Universität
Osnabrück

Informatik B - Objektorientierte Programmierung in Java

Vorlesung 25:
Reflection 2

Inhalt

- Reflection 2
  - Verwenden parametrisierter Konstruktoren
  - Membervariablen
  - Arrays
  - Beispiele/Ausblick
- Siehe auch Krüger: Handbuch der Java-Programmierung, Kapitel 43; **www.javabuch.de**

Parametrisierte Konstruktoren

- `Constructor getConstructor(Class[] parameterTypes)`
- `Constructor[] getConstructors()`
- `Object newInstance(Object[] initargs)`
- `java.lang.reflect.Constructor`

Bsp: ParametrisierteKonstruktoren.java

```
class TestConstructors
{
  private String arg1;
  private String arg2;
  ...

  public TestConstructors(String arg1, String arg2)
  {
    this.arg1 = arg1;
    this.arg2 = arg2;
  }
  public void print()
  {
    System.out.println("arg1 = " + arg1);
    System.out.println("arg2 = " + arg2);
  }
}
```

Bsp: (2) ParametrisierteKonstruktoren.java

```
Class clazz = TestConstructors.class;
//Formale Parameter definieren
Class[] formparas = new Class[2];
formparas[0] = String.class;
formparas[1] = String.class;
try {
  Constructor cons =
           clazz.getConstructor(formparas);
  //Aktuelle Argumente definieren
  Object[] actargs = new Object[] {"eins",
                                   "zwei"};
  Object obj = cons.newInstance(actargs);
  ((TestConstructors)obj).print();
```


Exkurs: `instanceof`

- Finden und Umwandeln des Typs eines Objekts
- Zur Erinnerung: Collections
- Verbesserung des Beispiels:

```
if (obj instanceof TestConstructors)
    ((TestConstructors)obj).print();
```

- Warum kann man ein Object nicht nach seinem Typ fragen?
- Wieviele Typen kann ein Object haben?

Zugriff auf Membervariablen

- Um bestimmte oder alle öffentlichen Membervariablen zu erhalten, werden auf das Klassenobjekt folgende Methoden angewendet:
- **`Field getField(String name)`**
- **`Field[] getFields()`**

Alle, auch nicht-öffentliche Membervariablen:

- **`Field getDeclaredField(String name)`**
- **`Field[] getDeclaredFields()`**

Zugriff auf Membervariablen (2)

- **`java.lang.reflect.Field`** enthält Methoden zum Zugriff auf die Variablen:

- **`Class getType()`**
  liefert das Klassenobjekt zum Typ der Variable
- **`String getName()`**
  liefert den Namen der Variablen
- **`Object get(Object obj)`**
  liefert den Wert der Variablen
- **`void set(Object obj, Object value)`**
  setzt den Wert der Variablen

Beispiel: `PrintableObject.java`

- Überlagert die Methode **`toString`**
- Diese soll in der Lage sein, die Namen und Inhalte aller Membervariablen des zugehörigen Objekts auszugeben.

Beispiel: `PrintableObject.java`

```
import java.lang.reflect.*;
 public static void main(String[] args)
  {
    JavaProgrammer jim = new JavaProgrammer();
    jim.name           = "Jim Miller";
    jim.department     = "Operating Systems";
    jim.age            = 32;
    String[] langs     = {"C","Pascal","PERL","Java"};
    jim.languages      = langs;
    jim.linesofcode    = 55000;
    jim.jdk12          = true;
    jim.swing          = false;
    System.out.println(jim);
  }
} //..
```


Beispiel (Fortsetzung)

```
class Employee
extends PrintableObject
{
  public String name;
  public String department;
  public int    age;
}
class Programmer
extends Employee
{
  public String[] languages;
  public int      linesofcode;
}
class JavaProgrammer
extends Programmer
{
  public boolean jdk12;
  public boolean swing;
}
```


Beispiel (Fortsetzung)

```
public class PrintableObject
{
  public String toString()
  {
    StringBuffer sb = new StringBuffer(200);
    Class clazz = getClass();
    while (clazz != null) {
      Field[] fields = clazz.getDeclaredFields();
      for (int i = 0; i < fields.length; ++i) {
        sb.append(fields[i].getName() + " = ");
        try {
          Object obj = fields[i].get(this);
          if (obj.getClass().isArray()) {
            Object[] ar = (Object[])obj;
```

Beispiel (Fortsetzung)

```
            for (int j = 0; j < ar.length; ++j) {
              sb.append(ar[j].toString() + " ");
            }
            sb.append("\n");
          } else {
            sb.append(obj.toString() + "\n");
          }
        } catch (IllegalAccessException e) {
          sb.append(e.toString() + "\n");
        }
      }
      clazz = clazz.getSuperclass();
    }
    return sb.toString();
  }
```


Reflection und Arrays

- **`java.lang.reflect.Array`**
- Enthält statische Methoden zum
  - Erzeugen von Arrays
  - Zugriff auf Array-Elemente

Erzeugung von Arrays

- Eindimensional:
  `public static Object newInstance(Class componentType, int length) throws NegativeArraySizeException`
- Mehrdimensional:
  `public static Object newInstance(Class componentType, int[] dimensions) throws IllegalArgumentException, NegativeArraySizeException`

Beispiel: `ArrayErzeugung.java`

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

public class ArrayErzeugung
{
  public static void createArray1()
  {
    //Erzeugt ein eindimensionales int-Array
    Object ar =
         Array.newInstance(Integer.TYPE, 3);
    int[] iar = (int[])ar;
    for (int i = 0; i < iar.length; ++i) {
      iar[i] = i;
      System.out.println(iar[i]);
    };
  }//...
```

Beispiel: `ArrayErzeugung.java (2)`

```
public static void createArray2()
 {
   //Erzeugt ein zweidimensionales String-Array
   Object ar =
         Array.newInstance(String.class, new int[]{7, 4});
   String[][] sar = (String[][])ar;
   for (int i = 0; i < sar.length; ++i) {
     for (int j = 0; j < sar[i].length; ++j) {
       sar[i][j] = "(" + i + "," + j + ")";
       System.out.print(sar[i][j] + " ");
     }
     System.out.println();
   };
 }
 public static void main(String[] args)
 {
   createArray1();
   System.out.println("--");
   createArray2();
} }
```


Zugriff auf Array-Elemente

- Für vollkommen dynamischen Zugriff auf Array-Elemente (ohne **[]**-Notation) stellt **`java.lang.reflect.Array`** zur Verfügung:
- **`public static Object get(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`**
- **`public static void set(Object array, int index, object value) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`**
- **`public static int getLength(Object array) throws IllegalArgumentException`**

`get`-Methoden für primitive Typen

- **`public static boolean getBoolean(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`**

  analog für
  `byte char short int long float double`

`set`-Methoden für primitive Typen

- **`public static void setBoolean(Object array, int index, boolean z) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`**

  analog für
  `byte char short int long float double`

Beispiel: `ArrayZugriff.java`

```
import java.lang.reflect.*;
public class ArrayZugriff
{
  public static void createArray1()
  {
    //Erzeugt ein eindimensionales int-Array
    Object ar = Array.newInstance(Integer.TYPE, 3);
    for (int i = 0; i < Array.getLength(ar); ++i) {
      Array.set(ar, i, new Integer(i));
      System.out.println(Array.getInt(ar, i));
    };
  }
 public static void main(String[] args)
  {
    createArray1();
} }
```


Beispiel: Diskussion

- Unterschied zu **`ArrayErzeugung.java`**:
  - Setzen und abfragen der Elemente ausschließlich mit Methoden der Klasse **`Array`**
  - Setzen des **`int`**-Wertes mit **`Integer`**-Wrapper
  - Auslesen typkonform mit **`getInt`**

Hinweis: Mehrdimensionale Arrays

- Zugriff auf mehrdimensionale Arrays analog
- Mehrdimensionale Arrays werden als geschachtelte eindimensionale Arrays dargestellt.

  Beispiel: **`int [][] a = new int[2][3];`**

- Für Zugriff auf zweidimensionales Array sind zwei **`get`**-Aufrufe nötig. Der erste liefert das geschachtelte innere Array, der zweite wird auf dieses angewandt und liefert das Element.

Zusammenfassung

- Reflection
  - Komponentenbasierte Softwareentwicklung
  - Die Klassen **`Object`** und **`Class`**
  - Dynamisches Laden und Instantiieren von Klassen
  - Aufruf von Methoden ohne und mit Parametern
  - Verwenden parametrisierter Konstruktoren
  - Membervariablen
  - Arrays: Erzeugung und Zugriff
- Zugriffe auf Methoden und Membervariablen über Reflection sind deutlich langsamer.

Ausblick

- Reflection ist Grundlage für
  - Serialisierung
  - (Enterprise) Java Beans (Introspection)
- *JavaBeans* verwenden das **`reflection`**-API zur *Introspection* = Auslesen der Eigenschaften
- Reflection stellt die Grundlage für dynamische, hochkonfigurierbare Java-Anwendungen, gerade in Bereich der Business-Anwendungen dar.
- Mit Reflection werden komponentenorientierte Anwendungen möglich.