Reflection - Die Java Reflection API

Mit der Einführung der Java Beans wurde auch die Geburt von Reflection (vormalig "Introspection") gezwungenermaßen eingeleitet denn bislang waren Objekte lediglich in statischen Strukturen zu finden. Das wiederum bedeutete, dass die Klassen bereits zur Compilezeit verfügbar sein mussten. Da dieser Umstand aber nicht dem Wunsch generische Applikationen bilden zu können, die über eine Plugin-Schnittstelle erweitert werden können, entsprach, entschloss man sich eine neue Technik einzuführen – Introspection (später Reflection).

Der Begriff Reflection bedeutet dabei, dass Programme, die generisch aufgebaut sind, in die Lage versetzt werden, Objektinstanzen von für sie unbekannten Klassen (beispielsweise durch direkte Eingaben des Benutzers) zu erzeugen.
Die Reflection-API umfasst dabei folgende Funktionalitäten:

- Feststellung der Klasse eines Objekts
- Informationen über Klassenmodifikatoren, Member, Methoden, Konstruktoren und Superklassen holen.
- Herausbekommen welche Konstanten und Methodendeklarationen zu einem Interface gehören.
- Erzeugung einer Instanz einer erst zur Laufzeit bekannten Klasse.
- Lesen und Setzen von Werten eines Members selbst wenn das Member namentlich erst zur Laufzeit bekannt wird.
- Aufrufen von Methoden eines bis zur Laufzeit unbekannten Objekts.
- Erzeugung von Arrays, deren Größe und Typ bis zur Laufzeit nicht bekannt sind und Modifikationen an den einzelnen Array-Komponenten.

Inhaltsverzeichnis [Verbergen]

Untersuchung von Klassen

[bearbeiten]

Wenn Sie während der Laufzeit Ihres Programms Informationen über bestimmte Klassen benötigen (z.B. wenn Sie ein Analyse-Tool für Java-Klassen schreiben) so können Sie dies mittels der Reflection-API lösen. Für alles was mit Reflection zu tun hat benötigen Sie ein sogenanntes Class-Objekt.

Mit diesem Objekt, welches übrigens die Java Runtime Environment (JRE) für alle Klassen als unveränderliche Komponente ohne Einschränkung besitzt, ist es möglich alle erdenkliche Information über die Klasse zu erhalten. So gibt es die Klasse Method, die eine Methode aus der Klasse repräsentiert bzw. die Klassen Constructor und Field für Konstruktor-Informationen und Membervariablen.

Auch Interfaces lassen sich mit der Klasse Class abbilden. Hierbei muss allerdings darauf geachtet werden, dass man das Objekt entsprechend seines Typs verwendet. So macht es beispielsweise keinen Sinn eine Methode eines Class-Objekts aufzurufen, wenn das Class-Objekt ein Interface repräsentiert. Wie man erkennt ob es sich bei dem Class-Objekt um ein Interface handelt wird später behandelt.

Abfragen des Class-Objekts

[bearbeiten]

Es gibt verschiedene Wege um an ein Class-Objekt zu kommen:

- Wenn bereits eine Instanz des unbekannten Objekts existiert können Sie die Methode getClass aus java.lang.Object aufrufen. Als Beispiel hierfür soll das unbekannte Objekt unknownObject dienen:

```
Class theClass = unknownObject.getClass();
```

- Um das Class-Objekt der Superklasse eines Objektes zu erhalten, das wiederum von einem Class-Objekt „reflektiert“ wird zu erhalten ruft man die Methode getSuperClass(). Im folgenden Beispiel erhalten wir also eine Repräsentation der Klasse javax.

```
JTextField textField = new JTextField();
Class theClass = textField.getClass();
Class theSuperClass = theClass.getSuperClass();
```

- Wenn die Klasse bekannt ist kann man durch hinten anstellen von „.class“ an den Klassennamen einfach das Class-Objekt holen. Für einen String (java.lang.String) etwa:

```
Class theClass = java.lang.String.class;
```

- Wenn der Klassenname zur Compilezeit zwar unbekannt ist aber zur Laufzeit sich ergibt (weil z.B. der Benutzer den Namen eingegeben hat) kann auch ein Class-Objekt erzeugt werden. Hierfür benutzt man die forName(...)-Methode (aus java.lang.Class). Im folgenden Beispiel soll die Stringvariable strg den Inhalt javax.swing.JTable haben. In diesem Fall würde die Variable theClass die Klasse JTable reflektieren.

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

Abfrage des Klassennamens

[bearbeiten]

Manchmal ist es interessant zu wissen, wie der Name einer Klasse eines unbestimmten Objekts lautet – beispielsweise um ihn in einem Fenster zur Auswahl zu stellen. Jede Klasse besitzt mit dem Punkt der Deklaration einen Namen. Der Name für die folgende Klasse wäre HyperlinkLabel.

```
public class HyperlinkLabel {
        String text;
        ...
}
```

Um nun an den Namen zur Laufzeit zu gelangen gibt es die Methode getName() in der Klasse java.lang.Class. Das

folgende Beispiel soll die Nutzung verdeutlichen:

```
import java.lang.reflect.*;
import javax.swing.*;
public class SampleName {
        public static void main(String[] args) {
                JButton myButton = new JButton(„Ok“);
                printName(myButton);
        }

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

Output:

```
javax.swing.
JButton
```

Herausfinden von Klassenmodifikatoren [bearbeiten]

Eine Klassendeklaration kann über die Schlüsselwörter public, final und abstract eingeleitet werden. Um nun diese Modifikatoren zur Laufzeit eines Programms zu erfahren muss auf ein Class-Objekt die Methode getModifiers() aufgerufen werden. Das Ergebnis dieses Methodenaufrufs kann dann wiederum auf die einzelnen Eigenschaften über die Methoden isPublic(...), isFinal(...) und isAbstract(...) gegengeprüft werden.

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

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

   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");
   }
}
```

Output:

```
public
 final
```

Superklassen identifizieren

[bearbeiten]

Wie in Kapitel 1.1 angedeutet ist es auch möglich die Vaterklasse (=Superklasse) einer Klasse zu ermitteln. Dazu bedient man sich der getSuperClass()-Methode aus java.lang.Class, welche als Ergebnis ein Class-Objekt besitzt, das die Superklasse reflektiert bzw. null ist wenn keine Vaterklasse existiert.

```
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();
                }
        }
}
```

Output:

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

Identifikation von implementierten Interfaces

[bearbeiten]

Ein wesentliches Merkmal einer Java-Klasse ist die Liste der von ihr implementierten Interfaces. Der Aufruf von getInterfaces () auf ein Class-Objekt liefert ein Array von Class-Elementen welches als Inhalt alle von der Klasse implementierten Interfaces besitzt (oder null, wenn kein Interface implementiert wurde). Mit diesen einzelnen Class-Objekten lässt sich dann der Name des Interfaces via getName() finden. Um sicher zu sein, dass es sich bei dem vorliegenden Class-Objekt um ein Interface bzw. um eine Klasse handelt verwendet man die Methode isInterface(), welche einen bool'schen Wert {false, true} zurückliefert. Das heisst true, wenn es sich um ein Interface handelt und ansonsten false (dies ist der Umkehrschluss: Ist es kein Interface so ist es garantiert eine Klasse).

```
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);
                }
        }
}
```

Output:

java.io.DataOutput

java.io.DataInput

Beachten: Es werden die vollqualifizierenden Namen ausgegeben!

Klassen nach ihren Membern (Fields) analysieren

[bearbeiten]

In dem Eingangs erwähnten Beispiel einen Klassenbrowser zu entwickeln ist es natürlich notwendig alle (öffentlichen) Member einer Klasse darzustellen. Um dies zu bewerkstelligen stellt die Reflection-API mehrere beschreibende Klassen zur Verfügung. Da ein Member eine Methode oder eine Variable sein kann existieren auch Klassen für genau diese beiden Objektarten. Für die Analyse der Membervariablen wird die Klasse javax.reflection.Field benutzt.

Auf eine öffentliche Variable kann nur dann zugegriffen werden, wenn

1. sich die Membervariable in dieser Klasse befindet
2. sich die Membervariable in der Vaterklasse dieser Klasse befindet

3. sich die Membervariable im implementierten Interface dieser Klasse befindet oder
4. sich die Membervariable im sog. Vaterinterface des aktuellen Interfaces befindet

Alle notwendigen Daten, die es über eine Membervariable gibt werden von der Klasse Field zur Verfügung gestellt. Dazu gehören ihr Name, ihr Typ und ihre Zugriffsmodifikatoren.

Das folgende Beispiel bedient sich der Klasse java.lang.String um die oben angesprochene Funktionalität zu verdeutlichen:

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

class SampleField {
        public static void main(String[] args) {
                String myString = new String("Hallo");
                printFieldNames(myString);
        }

        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);
                }
        }
}
```

Output:

```
Name: CASE_INSENSITIVE_ORDER, Type: java.util.
                                Comparator
```

Konstruktoren einer Klasse ermitteln

[bearbeiten]

Zweifellos ist das wichtigste über eine Klasse die Zahl ihrer Konstruktoren sowie deren Parameter. Dies ist wichtig um zur Laufzeit Objekte dieser Klasse erzeugen zu können (es sei denn die Klasse ist zur Compile-Zeit bereits bekannt – dann bräuchte man sich aber auch nicht die Finger wegen Reflection brechen ;-)).

Wie gesagt werden zur Instantiierung von Objekten Informationen über den Konstruktor benötigt. Da es sich bei Konstruktoren ähnlich verhält wie mit normalen Methoden, können auch Konstruktoren überladen werden. Ergo kann es eine ganze Menge von unterschiedlichen(!) Konstruktoren geben. Interessant allerdings sind nur die, die auch öffentlich gehalten sind. Da ein Zugriff auf private bzw. geschützte Konstruktoren nicht möglich ist, kann auch darauf verzichtet werden, diese in der Liste der Konstruktoren aufzuzeigen.

Langer Rede kurzer Sinn: Um die Konstruktoren einer Klasse herauszufinden muss lediglich eine einzige Methode namens getConstructors() auf ein Class-Objekt angewandt werden. Diese Methode liefert ein Array von Constructor-Objekten zurück. Die Detailinformation über den Konstruktor liefern die Hilfsmethoden getParameterTypes() in Verbindung mit getName().

```
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(")");
                }
        }
}
```

Output:

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

Die Methoden einer Klasse darstellen

[bearbeiten]

Um alle öffentlichen Methoden einer Klasse zu bekommen muss einfach die Methode getMethods() aus der Reflection-API aufgerufen werden (java.reflection.Class). Diese Methode liefert ähnlich wie die getConstructors bzw. getFields-Methode ein Array von Objekten zurück. Diese Objekte vom Typ Method besitzen ihrerseits Methoden, mit denen es möglich wird alle Eigenschaften einer Methode zu erhalten. Mit der invoke()-Methode ist es sogar möglich diese Methode mit ihren Parametern aufzurufen!

```
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();
                }
        }
}
```

Output:

```
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
```

Objekte manipulieren [bearbeiten]

GUI-Builder, Debugger und ähnliche Programme müssen in der Lage sein, zur Laufzeit bestimmte Objekte zu verändern.

Ein GUI-Builder muss z.B. dem Benutzer eine Liste von graphischen Komponenten zur Auswahl präsentieren, eine Komponente selektieren lassen und sofort nach Einfügen der Komponente in ein Formular auf die Komponente (beispielsweise ein JButton) klicken lassen. In den vorliegenden Unterkapiteln widmen wir uns zu diesem Zweck der Erzeugung und Manipulation von Objekten zur Laufzeit eines Java-Programmes mittels der Reflection-API.

Objekterzeugung [bearbeiten]

Der einfachste Weg in Java ein Objekt zu erzeugen ist dies unter Zuhilfenahme von **new** zu machen.

```
JFrame myFrame = new JFrame();
```

Dies ist auch logisch, denn in der Regel weiss man, welche Art von Objekt man benötigt. Versetzt man sich aber in die Lage eines GUI-Builders so ist dort die Objekterzeugung nicht so einfach zu handhaben. Man könnte meinen, dass auch zur Laufzeit Objekte ähnlich zum oben dargestellen Beispiel erzeugt werden könnten.

```
String className = "javax.swing.JFrame";

// ... laden der Klasse
```

```
Object o = new(className);    // ungültige Anweisung, new erwartet keine Parameter!
```

Zum Glück gibt es die Reflection-API, die dies für uns möglich macht. Allerdings muss zwischen zwei Fällen der Objekterzeugung unterschieden werden. Diese Unterschiede sollen in den folgenden zwei Abschnitten erläutert werden.

Erzeugung von Objekten mittels parameterlosen Konstruktoren

[bearbeiten]

Ein parameterloser Konstruktor wird auch Default-Konstruktor genannt. Sollten Sie über diesen ein Objekt erzeugen wollen so steht Ihnen die Methode newInstance aus der Klasse Class zur Verfügung. Falls die Klasse, von der Sie ein Objekt wünschen, keinen Default-Konstruktor besitzt so wird eine NoSuchMethodException geworfen.

Das folgende Beispiel soll die Vorgehensweise verdeutlichen:

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

class SampleNoArg {

   public static void main(String[] args) {
      Rectangle r = (Rectangle) createObject("java.awt.Rectangle");
      System.out.println(r.toString());
   }

   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;
    }
}
```

Output:

```
java.awt.Rectangle[x=0,y=0,width=0,
                   height=0]
```

Erzeugung von Objekten mittels parametrisierten Konstruktoren

[bearbeiten]

Um ein Objekt einer Klasse zu erzeugen, welches parametrisierte Konstruktoren anbietet, kann die Methode newInstance aus der Klasse Constructor (**VORSICHT: vgl. Erzeugung von Objekten mittels parameterlosen Konstruktoren**) genutzt werden. Dazu verfolgt man folgende Schritte:

1. Erzeugung eines Class-Objekts des benötigten Objekts
2. Erzeugung eines Constructor-Objekts mittels der Methode getConstructor aus dem Objekt aus (1). Die Methode besitzt einen Parameter, der ein Array von Class-Objekten erwartet, entsprechend der Parameter des gewünschten Konstruktors.
3. Erzeugung des Objekts mittels newInstance auf dem Constructor-Objekt wobei die für den Konstruktor notwendigen Parameter in einem Array verpackt an newInstance übergeben werden.

Das untere Beispielprogramm zeigt die oben aufgeführten Schritte im Java-Code auf. Es wird dazu genutzt ein Objekt vom

Typ Rectangle mit den Parametern "12" und "34" zu erzeugen. Es macht also nichts anderes als

```
Rectangle rectangle = new Rectangle(12, 34);
```

Die Typen der Parameter dieses Konstruktors gehören zu den primitiven Datentypen. Da aber stets Objekte für die Erzeugung benötigt werden muss hier auf den Wrapper Integer zurückgegriffen werden.

```
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;
    }
 }
```

Output:

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

Werte von Membern einlesen

[bearbeiten]

Manchmal ist es nötig, Werte von Objekten, deren Struktur Sie nicht genau kennen, zu lesen und anzuzeigen. Zum Beispiel bei der Entwicklung eines Debuggers. Um dies zu machen müssen drei Schritte unternommen werden.

1. Erzeugen Sie ein Class-Objekt
2. Erzeugen Sie ein Field-Objekt mittels der Methode getField des Class-Objekts
3. Lesen Sie den Wert des Members über die entsprechende get-Methode auf

Für Member mit primitiven Datentypen bietet die Klasse Field spezielle Methoden an (getInt, getFloat, ...). Sollte die Member-Variable ein echtes Objekt sein so benutzen Sie die get-Methode.

Im folgenden Beispiel ist der Name der Member-Variable bereits bekannt. Dies muss in Ihrer Anwendung nicht der Fall sein. Um auch dieses Problem zu lösen benutzen Sie die Techniken zur Identifikation von Member-Variablen.

```
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);
       }
    }
 }
```

Output:

```
Height:
325
```

Werte von Membern setzen

[bearbeiten]

Es ist nicht nur möglich, die Werte von Member-Variablen zur Laufzeit zu lesen sondern ebenso zu setzen. Das ist beispielsweise Vorteilhaft bei der Entwicklung eines Debuggers, der es erlaubt zur Laufzeit Werte von Membern zu ändern.

Um die Werte neu zu setzen genügt es die set-Methoden aus der Klasse Field zu nutzen.

```
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);
      }
   }
}
```

Output:

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

Aufruf von Methoden

[bearbeiten]

In manchen Entwicklungsumgebungen ist es möglich, beim Debuggen Methoden bestimmter Objekte aufzurufen um entweder irgendwelche Werte neu berechnen zu lassen oder im simpelsten Fall Membern neue Werte zu setzen. Um dies zu tun sind lediglich folgende Schritte nötig:

1. Erzeugen Sie ein Class-Objekt dessen Methode Sie aufrufen möchten.
2. Erzeugen Sie ein Method-Objekt durch Aufruf von getMethod auf das Class-Objekt. Die getMethod-Methode besitzt zwei Parameter: Der erste Parameter vom Typ String, der den Namen der aufzurufenden Methode darstellt. Der zweite Parameter ist ein Array von Class-Objekten, welches die einzelnen Parameter der aufzurufenden Methode darstellt.
3. Rufen Sie die Methode via invoke-Methode auf.

```
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;
    }
}
```

Output:

```
Hello everybody.
```

Arbeiten mit Arrays [bearbeiten]

Erkennung von Arrays [bearbeiten]

Erkennung des Array-Typs [bearbeiten]

[bearbeiten]

Arrays erzeugen

Lesen und Schreiben von Array-Elementen [bearbeiten]

Zusammenfassung aller Reflection-Klassen [bearbeiten]

Klasse	Beschreibung
Array	Bietet statische Methoden an um zur Laufzeit Arrays zu erzeugen und zu nutzen.
Class	Repräsentiert Klassen und Interfaces.
Constructor	Liefert Informationen über Konstruktoren einer Klasse und macht sie greifbar. Hiermit werden zur Laufzeit Instanzen eines bestimmten Typs generiert.
Field	Liefert Informationen über Member-Variablen einer Klasse oder eines Interfaces.
Method	Liefert Informationen über Methoden einer Klasse und bietet darüber hinaus die Möglichkeit Methoden explizit aufzurufen.
Modifier	Liefert Informationen über die Zugriffsmodifikatoren von Fields und Methoden.
Object	Wird für getClass genutzt.

Dieser Artikel basiert auf http://java.sun.com/docs/books/tutorial/reflect/

Detailbeschreibung: Beispiele zum Thema wurden aus dem Java Tutorial von ***Dale Green*** übernommen

Seitenkategorien: Tutorial | API

Navigation

- Hauptseite
- Wiklet-Portal
- Aktuelle Ereignisse
- Letzte Änderungen
- Zufälliger Artikel
- Hilfe
- Spenden

Suche

Los Suche

Werkzeuge

- Was zeigt hierhin
- Verlinkte Seiten
- Spezialseiten