Informatik B - Objektorientierte Programmierung in Java

Vorlesung 24:
Reflection 1

Inhalt

- Reflection
  - Motivation
  - Die Klassen **`Object`** und **`Class`**
  - Dynamisches Laden und Instantiieren von Klassen
  - Aufruf von Methoden ohne und mit Parametern
  - Beispiele
- Siehe auch Krüger: Handbuch der Java-Programmierung, Kapitel 43; **`www.javabuch.de`**

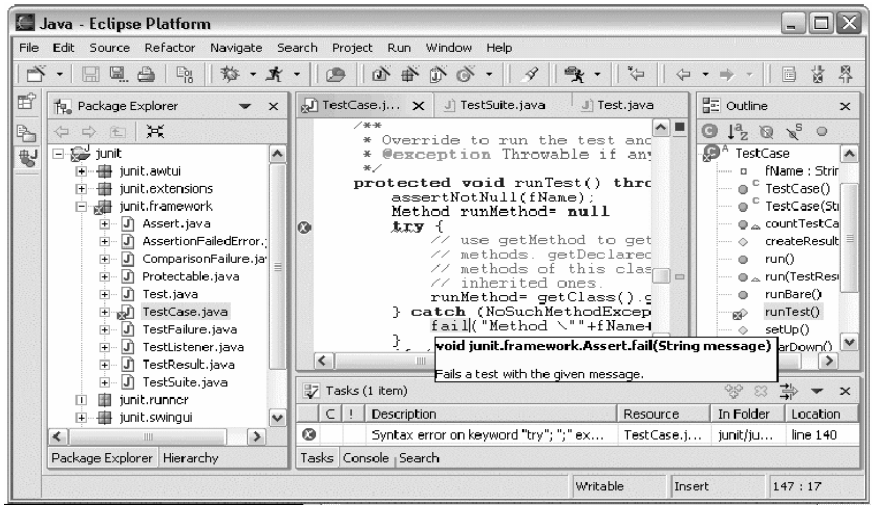
Problemstellung

- Um ein Objekt anzulegen, eine seiner Methoden aufzurufen oder auf eine seiner Membervariablen zuzugreifen, muss normalerweise der Code der Klasse zur *Compilezeit* bekannt sein. Bsp:

```
Person p = new Person();
p.setAlter(37);
```

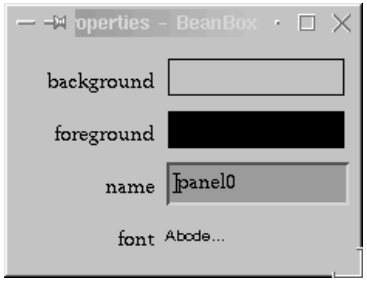
- Dies verhindert die Entwicklung von generischen Werkzeugen und hochkonfigurierbaren Anwendungen aus (zugekauften) Komponenten.

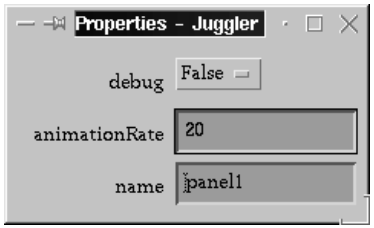
IDE und automatische Tests

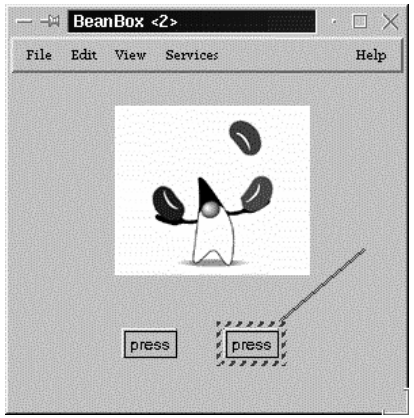


(IDE – Integrated Development Environment)

Generische Werkzeuge/Komponenten



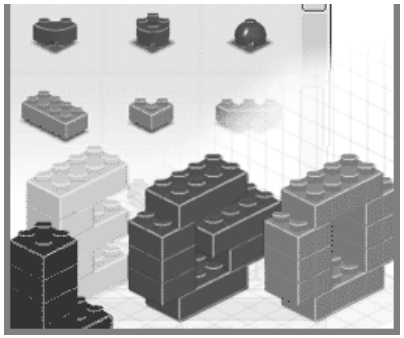




Komponentenorientierte Software

- Das Zusammenfügen von zugekauften Softwarekomponenten zu größeren Business-Anwendungen.

Vriwzduhe|Ohjr=



Wunsch

- Benötigt wird eine Möglichkeit in Java, Klassen zu laden und zu instantiieren, ohne dass bereits zur Compilezeit ihr Name bekannt sein muss.
- Zugriff auf Methoden und Membervariablen auch dann, wenn ihr Name erst zur Laufzeit des Programms bekannt ist.

Was kann ein Objekt?

- Weiß ein Objekt, von welcher Klasse es instantiiert wurde?
- Kann ein Java-Programm herausfinden, von welchem Typ ein Objekt ist?
- Kann ein Java-Programm ein Objekt instantiieren, dessen Klassennamen es erst zur Laufzeit erfährt?
- Kann ein Java-Programm an diesem außer dem Konstruktor auch weitere Methoden aufrufen?

Lösung: Reflection

- Durch das Konzept Reflection (seit JDK 1.1) hat Java folgende Fähigkeiten gewonnen:
- Zur Laufzeit
  - in die Objekte, Klassen, Schnittstellen hineinschauen zu können
  - Methoden, Variablen zu erkennen und darauf einwirken zu können
- Dies unterscheidet Java ganz wesentlich von anderen Programmiersprachen.

Die Klasse `Class`

```
class A
{
   Class classA;

   A()
   {
        classA = getClass();
        System.out.println("Name der Klasse:“
                                        + classA.getName());
   }
}
class TestClass
{
   public static void main(String args[])
   {
        A a = new A();
        System.out.println("oder auch: " + A.class.getName());
   }
}
```


`getClass()` etc.

- Mit der Methode **`getClass()`** der Klasse **`Object`** besitzt ein beliebiges Objekt die Fähigkeit, ein passendes *Klassenobjekt* zu liefern.
- Die Klasse **`Class`** stellt Methoden zur Abfrage von Eigenschaften der Klasse zur Verfügung, z.B.:
  - **`getName()`**
  - Und viele andere
- Die Klasse **`Class`** erlaubt es, Klassen dynamisch zu laden und Instanzen dynamisch zu erzeugen, deren Name zur Compilezeit nicht bekannt ist.

Beispiel: `DynamischesLaden.java`

```
interface HelloMeth
{
  public void hello();
}

class CA implements HelloMeth
{
  public void hello()
  {
    System.out.println("hello CA");
  }
}

class CC
{
  public void hello()
  {
    System.out.println("hello CC");
  }
}
```

Bsp: `DynamischesLaden.java` (2)

```
public class DynamischesLaden
{
  public static void main(String[] args)
  {
    ...
    while (true) {
    try {
      System.out.print("Klassenname oder ende eingeben: ");
      buf = in.readLine();
      if (buf.equals("ende")) {
        break;
      }
      Class c = Class.forName(buf);
      Object o = c.newInstance();
      ((HelloMeth)o).hello();
    } catch (IOException e) {
    ...
    } catch (ClassCastException e) {
    ...
```


Diskussion des Beispiels

- Methoden der Klasse **`Class`**
  - **`static Class forName(String className)`**: lädt eine Klasse mit Namen **`className`** und liefert das liefert das zugehörige Klassenobjekt.
  - Mit dem Klassenobjekt ist es möglich, Informationen über Konstruktoren, Membervariablen und Methoden der Klasse abzufragen.
  - **`Object newinstance():`** instantiiert ein Objekt der Klasse. Dieses kann auf einen bekannten Typ konvertiert werden, um dessen Methoden aufzurufen.

Aufruf von Methoden

- Im folgenden Beispiel sollen zur Compilezeit unbekannte Methoden aufgerufen werden.
- **`public Method[] getMethods()`** aus **`java.lang.Class`** liefert die öffentlichen Methoden.
- **`getDeclaredMethods`** liefert auch die nicht-öffentlichen Methoden.
- Zunächst werden Methoden ohne Parameter betrachtet:

Beispiel: `ParameterloseMethoden.java`

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

class Person1
{
  public void gibName()
  {
    System.out.println("Mustermann");
  }
  public void gibVorname()
  {
    System.out.println("Erika");
  }
} // bitte umblaettern
```

Beispiel: ParameterloseMethoden.java

```
Class c = Class.forName(buf);
Object o = c.newInstance();
Method[] methods = c.getMethods();
for (int i=0; i < methods.length; i++)
  {
       String name = methods[i].getName();
       // Methodenname beginnt nicht mit gib?
       if (!name.startsWith("gib")) continue;
       // Methode parameterlos?
       Class[] params = methods[i].getParameterTypes();
       if (params.length > 0) continue;
       // Methode static?
       int modifiers = methods[i].getModifiers();
       if (Modifier.isStatic(modifiers)) continue;
       System.out.println("Aufgerufen wird die Methode " + name);
       try {
              methods[i].invoke(o, new Object[0]);
       } catch (Exception e) {...
```


java.lang.reflect.Method

- **String getName()**
- **Class[] getParameterTypes()**
- **Object invoke(Object obj, Object[] args)**
- **int getModifiers()**

java.lang.reflect.Modifier

- **static boolean isAbstract(int mod)**
- **static boolean isFinal(int mod)**
- **static boolean isInterface(int mod)**
- **static boolean isPrivat(int mod)**
- **static boolean isPublic(int mod)**
- **static boolean isStatic(int mod)**
- **static boolean isSynchronized(int mod)**
- **...**

Parametrisierte Methoden

- Im kommenden Beispiel wird eine zur Compilezeit unbekannte Methode mit Parametern aufgerufen.
- **public Method getMethod(String name, Class[] parameterTypes)**

Bsp: ParametrisierteMethoden.java

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

class Person2
{
  int alter;

  public int setzeAlter(int alter)
  {
     this.alter = alter;
     return 12*alter;
  }
} // bitte umblättern
```


Bsp: ParametrisierteMethoden.java(2)

```
Class c = Class.forName("Person2");
Object o = c.newInstance();
Class[] formparams = new Class[1];
formparams[0] = Integer.TYPE;
Method method = c.getMethod("setzeAlter", formparams);
Object[] actargs = new Object[1];
actargs[0] = new Integer(37);
System.out.println("Aufgerufen wird setzeAlter(37) bei
                                          Person2");
System.out.println("setzeAlter hat den Returntyp " +
                              method.getReturnType());
try {
  Integer ret = (Integer)method.invoke(o, actargs);
  System.out.println("Rückgabewert: " + ret.intValue());
```


Class[] getParameterTypes()

- liefert ein Array mit Objekten vom Typ **`Class`**, das die Anzahl und Typen der formalen Argumente der Methode enthält.
- Für primitive Typen gibt es statische **`Class`**-Objekte:
  - **`Boolean.TYPE`** **`Character.TYPE`**
  - **`Integer.TYPE`** **`Byte.TYPE`**
  - **`Double.TYPE`** **`Short.TYPE`**
  - **`Float.TYPE`** **`Long.TYPE`**
  - **`Void.TYPE`**