Reflection

Programmiermethodik 2

Zum Nachlesen:
Christian Ullenboom: Java ist auch eine Insel, Kapitel 25:
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_25_001.htm

Wiederholung

- Innere Klassen
    - Mitgliedsklasse
    - Anonyme Innere Klasse
    - Weitere Typen
- Ereignisverarbeitung in JavaFX
    - Ereignisse und Event-Handler
    - Ereignis-Ursachen
    - Event-Handler-Umsetzungen

Ausblick

Agenda

- Einführung
- Eigenschaften einer Klasse
- Objektvariablen, Methoden und Konstruktoren
- Objekte Erzeugen und Manipulieren
- Methoden Aufrufen

Metadaten und Reflection

Einführung

- Reflection-Modell
    - Klassen und Objekte, die zur Laufzeit von der JVM im Speicher gehalten werden, zu untersuchen
    - in begrenztem Umfang zu modifizieren
- Anwendung
    - Hilfsprogrammen zum Debuggen
    - GUI-Builder
- solche Programme heißen: Metaprogramme
    - operieren auf Klassen und Objekten anderer Programme
- Reflection fällt daher auch in die Schlagwortkategorie »Meta-Programming«

Metadaten

- Ein Metadatum ist eine Information über eine Information.
- Beispiel: In Java beschreibt ein Class-Objekt, was Klassen »können«, also welche Konstruktoren und Methoden sie haben, welche Attribute sie besitzen und wie die Erweiterungsbeziehungen sind.

Metadaten durch JavaDoc-Tags

- Annotationen sind Metadaten
- Beispiele:

```
@Deprecated
public void setDate(int date) {
     getCalendarDate().setDayOfMonth( date );
}

@Override
public String toString() {
     return ...
}
```

Motivation

- Use Case:
  - Entwicklung eines Klassen-Browsers
  - soll Informationen zum laufenden Programm anzeigen:
    - Klassen, Variablenbelegung, deklarierte Methoden, Konstruktoren und Informationen über die Vererbungshierarchie
  - (siehe BlueJ)
- Unterschied zwischen Java und vielen herkömmlichen Programmiersprachen
  - Abfrage von Eigenschaften von Klassen vom gerade laufenden Programm mittels der Class-Objekte
  - Zugriff auf Metadaten explizit vorgesehen: Reflection

Eigenschaften einer Klasse

Bestimmen der Klasse

- Java: Bibliotheksklasse Class
  - Exemplare der Klasse Class sind Objekte
  - repräsentieren entweder eine Java-Klasse oder Java-Schnittstelle
  - besondere Form von Meta-Objekten
  - Beschreibung einer Java-Klasse, die aber nur ausgewählte Informationen preisgibt
- Class-Objekte selbst kann nur die JVM erzeugen
- Objekte sind unveränderlich und der Konstruktor ist privat
- Möglichkeiten zum Zugriff:
  - Methode 1: getClass()
  - Methode 2: Class.forName(String)
  - Methode 3: getSuperClass()

getClass()

- Exemplar der Klasse verfügbar: getClass()-Methode des Klassen-Objekts
- Jede Klasse enthält eine Klassenvariable mit Namen .class vom Typ Class, die auf das zugehörige Class-Exemplar verweist.
- Signatur: final Class<? extends Object> getClass()
- Beispiele:

```
Class<?> klasse1 = java.util.ArrayList.class;
Class<?> klasse2 = new java.util.ArrayList<String>().getClass();
```

Class.forName()

- Klassenmethode Class.forName(String) kann eine Klasse erfragen
  - liefert das zugehörige Class-Exemplar als Ergebnis
  - kannn ClassNotFoundException werfen
- Signatur:

```
static Class<?> forName( String className ) throws
    ClassNotFoundException
```

- Beispiel:

```
try {
  Class<?> klasse3 = Class.forName("java.util.ArrayList");
  System.out.println(klasse3);
} catch (ClassNotFoundException e) {
}
```

getSuperClass()

- Class-Objekt existiert bereits
    - Interesse an Vorfahren
- Erhalten eines Class-Objekts für die Basisklasse mit getSuperclass()
- Beispiel:

```
ArrayList.class.getSuperclass()
→ java.util.AbstractList
```

Klassen-Namen

- Class-Objekt zu einer Klasse zur Laufzeit
  - Anfrage des voll qualifizierten Namens
  - Methode getName()
  - jeder Typ hat Namen → Methode immer erfolgreich
- Beispiel:

```
new java.util.ArrayList().getClass().getName();
→ java.util.ArrayList
```

Welchen Zugriff verwenden?

- forName() ist sinnvoll, wenn der Klassenname bei der Übersetzung des Programms noch nicht feststand
- ansonsten: getClass()/.class eingängiger
    - und Compiler kann prüfen, ob es den Typ gibt
- Klassenobjekte für primitive Elemente liefert forName() nicht

```
Class.forName("boolean")
```

    - und

```
Class.forName(boolean.class.getName())
```

    - führen zu einer

```
java.lang.ClassNotFoundException
```

Unterscheidung: Klassen, Schnittstellen, Arrays

- Class-Exemplar verkörpert verschiedene Formen
  - Interface
  - Klasse
  - primitiver Datentyp
  - Array-Typ
- Abfrage mit
  - isInterface()
  - isPrimitive()
  - isArray()
  - alle false → gewöhnliche Klasse

Primitive Datentypen

- Konstante TYPE in den acht Wrapper-Klassen zu boolean, byte, char, short, int, long, float und double
- Zugriff auf das Class-Objekt für den primitiven Typ int
    - Integer.TYPE oder
    - int.class
- void (obwohl kein Typ)
    - System.out.println(void.class.isPrimitive()); // true
- Auch auf primitiven Datentypen ist das Ende .class erlaubt
    - Class-Objekt liefert die statische Variable TYPE der Wrapper-Klassen
    - also: int.class == Integer.TYPE

Primitive Datentypen

- Beispiel: Prüfen des Typen

```
if (klassenObjekt.isArray()) {
  System.out.println(klassenObjekt.getName() + ": Feld.");
} else if (klassenObjekt.isPrimitive()) {
  System.out.println(klassenObjekt + ": primitiver Typ.");
} else if (klassenObjekt.isInterface()) {
  System.out.println(klassenObjekt.getName() + ": Interface.");
} else {
  System.out.println(klassenObjekt.getName() + ": Klasse.");
}
```

Interfaces

- Zugriff mit getInterfaces()
    - liefert Array von Class-Objekten
- Name der Schnittstelle: getName()
- Beispiel

```
Class<?>[] interfaces = ArrayList.class.getInterfaces();
for (Class<?> iface : interfaces) {
  System.out.println(iface);
}
```

Ausgabe:
interface java.util.List
interface java.util.RandomAccess
interface java.lang.Cloneable
interface java.io.Serializable

Modifizierer

- Zugriff mit getModifiers()
  - liefert die (kombinierte Menge aller) Modifizierer verschlüsselt als Ganzzahl
- Beispiel

```
ArrayList.class.getModifiers(); // → 1
Modifier.toString( Modifier.class.getModifiers()) ); // → public
```

- Modifizierer können über Test-Methoden der Klasse java.lang.reflect.Modifier überprüft werden

```
static boolean isAbstract( int mod )
static boolean isFinal( int mod )
static boolean isInterface( int mod )
...
```

Modifizierer

- Hinweis: Schnittstellen, wie `java.io.Serializable`, tragen den Modifier `abstract`.
- Beispiel:

```
int modifier = Serializable.class.getModifiers();
System.out.println( modifier ); // → 1537
System.out.println( Modifier.toString(modifier) );
// → public abstract interface
```

Arrays

- Spezielle Methoden stehen für Arrays zur Verfügung
- siehe Literatur, z.B. "Java ist auch eine Insel"

Übung: Klassenobjekt prüfen

- Schreiben Sie eine Methode pruefeKlassenObjekt()
- Methode bekommt einen Parameter vom Typ Class<?>
- in der Methode
    - Ausgabe des Namens der Class-Objektes
    - falls es sich bei dem Objekt um eine Klasse handelt
    - Ausgabe der Interfaces, die es implementiert
- Aufruf der Methode für das folgende Objekt:
- LocalDate datum = LocalDate.now();

Objektvariablen, Methoden und Konstruktoren

Exceptions

- Reflection ist sehr dynamisch
  - viele Fehler möglich
  - nahezu alle Methoden zum Zugriff auf Laufzeitinformationen können `Exceptions` auslösen
- Reflection-spezifische Exception-Typen
  - `NoSuchFieldException und NoSuchMethodException`
  - `ClassNotFoundException`
  - `InstantiationException`
  - `IllegalAccessException`
  - `InvocationTargetException`

Objektvariablen

- Anwendungen: Werte auszulesen und verändern
- Lesen für Konstanten und Variablen gleichermaßen erlaubt

- Variablen aus Klasse bzw. aus Oberklassen geerbten:

  `getFields()`
  - Zugriff nur auf öffentliche (`public`) Elemente mit (gewöhnlicher) Reflection
  - kein schreibender Zugriff bei Schnittstellen (können nur Konstanten deklarieren)

- Zugriff auf die Felder, die direkt in der Klasse deklariert werden

  `getDeclaredFields()`
  - keine ererbten Felder
  - Zugriff auch auf private Felder

Zugriff auf Objektvariablen

- Ergebnis: Array von Field-Objekten
  - jeder Array-Eintrag beschreibt eine Objekt- oder Klassenvariable
  - Einträge sind unsortiert

```
Field[] variablen = fahrzeug.getClass().getFields();
System.out.println("Öffentliche Felder:");
for (Field variable : variablen) {
  System.out.println("  " + variable.getName() + variable.getType());
}
```

- Alternative: Zugriff auf ein bestimmtes Feld mit

```
getField( String name ) throws NoSuchFieldException
getDeclaredField( String name ) throws
    NoSuchFieldException
```

- Objektvariable = Typ + Bezeichner

```
getType(), getName()
```

Methoden

- analog zu den Feldern:
  - Array mit `Method`-Objekten
- öffentlich sichtbare Methoden (auch ererbt):
  - `getMethods()`
- alle Methoden der Klasse (auch privat):
  - `getDeclaredMethods()`
- Alternative
  - Zugriff auf einzelne Methode mit Name/Parameterliste

```
Method[] methoden = fahrzeug.getClass().getMethods();
System.out.println("Öffentlich sichtbare Methoden:");
for (Method methode : methoden) {
  System.out.println("  " + toString(methode));
}
```

Methoden

- Parameter werden als Objekte vom Typ Parameter repräsentiert
    - haben Typ und Name
- Rückgabewert wieder als Class<?>-Objekt
- Beispiel (Generieren einer Methodensignatur):

```
String signatur = methode.getReturnType() + " " + methode.getName() + "(";
for (int i = 0; i < methode.getParameters().length; i++) {
  Parameter param = methode.getParameters()[i];
  signatur += param.getType() + " " + param.getName();
  if (i < methode.getParameters().length - 1) {
    signatur += ",";
  }
}
signatur += ")";
```

Konstruktoren

- Konstruktoren und Methoden haben einige Gemeinsamkeiten
- aber unterschiedlich: Konstruktoren haben keinen Rückgabewert
- Zugriff auf Array von `Constructor`-Objekten

```
Constructor[] getConstructors()
```

- Zugriff auf einen Konstruktor

```
Constructor<T> getConstructor(Class...
    parameterTypes)throws NoSuchMethodException
```

- liefert für jeden Konstruktor
  - Name
  - Modifizierer
  - Parameter
  - Exceptions

Konstruktoren

```
for (Constructor<?> c : ArrayList.class.getConstructors()) {
  System.out.println(c);
}
```

- Ausgabe:

public java.util.ArrayList(java.util.Collection)

public java.util.ArrayList()

public java.util.ArrayList(int)

Übung: Werte abfragen

- Schreiben Sie Code mit folgende Funktionalität:
  - Geben Sie die Bezeichner und Typen aller Objektvariablen, die in der Klasse Bruch definiert sind, aus.
  - Geben Sie die Bezeichner und Rückgabetypen aller öffentlich sichtbaren Methoden, die in der Klasse Bruch definiert sind, aus.

Objekte Erzeugen und Manipulieren

Objekte Erzeugen und Manipulieren

- Bisher: Abfrage von Informationen zu
  - Klassen
  - Variablen
  - Methoden
  - Konstruktoren
- nächster Schritt:
  - Objekte erzeugen
  - Werte von Variablen abfragen und verändern
  - Methoden dynamisch per Reflection aufrufen

Instanzen Erzeugen

- Erzeugen eines Objektes zur Laufzeit:
    - new-Operator
- Compiler muss den Namen der Klasse kennen
- Problem: Falls Name der gewünschten Klasse erst zur Laufzeit bekannt
    - new-Operator kann nicht verwendet werden

Instanzen Erzeugen

- Dynamisches Erzeugen von Exemplaren bestimmter Klassen
  - passendes Class-Objekt benötigt
- Vorgehen
  - Holen eines Konstruktor-Objekts: getConstructor()

```
Constructor<T> getConstructor( Class... parameterTypes )
    throws NoSuchMethodException
```

  - Verwenden der Methode newInstance(Object[])

```
T newInstance( Object... initargs ) throws
    InstantiationException, IllegalAccessException,
    IllegalArgumentException, InvocationTargetException
```

  - Erschaffen eines neuen Exemplars
  - Parameter von newInstance() ist ein Feld von Werten, die an den echten Konstruktor gehen
  - parameterlosen Konstruktor: newInstance(null)

Beispiel

- Hinweis: Verzicht auf Ausnahmebehandlung zur besseren Übersicht

```
Class<Point> pointClass = Point.class;
Constructor<Point> constructor =
     pointClass.getConstructor( int.class, int.class );
Point p = constructor.newInstance( 10, 20 );
System.out.println( p );
```

Belegung von Variablen erfragen

- teilweise nicht ausreichend, nur Namen und Datentypen einer Variablen zu kennen
- Gewünscht: lesender und schreibender Zugriff auf ihre Inhalte
- Vorgehen
    - Class-Objekt erfragen
    - Array von Attributbeschreibungen mit getFields() oder getField(String)
    - Zugriff auf den Variablenwert mit get()
        - möglich für alle Variablentypen möglich
        - Konvertierung in Wrapper-Objekte für primitive Typen war

Variablenwerte Auslesen

- Alternative zu get()
  - spezielle getXXX()-Methoden für primitive Typen
  - z.B. getDouble(), getInt()
  - falscher Zugriff: IllegalArgumentException, IllegalAccessException
- getXXX()-Methoden zum Erfragen erwarten ein Argument mit dem Verweis auf das Objekt, welches die Variable besitzt
  - Argument wird bei statischen Variablen ignoriert

Übung: Rechteck

- Ein Programm soll ein Rechteck-Objekt mit einer Belegung 23, 42 (Höhe und Breite) erzeugen
- Aufgabe: Auslesen der Höhe hoehe (Typ: double)

Variablen Setzen

- Setzen der Werte von Variablen
- Analoges Vorgehen: setXXX() statt getXXX()
- z.B: setBoolean(), setDouble()
- Allgemeine set()-Methode für Objektreferenzen
    - kann auch mit Wrapper-Objekten für Variablen von primitiven Datentypen umgehen
- im Fehlerfall: IllegalArgumentException, IllegalAccessException

Variablen Setzen

- Beispiel: Erzeugen eines Point-Objekts mit dem Konstruktor, der die Koordinaten 23 und 42 setzt
- Aufgabe: Veränderung der zweiten Koordinaten auf -1

Änderung Privater Attribute

- Möglichkeit private- oder protected
  - Attribute zu ändern
  - Methoden und Konstruktoren eingeschränkter Sichtbarkeit aufrufen
- nur wenn der Sicherheitsmanager es zulässt
- dazu notwendig: Oberklasse java.lang.reflect.AccessibleObject
  - diese vererbt an Constructor, Field und Method die Methode setAccessible(boolean)
  - Argument vererbt auf true setzen
  - Achtung: nicht immer möglich!

Methoden Aufrufen

Methoden Aufrufen

- letzter Schritt: Aufrufen von Methoden per Reflection
- Aufrufen der Methode anhand des Namens als Zeichenkette
    - wenn zur Compile-Zeit der Name der Methode nicht feststeht

Vorgehen

- Class-Objekt aus, das die Klasse des Objekts beschreibt
- Bedarf für ein Method-Objekt als Beschreibung der gewünschten Methode
- Zugriff über die Methode getMethod()
    - Argument 1: String mit dem Namen der Methode
    - Argument 2: Array von Class-Objekten, je ein Element pro einem Parametertyp aus der Signatur der Methode
- Aufrufen der Zielmethode mit invoke()
    - Fachbegriff: dynamic invocation
    - Argument 1: ein Array mit Argumenten, die der aufgerufenen Methode übergeben werden
    - Argument 2: und eine Referenz auf das Objekt, auf dem die Methode aufgerufen werden soll und zur Auflösung der dynamischen Bindung dient.

Beispiel

- Erzeugen ein Point-Objekts
- Initialisierung mit den Koordinaten 23 und 42
- Aufgabe: Dynamische Abfrage der x-Koordinate über die Methode getX()

```
Point p = new Point( 10, 0 );
Method method = p.getClass().getMethod( "getX" );
String returnType = method.getReturnType().getName();
Object returnValue = method.invoke( p );
System.out.printf( "(%s) %s", returnType, returnValue );
// → (double) 10.0
```

Zusammenfassung

- Reflection
- Metadaten von Klassen
- Attribute, Methoden und Konstruktoren
- Objekte Erzeugen und Manipulieren
- Methoden Aufrufen