Informatik B

Vorlesung 15
Reflection

Rückblick

- Serialisierung von Objekten
  - Serialisieren
  - Persitenz
  - Deserialisieren
  - **`serialVersionUID`**
  - Speicherung als XML
- *AutoBoxing*

Reflection

- Mit dem Reflection-Modell können Klassen und Objekte zur Laufzeit untersucht und in begrenztem Umfang modifiziert werden
- Das Konzept der Reflection ist besonders bei *JavaBeans,* Hilfsprogrammen zum Debuggen, IDEs oder GUI-Buildern interessant
- Diese Programme heißen auch Meta-Programme, da sie auf den Klassen und Objekten anderer Programme operieren
- Ein Metadatum ist eine Information über eine Information
- In Java beschreibt ein **`class`**-Objekt, was Klassen können: z.B. welche Konstruktoren, Methoden, Attribute sind vorhanden

`Class`-Objekt

- Um ein **`Class`**-Objekt zu erhalten gibt es verschiedene Vorgehensweisen:
  - Ist ein Objekt der Klasse verfügbar, erhält man das **`Class`**-Objekt mit der Methode **`getClass()`** (praktisch, wenn der Typ des Objekts nicht genau bekannt ist)
  - Will man zu einem **`Class`**-Objekt die Oberklasse wissen, kann man diese mit **`getSuperclass()`** erhalten
  - Die Klassenmethode **`Class.forName(String)`** kann eine Klasse erfragen, und liefert das zugehörige **`Class`**-Exemplar als Ergebnis, ist die Klasse noch nicht geladen, bindet **`forName()`** die Klasse ein, aber da das Suchen schief gehen kann, ist eine **`ClassNotFoundException`** möglich

`Class`-Objekt

- Jede Klasse enthält eine Klassenvariable mit Namen **`class`** vom Typ **`Class`**, die auf das zugehörige **`Class`**-Objekt verweist: z.B. **`Exception.class`**
- Auch primitive Datentypen besitzen ein **`Class`**-Objekt, welches man z.B. mit **`int.class`** erhält
- Alternativ erhält man das **`Class`**-Objekt eines primitiven Datentyps mit dem Zugriff auf die statische Variable **`TYPE`** der Wrapper-Klasse: **`Integer.TYPE`**

- Beispiel: **`reflection01`**

Felder erfragen

- Über das **`Class`**-Objekt kann man verschiedene Informationen über eine Klasse bzw. ein Interface erhalten
- Neben den Oberklassen oder den implementierten Interfaces kann man auch die enthaltenen sichtbaren Felder abfragen
- Dazu existiert die Methode **`Field[] getFields()`**
- Jedes Feld kann wiederum nach Eigenschaften befragt werden (Declaring Class, Type, Name, Modifier)
- Beispiel: **`reflection02`**

Methoden erfragen

- Ebenso einfach kann man die sichtbaren Methoden einer Klasse erfragen
- Dazu gibt es die Methode
  `Method[] getMethods()`
- Beispiel: **`reflection03`**
- Auch die Existenz einer bestimmten Methode kann mittels der Signatur und der Methode
  `Method getMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException` bestimmt werden
- Beispiel: **`reflection04`**

Konstruktoren erfragen

- Man kann auch die Konstruktoren einer Klasse erfragen
- Dazu gibt es die Methode **`Constructor[] getConstructors()`**
- Auch die Existenz eines bestimmten Konstruktors kann mittels der Signatur und der Methode **`Constructor getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException`** bestimmt werden
- Beispiel: **`reflection05`**

Objekterzeugung

- Es wurde gezeigt, wie man Metainformationen zu einer Klasse und deren Feldern, Methoden und Konstruktoren erhält
- Mittels Reflection können auch Objekte erzeugt werden
- Dies ist besonders dann sinnvoll, wenn erst zur Laufzeit ein Klassenname bekannt ist

Objekterzeugung

- Um ein Objekt einer bestimmten Klassen dynamisch zu erzeugen, braucht man ein passendes **`Class`**-Objekt
- Mit **`getConstructor(Class<?>... parameterTypes)`** kann man sich dann ein **`Constructor`**-Objekt beschaffen, das den gewünschten Konstruktor beschreibt
- Jedes **`Constructor`**-Objekt besitzt eine **`T newInstance(Object … o)`** Methode, die ein neues Objekt erzeugt
- Bei einem parameterlosen Konstruktor kann man einfach **`newInstance()`** aufrufen
- Beispiel: **`reflection06`**
- Beispiel: **`reflection07`**

Variablenbelegung

- Existiert ein Objekt einer Klasse können die Instanzvariablen nach ihrem Wert befragt werden
- Dazu können mit der Methode **`Field[] getFields()`** alle sichtbaren Felder eines Objektes (auch die geerbten) ermittelt werden
- Mit der Methode **`Field[] getDeclaredFields()`** können Informationen über alle Felder eines Objektes (nicht geerbte Felder) abgefragt werden
- Von diesem **`Field`**-Objekt können dann die Werte erfragt werden
- Dazu gibt es verschiedene get-Methoden für die primitiven Datentypen, bzw. eine get-Methode für referenzierte Objekte

Variablenbelegung

- Wird mit der Methode **`getDeclaredFields()`** ein **`Field`**-Array über alle Felder geliefert, können von jedem der Array-Einträge Metainformationen erfragt werden
- Ist eine Variable aufgrund eines Modifiers aber nicht zugänglich, kann der Wert der Variablen nicht erfragt werden
- Ein Versuch führt zu einer **`IllegalAccessException`**
- Beispiel: **`reflection08`**

Variablenbelegung

- Ebenso wie das Abfragen von Werten ist auch ein Setzen von Variablen über Reflection möglich
- Dazu muss wiederum zunächst das **`Field`**-Objekt ermittelt werden
- An diesem können diverse set-Methoden aufgerufen werden, um einen Wert zu setzen
- Es sind set-Methoden für primitive Datentypen und für Objekte vorhanden
- Beispiel: **`reflection09`**

Zugriff auf private Variablen

- Mit einem kleinen Umweg kann man jedoch zur Laufzeit mittels Reflection auf private Felder zugreifen und diese auslesen und verändern
- Dazu kann die Methode **`setAccessible`** der Klasse **`AccessibleObjekt`** verwendet werden
- Als Attribut erwartet die Methode ein Objekt vom Typ **`AccessibleObjekt`** und einen **`boolean`** Wert, der angibt, oder Zugriff erlaubt sein soll oder nicht
- **`AccessibleObject`** ist die Oberklasse von **`Constructor`**, **`Field`** und **`Method`**
- Beispiel: **`reflection10`**

Zugriff verhindern

- Der Zugriff auf private Datenfelder ist als kritisch zu sehen
- Dadurch werden Wege geöffnet an sensible Daten zu gelangen
- Verhindern kann man dies nur, indem man beim Start des Programmes angibt, dass ein *SecurityManager* verwendet werden soll
- Dazu wird als JVM-Parameter folgendes angegeben: **`-Djava.security.manager`**
- Dann werden Zugriffe der **`setAccessible`** Methode abgewiesen
- Beispiel: **`reflection10`** (mit entsprechendem Parameter)

Aufruf von Instanzmethoden

- Wenn zur Compile-Zeit der Name der Methode nicht feststeht, lässt sich zur Laufzeit eine im Programm definierte Methode aufrufen, wenn ihr Name als Zeichenkette vorliegt
- Man geht wieder von einem **`Class`**-Objekt aus, für das eine Instanzmethode aufgerufen werden soll
- Anschließend wird ein **`Method`**-Objekt mit der Methode **`getMethod()`** beschafft
- **`getMethod()`** verlangt zwei Argumente:
  - Einen String mit dem Namen der Methode
  - Ein Array von **`Class`**-Objekten die die Parameterliste angeben

Aufruf von Instanzmethoden

- **`invoke()`** ruft die Methode auf (*dynamic invocation*)
- **`invoke()`** erwartet zwei Argumente:
  - Ein Array mit Parametern, die der aufgerufenen Methode übergeben werden
  - Eine Referenz auf das Objekt, an dem die Methode aufgerufen werden soll
- Wie auch bei privaten Datenfeldern können auch **`private`** Methoden aufgerufen, wenn kein *SecurityManager* gesetzt ist
- Beispiel: **`reflection11`**

Aufruf von Klassenmethoden

- Der Aufruf einer Klassenmethode erfolgt analog zum Aufruf einer Instanzmethode
- Es muss ein **`Class`**-Objekt existieren
- Mit **`getMethod()`** wird eine Methode anhand des Namens und der Parameterliste herausgesucht
- Mit der Methode **`invoke()`** wird die Methode ausgeführt, wobei als erstes der Parameter **`null`** angegeben wird, da diese Methode nicht an ein Objekt gebunden ist, die weiteren Parameter sind die entsprechenden Argumente
- Beispiel: **`reflection12`**

Zusammenfassung

- Reflection
  - Beschaffen des **`Class`**-Objektes
  - Ermitteln der Metainformationen
  - Erzeugen von Objekten
  - Aufruf von Methoden
  - Auslesen und setzen von Variablen
  - Zugriff auf **`private`** Datenfelder und Methoden
  - Schutz vor dem Zugriff auf **`private`** Eigenschaften einer Klasse

Ausblick

- Netzwerkprogrammierung