfikatory klasy `String`:

```
public
final
```

Znajdowanie klasy nadrzędnej

W jawie zaimplementowano mechanizm jednokrotnego dziedziczenia. Aby określić, która klasa jest klasą nadrzędną danej klasy, wystarczy wywołać metodę `getSuperclass`. Metoda ta zwraca obiekt `Class` reprezentujący klasę nadrzędną, lub zwraca null, jeśli klasa nadrzędna nie istnieje. Aby więc znaleźć wszystkich poprzedników danej klasy, wystarczy wywoływać iteracyjnie `getSuperclass` aż do chwili, gdy metoda ta zwróci null.

W poniższym programie znajdowane są wszystkie klasy nadrzędne klasy `Button`:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleSuper {

   public static void main(String[] args) {
      Button b = new Button();
      printSuperclasses(b);
   }

   static void printSuperclasses(Object o) {
      Class subclass = o.getClass();
      Class superclass = subclass.getSuperclass();
      while (superclass != null) {
         String className = superclass.getName();
         System.out.println(className);
         subclass = superclass;
         superclass = subclass.getSuperclass();
      }
   }
}
```

Wynikiem tego programu jest:

```
java.awt.Component
java.lang.Object
```

Identyfikowanie interfejsu implementowanego przez Class

Typ obiektu jest określony nie tylko przez jego klasę czy też klasę nadrzędną, ale również przez interfejs. W deklaracji klasy nazwy implementowanych przez klasę interfejsów występują za słowem kluczowym `implements`. Na przykład klasa `RandomAccessFile` implementuje interfejsy `DataOutput` oraz `DataInput`:

```
public class RandomAccessFile implements DataOutput, DataInput
```

W celu ustalenia, jakimi interfejsami dysponuje klasa można wywołać metodę `getInterfaces`. Metoda ta zwraca tablicę obiektów `Class`. Typ `Class` jest używany do reprezentowania obiektów będących odzwierciedleniami interfejsów. Każdy obiekt `Class` w zwróconej tablicy reprezentuje jeden interfejs implementowany przez klasę. Nazwę interfejsu można określić wywołując metodę `getName` obiektu `Class` z tablicy. Więcej szczegółów na temat uzyskiwania dalszych informacji o interfejsach można znaleźć w podrozdziale Prześwietlanie interfejsów.

Program poniżej wypisuje wszystkie interfejsy implementowane przez klasę `RandomAccessFile` class.

```
import java.lang.reflect.*;
import java.io.*;

class SampleInterface {

   public static void main(String[] args) {
      try {
          RandomAccessFile r = new RandomAccessFile("myfile", "r");
          printInterfaceNames(r);
      } catch (IOException e) {
          System.out.println(e);
      }
   }

   static void printInterfaceNames(Object o) {
      Class c = o.getClass();
      Class[] theInterfaces = c.getInterfaces();
      for (int i = 0; i < theInterfaces.length; i++) {
         String interfaceName = theInterfaces[i].getName();
         System.out.println(interfaceName);
      }
   }
}
```

Wynikiem programu jest są w pełni kwalifikowane nazwy interfejsów

```
java.io.DataOutput
java.io.DataInput
```

Prześwietlanie interfejsów

Obiekt `Class` jest używany do reprezentacji zarówno klas, jak i interfejsów. Aby sprawdzić, czym taki obiekt faktycznie jest, należy wywołać metodę `isInterface`.

W celu uzyskania informacji na temat interfejsu należy wywołać metody klasy `Class`. Metoda `getFields` pozwala np. znaleźć publiczne stałe interfejsu. Podrozdział Identyfikacja pól klasy ma przykład zawierający wywołanie `getFields`. Aby uzyskać informacje o metodach interfejsu należy użyć metody `getMethods` (co zostało opisane w podrozdziale Otrzymywanie Informacji o metodach). Modyfikatory interfejsów można uzyskać wywołując metodę `getModifiers` (jak to było omówione w podrozdziale Odkrywanie modyfikatorów klas).

Wywołując `isInterface` poniższy program znajduje, że `Observer` jest interfejsem, zaś `Observable` jest klasą:

```
import java.lang.reflect.*;
import java.util.*;

class SampleCheckInterface {

   public static void main(String[] args) {
      Class observer = Observer.class;
      Class observable = Observable.class;
      verifyInterface(observer);
      verifyInterface(observable);
   }

   static void verifyInterface(Class c) {
```

```
            String name = c.getName();
            if (c.isInterface()) {
               System.out.println(name + " is an interface.");
            } else {
               System.out.println(name + " is a class.");
            }
         }
      }
```

Wynikiem programu jest:

```
java.util.Observer is an interface.
java.util.Observable is a class.
```

Identyfikacja pól klasy.

Do identyfikacji pól klasy używa się metody `getFields` obiektu `Class` object. Metoda `getFields` zwraca tablicę obiektów `Field` zawierającą odpowiedniki wszystkich dostępnych publicznych pól klasy. Pole typu `public` jest osiągalne jeśli jest ono członkiem:

- tej klasy
- klasy nadrzędnej danej klasy
- interfejsem implementowanym przez klasę
- interfejsem rozszerzonym z interfejsu zaimplementowanego przez tą klasę

Metody dostarczane przez `Field` pozwalają na odczytanie nazwy pola, jego typu i modyfikatorów. Można nawet otrzymać i ustawić wartość pola, jak to opisano w podrozdziałach Uzyskiwanie wartości Pól oraz Ustawianie Wartości Pól.
Poniższy program drukuje nazwy oraz typy pól jakimi dysponuje klasa `GridBagConstraints`. Program najpierw uzyskuje obiekty `Field` dla klasy przez wywołanie `getFields`, a następnie wywołuje metody `getName` i `getType` na każdym z obiektów `Field`.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleField {

   public static void main(String[] args) {
      GridBagConstraints g = new GridBagConstraints();
      printFieldNames(g);
   }

   static void printFieldNames(Object o) {
      Class c = o.getClass();
      Field[] publicFields = c.getFields();
      for (int i = 0; i < publicFields.length; i++) {
         String fieldName = publicFields[i].getName();
         Class typeClass = publicFields[i].getType();
         String fieldType = typeClass.getName();
         System.out.println("Name: " + fieldName +
           ", Type: " + fieldType);
         }
      }
}
```

Obcięty wydruk wyjścia generowanego przez powyższy program:

```
Name: RELATIVE, Type: int
Name: REMAINDER, Type: int
Name: NONE, Type: int
Name: BOTH, Type: int
Name: HORIZONTAL, Type: int
Name: VERTICAL, Type: int
.
.
.
```

Odkrywanie konstruktorów klasy

Aby utworzyć instancję klasy należy wywołać jej konstruktor. Konstruktory, tak jak i inne metody, mogą być przeciążone i rozpoznane po sygnaturze (nazwie, liście parametrów).
Informacje o konstruktorach klasy udostępniane są w wynik wywołania metody `getConstructors`. Metoda ta zwraca tablicę obiektów `Constructor`. Można używać metod klasy `Construktor` aby: określić nazwę konstruktora, zbiór modyfikatorów, typów parametrów, oraz zbiór wyrzucanych wyjątków. Instancję typu `Constructor` wykorzystuje się do tworzenia obiektów poprzezz jej metodę `Constructor.newInstance`. (zobacz rozdział Manipulowanie obiektami).
Poniższy przykład wypisuje typu parametrów dla każdego konstruktora klasy `Rectangle`. Program działa w następujący sposób:

1. Pobiera tablicę obiektów `Constructor` z obiektu `Class` przez wywołanie `getConstructors`.
2. Dla każdego elementu tablicy `Constructor`, jest utworzona tablica obiektów typu `Class` przez wywołanie `getParameterTypes`. Obiekt `Class` w tablicy reprezentuje parametry lub konstruktor.
3. Program wywołuje metodę `getName` aby poznać nazwy klas dla każdego elementu w tablicy.

Przykładowy program:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleConstructor {

   public static void main(String[] args) {
      Rectangle r = new Rectangle();
      showConstructors(r);
   }

   static void showConstructors(Object o) {
      Class c = o.getClass();
      Constructor[] theConstructors = c.getConstructors();
      for (int i = 0; i < theConstructors.length; i++) {
         System.out.print("( ");
         Class[] parameterTypes =
            theConstructors[i].getParameterTypes();
         for (int k = 0; k < parameterTypes.length; k ++) {
            String parameterString = parameterTypes[k].getName();
            System.out.print(parameterString + " ");
            }
         System.out.println(")");
         }
   }
```

```
}
```

W pierwszej linii wyjścia utworzonego przez program nie występują żadne parametry, gdyż dla konkretnego obiektu odpowiedni konstruktor nie posiadał żadnych parametrów.
W kolejnych liniach wypisywane są albo parametry typu podstawowego albo w pełni kwalifikowane nazwy obiektów. Wyjściem z programu jest:

```
( )
( int int )
( int int int int )
( java.awt.Dimension )
( java.awt.Point )
( java.awt.Point java.awt.Dimension )
( java.awt.Rectangle )
```

Aby wywnioskować, która z metod publicznych należy do klasy należy wywołać metodę `getMethods`. Tablica zwrócona przez `getMethods` zawiera obiekty typu `Method`. Obiekty `Method` pozwalają odsłonić nazwy metody, zwracane wartości, typy parametrów, zbiór modyfikatorów, oraz zbiór wyrzucanych wyjątków. Wszystkie te informacje przydają się przy implementowaniu przeglądarek klas i zasobów. Metoda `Method.invoke` pozwala samemu wywołać arbitralną metodę. Zobacz też: Wywoływanie metod.

Poniższy program drukuje nazwy, zwracany typ, typy parametrów dla każdej metody publicznej w klasie `Polygon`. Program wykonuje następujące zadania:

1. Wywołując `getMethods` obiektu `Class` któw `Method`
2. Dla każdego elementu w tablicy `Method` program:
    a. uzyskuje nazwy metod przez wywołanie `getName`
    b. uzyskuje typ zwracany przez wywołanie `getReturnType`
    c. tworzy tablicę obiektów `Class` wywołując `getParameterTypes`
3. Tablica obiektów `Class` utworzona w kroku poprzednim reprezentuje parametry metod. Aby uzyskać nazwę klasy dla każdego z tych parametrów, program wywołuje `getName` dla każdego obiektu `Class` w tablicy.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleMethod {

   public static void main(String[] args) {
      Polygon p = new Polygon();
      showMethods(p);
   }

   static void showMethods(Object o) {
      Class c = o.getClass();
      Method[] theMethods = c.getMethods();
      for (int i = 0; i < theMethods.length; i++) {
         String methodString = theMethods[i].getName();
         System.out.println("Name: " + methodString);
         String returnString =
           theMethods[i].getReturnType().getName();
         System.out.println("   Return Type: " + returnString);
         Class[] parameterTypes = theMethods[i].getParameterTypes();
         System.out.print("   Parameter Types:");
         for (int k = 0; k < parameterTypes.length; k ++) {
            String parameterString = parameterTypes[k].getName();
            System.out.print(" " + parameterString);
         }
         System.out.println();
      }
```

```
    }
}
```

Skrócona postać wyjścia generowanego przez program:

```
Name: equals
   Return Type: boolean
   Parameter Types: java.lang.Object
Name: getClass
   Return Type: java.lang.Class
   Parameter Types:
Name: hashCode
   Return Type: int
   Parameter Types:
.
.
.
Name: intersects
   Return Type: boolean
   Parameter Types: double double double double
Name: intersects
   Return Type: boolean
   Parameter Types: java.awt.geom.Rectangle2D
Name: translate
   Return Type: void
   Parameter Types: int int
```

Manipulowanie obiektami

Manipulacja obiektami szczególnie przydaje się w aplikacjach do graficznego tworzenia aplikacji (*GUI builder*) lub debuggerach. Na przykład wiele aplikacji *GUI builder* pozwala użytkownikowi końcowemu wyselekcjonować `Button` z menu komponentów, utworzyć obiekt `Button`, i następnie kliknąć `Button` podczas działania aplikacji wewnątrz *GUI builder*.

Tworzenie obiektów

Najłatwiejszym sposobem tworzenia obiektów w Javie jest użycie operatora `new`:

```
Rectangle r = new Rectangle();

static Object createObject(String className) {
      Object object = null;
      try {
          Class classDefinition = Class.forName(className);
          object = classDefinition.newInstance();
      } catch (InstantiationException e) {
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println(e);
      } catch (ClassNotFoundException e) {
          System.out.println(e);
      }
      return object;
   }
```

Bardziej złożony przykład tworzenia obiektów:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleInstance {

   public static void main(String[] args) {

      Rectangle rectangle;
      Class rectangleDefinition;
      Class[] intArgsClass = new Class[] {int.class, int.class};
      Integer height = new Integer(12);
      Integer width = new Integer(34);
      Object[] intArgs = new Object[] {height, width};
      Constructor intArgsConstructor;

      try {
        rectangleDefinition = Class.forName("java.awt.Rectangle");
        intArgsConstructor =
            rectangleDefinition.getConstructor(intArgsClass);
        rectangle =
            (Rectangle) createObject(intArgsConstructor, intArgs);
      } catch (ClassNotFoundException e) {
          System.out.println(e);
      } catch (NoSuchMethodException e) {
          System.out.println(e);
      }
   }

   public static Object createObject(Constructor constructor,
                                     Object[] arguments) {

      System.out.println ("Constructor: " + constructor.toString());
      Object object = null;

      try {
        object = constructor.newInstance(arguments);
        System.out.println ("Object: " + object.toString());
        return object;
      } catch (InstantiationException e) {
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println(e);
      } catch (IllegalArgumentException e) {
          System.out.println(e);
      } catch (InvocationTargetException e) {
          System.out.println(e);
      }
      return object;
   }
}
```

Powyższy program drukuje opis konstruktora oraz obiektu, który on tworzy:

```
Constructor: public java.awt.Rectangle(int,int)
Object: java.awt.Rectangle[x=0,y=0,width=12,height=34]
```

Uzyskiwanie wartości pól

Uzyskiwanie wartości pól klasy jest procesem trzystopniowym, na który składa się:

1. Utworzenie obiektu `Class`.
2. Utworzenia obiektu `Field` przez wywołanie `getField` obiektu `Class`.
3. Wywołanie jednej z metod `get` na obiekcie `Field`.

Oto przykład programu:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleGet {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 325);
      printHeight(r);

   }

   static void printHeight(Rectangle r) {
      Field heightField;
      Integer heightValue;
      Class c = r.getClass();
      try {
        heightField = c.getField("height");
        heightValue = (Integer) heightField.get(r);
        System.out.println("Height: " + heightValue.toString());
      } catch (NoSuchFieldException e) {
          System.out.println(e);
      } catch (SecurityException e) {
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println(e);
      }
   }
}
```

Wyjściem z programu jest:

```
Height: 325
```

Ustawianie wartości pól

Niektóre z debuggerów pozwalają zmieniać wartość pola podczas sesji debuggowania. We własnych implementacjach, aby zmienić wartość jakiegoś pola należy użyć metody `set` klasy `Field`. Modyfikacja wartości pól klasy przeprowadzana jest w następujących krokach:

1. Utworzenie obiektu `Class`.
2. Utworzenie obiektu `Field` przez wywołanie metody `getField` obiektu `Class`.
3. Wywołanie odpowiedniej metody `set` obiektu `Field`.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleSet {

   public static void main(String[] args) {
      Rectangle r = new Rectangle(100, 20);
```

```
        System.out.println("original: " + r.toString());
        modifyWidth(r, new Integer(300));
        System.out.println("modified: " + r.toString());
    }

    static void modifyWidth(Rectangle r, Integer widthParam ) {
        Field widthField;
        Integer widthValue;
        Class c = r.getClass();
        try {
          widthField = c.getField("width");
          widthField.set(r, widthParam);
        } catch (NoSuchFieldException e) {
            System.out.println(e);
        } catch (IllegalAccessException e) {
            System.out.println(e);
        }
    }
}
```

Wynikiem powyższego programu jest:

```
original: java.awt.Rectangle[x=0,y=0,width=100,height=20]
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]
```

Wywoływanie metod

Wywoływanie metod klas, które nie były znane w chwili pisania programu wymaga wykonania następujących czynności:

1. Utworzenia obiektu `Class` odpowiadającego obiektowi, którego metody mają być wywołane.
2. Utworzenia obiektu `Method` przez wywołanie metody `getMethod` obiektu `Class`. Metoda `getMethod` posiada dwa argumenty: argument typu `String` zawierający nazwę metody, oraz tablicę obiektów klasy `Class`. Każdy element w tablicy odpowiada parametrowi metody, która jest wywoływana.
3. Wywołanie właściwej metody przez użycie `invoke`. Metoda `invoke` ma dwa argumenty: tablicę wartości argumentów, jakie mają być przekazane do metody wywoływanej; oraz obiekt, którego klasa deklaruje lub dziedziczy wywoływaną metodę.

Przykładowy program pokazuje jak dynamicznie wywoływać nieznane wcześniej metody. Program uzyskuje obiekt `Method` dla metody `String.concat` i używa następnie `invoke` do połączenia dwóch obiektów typu `String`.

```
import java.lang.reflect.*;

class SampleInvoke {

    public static void main(String[] args) {
        String firstWord = "Hello ";
        String secondWord = "everybody.";
        String bothWords = append(firstWord, secondWord);
        System.out.println(bothWords);
    }

    public static String append(String firstWord, String secondWord) {
```

```
            String result = null;
            Class c = String.class;
            Class[] parameterTypes = new Class[] {String.class};
            Method concatMethod;
            Object[] arguments = new Object[] {secondWord};
            try {
              concatMethod = c.getMethod("concat", parameterTypes);
              result = (String) concatMethod.invoke(firstWord, arguments);
            } catch (NoSuchMethodException e) {
                System.out.println(e);
            } catch (IllegalAccessException e) {
                System.out.println(e);
            } catch (InvocationTargetException e) {
                System.out.println(e);
            }
            return result;
        }
    }
```

Wyjściem z programu jest komunikat:

```
Hello everybody.
```

Praca z tablicami

Klasa `Array` dostarcza metod pozwalających na dynamiczne tworzenie tablic oraz na dostęp do nich.

Identyfikacja tablic

Aby sprawdzić, czy obiekt ukrywający się pod daną referencją jest rzeczywiście tablicą wystarczy posłużyć się metodą `Class.isArray`. W poniższym przykładzie program wypisze nazwy tablic ukrytych w danym obiekcie. Program wykonuje następujące kroki:

1. Uzyskuje obiekt `Class` reprezentujący obiekt badany.
2. Otrzymuje obiekty `Field` dla obiektu `Class` otrzymanego w poprzednim kroku.
3. Dla każdego obiektu `Field` uzyskuje odpowiadający mu obiekt `Class` przez wywołanie metody `getType`.
4. Obiekty otrzymane w kroku 3 sprawdzane są na okoliczność bycia tablicami za pomocą metody `isArray`.

Oto źródło programu:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleArray {

    public static void main(String[] args) {
        KeyPad target = new KeyPad();
        printArrayNames(target);
    }

    static void printArrayNames(Object target) {
```

```
            Class targetClass = target.getClass();
            Field[] publicFields = targetClass.getFields();
            for (int i = 0; i < publicFields.length; i++) {
               String fieldName = publicFields[i].getName();
               Class typeClass = publicFields[i].getType();
               String fieldType = typeClass.getName();
               if (typeClass.isArray()) {
                  System.out.println("Name: " + fieldName +
                     ", Type: " + fieldType);
               }
            }
         }
      }

      class KeyPad {

         public boolean alive;
         public Button power;
         public Button[] letters;
         public int[] codes;
         public TextField[] rows;
         public boolean[] states;
      }
```

Wyjściem z programu jest:

```
Name: letters, Type: [Ljava.awt.Button;
Name: codes, Type: [I
Name: rows, Type: [Ljava.awt.TextField;
Name: states, Type: [Z
```

gdzie lewy nawias kwadratowy oznacza, że obiekt jest tablicą (opis deskryptorów typu zwracanych przez `getType` zawarty jest w dokumencie *The Java Virtual Machine Specification*).

Uzyskiwanie informacji o typach elementów przechowywanych w tablicach (typach komponentów)

Jeśli mamy tablicę `arrowKeys` zdefiniowaną jak niżej:

```
Button[] arrowKeys = new Button[4];
```

to typem elementów przechowywanych w tablicy jest `Button`. Typem komponentu wielowymiarowej tablicy jest tablica (o wymiarze mniejszym o jeden – rekurencja ta ciągnie się aż do tablicy jednowymiarowej). W poniższym przykładzie typem komponentu tablicy `matrix` jest `int[]`:

```
int[][] matrix = new int[100][100];
```

Wywołanie metody `getComponentType` dla obiektu `Class` reprezentującego tablicę pozwala uzyskać typ elementów tej tablicy. W przykładzie poniżej po wywołaniu metody `getComponentType` wypisana zostanie nazwa klasy każdego typu komponentu tablicy (poprzedzona nazwą klasy tablicy).

```
import java.lang.reflect.*;
import java.awt.*;
```

```
class SampleComponent {

   public static void main(String[] args) {
      int[] ints = new int[2];
      Button[]  buttons = new Button[6];
      String[][] twoDim = new String[4][5];

      printComponentType(ints);
      printComponentType(buttons);
      printComponentType(twoDim);
   }

   static void printComponentType(Object array) {
      Class arrayClass = array.getClass();
      String arrayName = arrayClass.getName();
      Class componentClass = arrayClass.getComponentType();
      String componentName = componentClass.getName();
      System.out.println("Array: " + arrayName +
         ", Component: " + componentName);
   }
}
```

Wynikiem programu jest:

```
Array: [I, Component: int
Array: [Ljava.awt.Button;, Component: java.awt.Button
Array: [[Ljava.lang.String;, Component: [Ljava.lang.String;
```

Tworzenie tablic

Do tworzenia tablic w sposób dynamiczny (tj. gdy typ komponentów w tablicy dostępny będzie dopiero w czasie uruchomienia programu) służy metoda `Array.newInstance`.

Poniższy przykład pokazuje, jak użyć metody `newInstance` do utworzenia kopii tablicy o dwukrotnie większym rozmiarze, niż tablica oryginalna. Metoda `newInstance` przyjmuje jako argumenty długość nowej tablicy oraz typ komponentu. Kod przykładu:

```
import java.lang.reflect.*;

class SampleCreateArray {

   public static void main(String[] args) {
      int[] originalArray =  {55, 66};
      int[] biggerArray = (int[]) doubleArray(originalArray);
      System.out.println("originalArray:");
      for (int k = 0; k < Array.getLength(originalArray); k++)
         System.out.println(originalArray[k]);
      System.out.println("biggerArray:");
      for (int k = 0; k < Array.getLength(biggerArray); k++)
         System.out.println(biggerArray[k]);
   }

   static Object doubleArray(Object source) {
      int sourceLength = Array.getLength(source);
      Class arrayClass = source.getClass();
      Class componentClass = arrayClass.getComponentType();
      Object result = Array.newInstance(componentClass,
```

```
                                            sourceLength * 2);
        System.arraycopy(source, 0, result, 0, sourceLength);
        return result;
    }
}
```

Program wypisze na konsoli co następuje:

```
originalArray:
55
66
biggerArray:
55
66
0
0
```

Metody `newInstance` można również używać do tworzenia tablic wielowymiarowych. W takim przypadku parametrami metody są typ komponentu oraz tablica elementów typu `int` reprezentująca wymiary nowej tablicy. W przykładzie poniżej pokazano, jak utworzyć tablicę wielowymiarową:

```
import java.lang.reflect.*;

class SampleMultiArray {

   public static void main(String[] args) {

      // The oneDimA and oneDimB objects are one
      // dimensional int arrays with 5 elements.

      int[] dim1 = {5};
      int[] oneDimA = (int[]) Array.newInstance(int.class, dim1);
      int[] oneDimB = (int[]) Array.newInstance(int.class, 5);

      // The twoDimStr object is a 5 X 10 array of String objects.

      int[] dimStr = {5, 10};
      String[][] twoDimStr =
                 (String[][]) Array.newInstance(String.class,dimStr);

      // The twoDimA object is an array of 12 int arrays. The tail
      // dimension is not defined. It is equivalent to the array
      // created as follows:
      //    int[][] ints = new int[12][];

      int[] dimA = {12};
      int[][] twoDimA = (int[][]) Array.newInstance(int[].class, dimA);
   }
}
```

Uzyskiwanie oraz ustawianie wartości elementów

W większości programów dostęp do elementów tablicy realizowany jest w następujący sposób:

```
int[10] codes;
codes[3] = 22;
aValue = codes[3];
```

Takiego sposobu jednak nie można zastosować, jeśli nazwa tablicy znana jest dopiero w czasie wykonywania programu. W celu uzyskania dostępu do wartości elementów w tablicy oraz do ustawienia im nowych wartości należy posłużyć się metodami `set` oraz `get` klasy `Array` (w wersjach specjalizowanych do obsługiwania wartości typów standardowych). W poniższym przykładzie wartościa parametru metody `setInt` jest `int`, oraz obiekt zwrócony przez `getBoolean` jest opakowaniem dla wartości typu `boolean`.

Przykładowy program używa metod `set` oraz `get` do skopiowania elementów jednej tablicy do tablicy drugiej:

```
import java.lang.reflect.*;

class SampleGetArray {

   public static void main(String[] args) {
      int[] sourceInts = {12, 78};
      int[] destInts = new int[2];
      copyArray(sourceInts, destInts);
      String[] sourceStrgs = {"Hello ", "there ", "everybody"};
      String[] destStrgs = new String[3];
      copyArray(sourceStrgs, destStrgs);
   }

   public static void copyArray(Object source, Object dest) {
      for (int i = 0; i < Array.getLength(source); i++) {
         Array.set(dest, i, Array.get(source, i));
         System.out.println(Array.get(dest, i));
      }
   }
}
```

Wyjściem z programu jest:

```
12
78
Hello
there
everybody
```