In diesem Kapitel:

- *Wrapper Klassen*
  1. void
  2. Boolean
  3. Character
  4. Number
  5. Integer
  6. Floating Point / Fliesskomma
- *Reflection*
  1. Class
  2. Modifier
  3. Field
  4. Method
  5. Constructor
  6. Proxy
- *Laden von Klassen*

Programmieren mit Typen

I'm gonna wrap myself in papers,
I'm gonna dab myself with glue -
Stick some stamps on top of my head!
I'm gonna mail myself to you.
- Woody Guthrie, *Mail Myself to You*

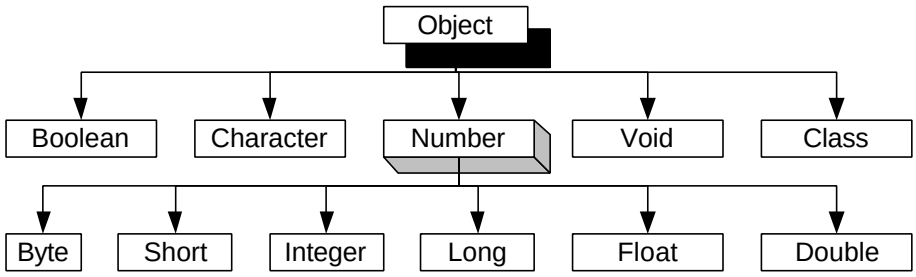
Typen werden durch Klassen und Interfaces repräsentiert. Für alle primitiven / Basis-Datentypen (byte, float, int und so weiter) existiert eine Klasse, welche diesen Typ repräsentiert. Zudem existiert eine Klasse Class, welche die anderen Klassentypen und Interfaces repräsentiert, sozusagen als Metaklasse. Diese Klassen haben drei Vorteile:

- Methoden, die zu den Datentypen gehören, inklusive Konversionsmethoden, haben einen klar definierten Platz, ein zuhause. Beispielsweise sind die Methoden, mit deren Hilfe eine Zeichenkette in eine float Zahl umgewandelt werden kann, statische Methoden der Float Klasse.

- beschreibende Methoden und Felder haben ein Zuhause. MIN_VALUE und MAX_VALUE Konstanten sind für die jeweiligen Basisdatentypen in den korrespondierenden Klassen definiert.
  Methoden der Klasse Class gestatten es Ihnen Objekte zu prüfen, zu verändern und zu manipulieren.

- für Basisdatentypen kann man Wrapper-Objekte kreieren, welche die Werte der Basisdatentypen aufnehmen können. Diese Objekte können überall eingesetzt werden, wo auch andere Objekte einsetzbar sind. Daher bezeichnet man diese Klassen als *Wrapper Klassen*.

Die Typenhierarchie für diese Klassen sieht folgendermassen aus:



In diesem Kapitel befassen wir uns mit diesen Typenklassen und mit deren Möglichkeiten. Im ersten Teil beschreiben wir die Wrapper Klassen der Basisdatentypen. Im zweiten Teil untersuchen wir, wie mit Hilfe der *Reflection* Klassen und Interfaces, inklusive der Klasse Class gearbeitet werden kann, um Details über Objekte zu bestimmen und Objekte dynamisch zu kreieren, deren Member Funktionen (Methoden) und Attribute zu bestimmen und einzusetzen. Am Schluss des Kapitels kommen wir nochmals auf den Ladevorgang, die Class Loader zurück. Mit Hilfe der Class Loader werden die Klassen in die Virtual Machine geladen. Wir möchten uns mit der Frage befassen, wie man selber Class Loader für Spezialanwendungen schreibt.

11.1. Wrapper Klassen

Alle primitiven Datentypen besitzen Wrapper Klassen, welche diesen Datentyp repräsentieren. Diese Klassen dienen zwei Zwecken.

Als erstes dienen sie als Zuhause für Methoden und Variablen, die mit diesem Datentyp zusammenhängen (Zeichenumwandlung, Datenbereiche, minimale und maximale Werte). Als Beispiel sei hier eine mögliche Überprüfung einer Berechnungsmethode erwähnt:
falls der Wert es gestattet (der Absolutwert ist innerhalb der Float Bereiches), wird eine schnelle Berechnungsmethode verwendet, sonst eine langsamere und genauere (Double)

```
double aval = Math.abs(value);
if (Float.MAX_VALUE >= aval && aval >= Float.MIN_VALUE)
      return fasterFloatCalc( (float)value );
else
      return slowerDoubleCalc(value);
```

Als zweites kann die Wrapper Klasse eingesetzt werden, um die Werte der Basisklasse aufzunehmen. Damit kann der Basisdatentyp auch dann eingesetzt werden (besser : seine Darstellung mit Hilfe der Wrapper Klasse), falls ein Objekt verlangt wird.

Die HashMap Klasse speichert beispielsweise lediglich Objektreferenzen, keine Basisdatentypen (die Klasse HashMap werden wir später noch detailliert besprechen). Falls wir eine int Zahl in einem HashMap verwenden möchten, müssen wir diese Variable wrappen, mit Hilfe der Integer Klasse.

```
int key = ...;
Integer keyObj = new Integer(key);
map.put(keyObj, value);
```

Die Reflection Klassen befassen sich, wie wir später sehen werden, auch mit generischen Objekten und benutzen die Wrapper Klassen für die Basisdatentypen.

In den folgenden Abschnitten besprechen wir Methoden und Konstanten der einzelnen Wrapper Klassen. Aber zuerst schauen wir uns einige Gemeinsamkeiten der Wrapper Klassen an.

Jede Wrapper Klasse definiert unmutierbare Objekte für Werte der Basisdatentypen, die sie wrapped / ummantelt. Das heisst, dass nach dem Kreieren eines Wrapper Objekts der Wert der durch dieses Objekt repräsentiert wird, nicht mehr verändert werden kann.

Beispielsweise beschreibt das Objekt

```
Integer intObj1 = new Integer(1)
```

eine int Zahl mit dem Wert 1. Die Wrapper Klasse Integer besitzt aber keine Methode, mit der der Wert dieses Objekts verändert werden könnte.

Alle Wrapper Klassen besitzen folgende Konstruktoren:

- einen Konstruktor, der den Wert eines Basisdatentyps als Parameter enthält und ein Objekt der entsprechenden Wrapper Klasse konstruiert.
  Beispielsweise konstruiert der Konstruktor Character(char1) ein Objekt mit dem Wert char1.

- einen Konstruktor, der eine einfache Zeichenkette als Parameter in den Initialwert des konstruierten Objekts umwandelt:

  new Float("6.02e23")

  Dies gilt jedoch nicht für die Character Wrapper Klasse. Diese besitzt diesen Konstruktor aus naheliegenden Gründen nicht.

  Für numerische Basisdatentypen wird angenommen, dass die Zeichendarstellung zur Basis 10 gehört. Falls der Wert nicht entschlüsselt werden kann, wird eine NumberFormatException geworfen.

Der Begriff *Radix* wird im Rahmen der Wrapper Klassen als Synonym für numerische Basis (Basis 10 / Zehnerdarstellung, ...) verwendet. Beispielsweise bedeutet "dekodieren in Radix 8" dasselbe wie "dekodieren in der Basis 8".

Jede der Wrapper Klassen besitzt auch folgende Methoden:

public static Type valueOf(String str)
: liefert ein neues Objekt des Typs Type mit einem Wert, der aus der Zeichenkette str hergeleitet wird. Falls es sich um einen numerischen Typ handelt, wird Radix 10 angenommen.
  Beispiele:
  Floar.valueOf("6.02e23"); Integer.valueOf("16");

Äquivalent dazu kann man einen Konstruktor mit new aufrufen und die Zeichenkette als Parameter angeben.

Falls str nicht dekodiert werden kann, wird eine NumberFormatException geworfen. Die Character Klasse kennt diese Methode nicht, da es auch keinen Konstruktor mit Zeichenkettenparameter gibt.

Boolean interpretiert die Zeichenkette str auf eigene Art und Weise!

public String toString()
Überschreibt Object.toString() und liefert eine Zeichenkettendarstellung des gewrappten Wertes.

public type typeValue()
Liefert den Wert des entsprechenden Wrapper Objekts.
new Integer(6).intValue(6) liefert beispielsweise 6.

public int compareTo(Type other)
Liefert einen Wert kleiner, grösser oder gleich Null, je nachdem das Objekt auf dem die Methode ausgeführt wird, kleiner, grösser oder gleich dem Objekt des selben Typs ist, welches als Parameter angegeben wird.
Die Methode existiert in der Klasse Boolean nicht.

public int compareTo(Object obj)
Falls obj vom selben Typ wie dieses Objekt ist, auf dem die Methode ausgeführt wird, dann agiert die Methode genau wie compareTo(Type).
Sonst wird eine ClassCastException geworfen.
Diese Methode implementiert die entsprechende Methode im Compare Interface (darauf kommen wir später).
Die Methode existiert in der Klasse Boolean nicht.

public boolean equals(Object obj)
Liefert true, falls die zwei Objekte (in obj1.equals(obj2)) vom selben Typus sind und den selben Wert wrappen.
Beispiel:
für zwei Integer x,y Objekte liefert x.equals(y) true genau dann, falls x.intValue() ==y.intValue()
Falls obj nicht vom selben Typ ist wie das aktuelle Objekt, oder null, liefert die Methode false.

public int hashcode()
liefert einen wertmässigen Hashcode zum Einsatz in Hashtabellen.

Alle Wrapper Klassen besitzen diese Methoden. Daher werden wir diese in der folgenden Auflistung der Methoden pro Klasse nicht nochmals auflisten!

Jede Wrapper Klasse besitzt ein statisches TYPE Datenfeld, das Class Objekt für den Basisdatentyp. Im Falls der Klasse Integer (JDK Dokumentation):
"public static final Class **TYPE :** The Class object representing the primitive type int."

11.1.1. Void

Die Void Klasse ist eine Ausnahme in jeder Beziehung, weil sie keinen Wert umschliesst, keine Methode zur Verfügung stellt und nicht instanziert werden kann. Die Klasse besitzt einzig ein statisches TYPE Datenfeld, welches eine Referenz auf das Class Objekt enthält, welches man mittels void.class erhält. Java kennt keinen void Datentyp - void ist ein Platzhalter, mit dem angezeigt werden kann, dass eine Methode keinen Rückgabewert und Rückgabetyp besitzt. Die Void Klasse repräsentiert diese "Leere"; die Klasse wird eigentlich nur im Reflection Interface benötigt (darauf kommen wir noch).

11.1.2. Boolean

Die Boolean Klasse repräsentiert den boolean Datentyp als Klasse. Sowohl der Konstruktor mit einer Zeichenkette als Parameter, als auch die valueOf Methode interpretieren "true" und jede Mischung davon in Gross- und Kleinbuchstaben ("True", "tRue", ...). Jede andere Zeichenkette wird als false interpretiert (Beispielsweise: new Boolean("Hallo");).

Die Boolean Klasse besitzt zwei statische Referenzen auf Objekte, welche dem Basisdatentyp boolean zugeordnet sind: Boolean.TRUE und Boolean.FALSE.

Boolean Objekte sind nicht vergleichbar: der Vergleichsoperator macht keinen Sinn bei boolean Variablen.

11.1.3. Character

Die Character Klasse repräsentiert den char Datentyp als Klasse. Die Klasse stellt Methoden zur Bestimmung der Zeichen und deren Konversion zwischen Grossbuchstaben und Kleinbuchstaben zur Verfügung. Viele der Methoden werden mit Hilfe einer "Unicode Attribute Table" zur Verfügung gestellt. Diese ordnet jedem Unicode Zeichen einen Namen zuordnet.

Zusätzlich werden die Konstanten MIN_VALUE und MAX_VALUE definiert. Die Klasse Character stellt Konstanten MIN_RADIX und MAX_RADIX zur Verfügung, welche die minimalen und maximalen Radices darstellen, welche bei einer Umwandlung zwischen digitalen Zeichen und integer Werten interpretiert werden können. Radix muss zwischen 2-36 (MIN_RADIX und MAX_RADIX) sein. Werte grösser als 9 sind die Zeichen A bis Z oder die entsprechenden Kleinbuchstaben.

Die Klasse stellt drei Konversionsmethoden zwiscen Zeichen und Integer Werten zur Verfügung:

public static int digit(char ch, int radix)

> Diese Methode liefert den Wert von ch als Zahl zur gegebenen Zahlenbasis radix. Falls radix ungültig ist, oder falls ch in der Zahlenbasis radix nicht darstellbar ist, wird -1 zurück geliefert.
> Beispiel:
> digit('A', 16) liefert den Wert 10
> digit('9', 10) liefert 9 und schliesslich
> digit('a',10) liefert den Wert -1.

public static int getNumericValue(char ch)

Liefert den nummerischen Wert des Zeichens ch gemäss Unicode Attribute Table.
Beispielsweise liefert das Zeichen \u217F das römische Zeichen (Zahl) M (Mille...), stellt also die Zahl 1000 dar! Der Aufruf von getNumericValue('\u217F') liefert also 1000.
Die nummerischen Werte sind alle nicht negativ.
Falls ch keinen nummerischen Wert darstellt, wird -1 als Rückgabewert geliefert.
Falls ch einen nummerischen Wert darstellt, aber keinen ganzzahligen, wird -2 zurück geliefert.
Beispielsweise liefert '\u00bc' den Wert -2, falls man die Methode darauf anwendet.

public static char forDigit(int digit, int radix)

Liefert das Zeichen für diese spezielle Zahl im entsprechenden Radix.
Falls die Zahl ungültig ist (im Radix) wird '\u0000' zurück geliefert.
Beispielsweise liefert forDigit(10,16) 'a', während forDigit(16,16) '\u0000' liefert.

In Unicode gibt es drei Zeichentypen: gross, klein und Titel. Grossbuchstaben und Kleinbuchstaben kennen die meisten Leute, aber Titel ist etwas Spezielles. Titel ist aber sehr praktisch: damit kann man beispielsweise den ersten Buchstaben in einer Überschrift speziell kennzeichnen. Traditionell wird der erste Buchstabe in einer Überschrift gross geschrieben. Beispielsweise soll in der Zeichenkette "ljepotica" der erste Buchstaben lj (Unicode \u01C9) ein Zeichen im erweiterten lateinischen Zeichensatz , welches in kroatisch eingesetzt wird.

Falls das Wort in einem Buchtitel erscheint, sollte der erste Buchstabe in jedem Wort gross geschrieben werden. Dies können Sie mit toTitleCase() angewandt auf das erste Zeichen, erreichen.

Damit erhalten wir "Ljepotica" (Lj ist \u01C8). Falls Sie die Methode toUpperCase() einsetzen würden, würden Sie "LJepotica" (als LJ oder Unicode \u01C7) erhalten.

Alle Gross-Kleinschreib-Regeln in Java entsprechen jenen im Unicode.

Stets gilt der Einheitsoperator (keine Wirkung):

Character.toUpperCase(Character.toLowerCase(ch));

unabhängig von Zeichensatz.

Die Gross/Klein/Titel Umwandlungsmethoden sind:

public static char toLowerCase(char ch)
Liefert Kleinbuchstaben zum Zeichen ch. Falls es keinen Kleinbuchstaben zu ch gibt, wird ch einfach zurückgeliefert.

public static char toUpperCase(char ch)
Liefert Grossbuchstaben zum Zeichen ch. Falls es keinen Grossbuchstaben zu ch gibt, wird einfach ch zurückgeliefert.

public static char toTitleCase(char ch)
Liefert Titelbuchstaben zum Zeichen ch. Falls es keinen Titelbuchstaben zu ch gibt, wird einfach ch zurückgeliefert.

Die Character Klasse verfügt über viele Methoden, mit deren Hilfe getestet werden kann, ob ein Zeichen zu einem bestimmten Typus gehört. Diese Methoden akzeptieren ein Zeichen char als Parameter und liefern einen boolean Wert, als Antwort auf die Anfrage.

Diese Methoden sind:

Methode	**Ist das Zeichen ...?**
isDefined	ist ch ein Unicodezeichen?
isDigit	ist ch eine Zahl in irgend einem Unicode Character Set
isIdentifierIgnorable	kann ein Zeichen im Programmcode oder Unicode sein
isISOControl	ein Latin-1 Steuerzeichen
isJavaIdentifierPart	gültig nach dem ersten Zeichen eines Bezeichners
isJavaIdentifierStart	gültig als erstes Zeichen eines Bezeichners
isLetter	ein Buchstabe in irgend einem Unicode Zeichensatz
isLetterOrDigit	ein Buchstabe oder eine Zahl aus irgend einem Unicode Zeichensatz
isLowerCase	ein kleingeschriebener Buchstabe
isSpaceChar	ein Leerzeichen in einem beliebigen Unicode Zeichensatz
isTitleCase	ein Titelzeichen
isUnicodeIdentifierPart	gültig nach dem ersten Zeichen eines Unicode Beeichners
isUnicodeIdentifierStart	gültig als erstes Zeichen eines Unicode Identifiers
isUpperCase	ein Grossbuchstabe
isWhitespace	ein Whitespace Zeichen (Fluchtsymbol)

Unicode Bezeichner werden im Unicode Standard bezeichnet. Unicode Identifier müssen mit einem Buchstaben beginnen (verbindende Zeichen wie _ oder Währungszeichen wie $, das Yen Zeichen oder ähnliche gelten in Unicode nicht als Buchstaben, allerdings in Java!).

UnicodeBezeichner / Identifier dürfen nur Buchstaben, verbindende Zeichen (wie _), Zahlen, numerische Buchstaben (wie die römischen Zeichen) und vernachlässigbare Steuerzeichen.

Alle diese Zeichentypen und weitere werden im Unicode Standard definiert. Mit der statischen Methode getType liefert eine ganze Zahl (einen int Wert), mit dem der Unicode Typus festgestellt werden kann. Als Rückgabe liefert die Methode eine der folgenden Konstanten:

COMBINING_SPACING_MARK	MODIFIER_SYMBOL
CONNECTOR_PUNCTATION	NON_SPACING_MARK
CONTROL	OTHER_LETTER
CURRENCY_SYMBOL	OTHER_NUMBER
DASH_PUNCTATION	OTHER_PUNCTATION
DECIMAL_DIGIT_NUMBER	OTHER_SYMBOL
ENCLOSING_MARK	PARAGRAPH_SEPERATOR
END_PUNCTATION	PRIVATE_USE
FORMAT	SPACE_SEPARATOR
LETTER_NUMBER	START_PUNCTATION
LINE_SEPARATOR	SURROGATE
LOWERCASE_LETTER	TITLECASE_LETTER
MATH_SYMBOL	UNASSIGNED
MODIFIER_LETTER	UPPERCASE_LETTER

Unicode wird aufgeteilt in Blöcke zusammenhängender Zeichen. Die statische (innere) Klasse Character.Subset kann eingesetzt werden, um Teilmengen der Unicode Zeichen zu bilden. Die statische (innere) Klasse Character.UnicodeBlock erweitert Subset, um Standard- Unicode Zeichenblöcke zu definieren, welche dann als statische Felder von UnicodeBlock zur Verfügung stehen. Die statische Methode UnicodeBlock.of( ) liefert das UnicodeBlock Objekt zu einem bestimmten Zeichen. Der UnicodeBlock definiert auch Konstanten für alle Blöcke, beispielsweise GREEK, KATAKANA, TELUGU oder COMBINING_MARKS_FOR_SYMBOLS. Die statische of( ) Methode liefert einen dieser Namen oder null, falls das Zeichen zu keinem Block gehört.

Hier ein Beispiel:

```
boolean isShape = (Character.UnicodeBlock.of(ch) ==
Character.UnicodeBlock.GEOMETRIC_SHAPES);
```

testet, ob ein Zeichen zum Unicode Block GEOMETRIC_SHAPES. Die Liste aller Blocknamen finden Sie bei der Unicode Beschreibung oder auf dem Unicode Web Site (http://www.unicode.org). Sie können auch selbst sehr einfach ein Programm schreiben, welches die Unicode Tabelle ('\u1234', wobei 1234 Zahlen sind) generiert und abfragt, zu welchem Block das Zeichen gehört.

Zwei Subset Objekte definieren den selben Block, falls sie das selbe Objekt darstellen. Die equals() und die hashcode() Methode wurden in der Klasse als final deklariert, können also nicht überschrieben werden. Damit will man vermeiden, dass man unsinnigerweise eigene Unicode Blöcke definiert, die mit fix definierten Blöcken überlappen.

11.1.4. Number

Die Number Klasse ist eine abstrakte Klasse, welche von allen Wrapper Klassen erweitert werden, welche numerische Basisdatentypen darstellen: Byte, Short, Integer, Long, Float und Double.

Die abstrakten Methoden von Number liefern den Wert des Objekts als numerische Datentypen:

```
public byte byteValue()
public short shortValue()
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

Jede erweiternde Klasse überschreibt diese Methoden, um passende Datentypen zu definieren. Beispielsweise liefert die Konversion eines Float Objekts mit dem Wert 32.87 beim Aufruf der Methode intValue einfach den Wert 32, genau wie (int)32.87.

11.1.5. Die Integer Wrappers

Die Klassen Byte, Short, Integer und Long erweitern Number, um die entsprechenden Integertypen als Klassen darzustellen. Zusätzlich zu den grundlegenden Number Methoden kennt jede der Integer Wrapper Klassen folgende Methoden:

public static *type* parse*Type*(String str, int radix)
: konvertiert die Zeichenkette str in einen numerischen Wert des spezifischen Typs (*type*) mit Hilfe des vorgegebenen Radix. Zum Beispiel liefert Integer.parseInt("1010",2) den Wert 10, während Integer.parseInt("-1010",2) den Wert -10 liefert. Falls str nicht in der Basis radix konvertiert werden kann, wird eine NumberFormatException geworfen.

public static *type* parse*Type*(String str)
: Diese Methode ist äquivalent zu parse*Type*(str,10).

public static *type* valueOfString(String str, int radix)
: Liefert ein Wrapper Objekt der Klasse *Type* mit einem Wert, der sich aus der Dekodierung von str zur Basis radix ergibt. Es gibt keinen Konstruktor dieser Form!
Die Methode kombiniert zwei Schritte:
1) parsen der Zeichenkette
2) Konstruktion eines neuen Wrapper Objekts mit diesem ermittelten Wert.
Beispiel:
Integer.valueOf("1010",2) ist äquivalent zu
new Integer(Integer.parseInt("1010",2)).
Falls str nicht dekodiert werden kann oder in der Basis radix nicht darstellbar ist, wird eine NumberFormatException geworfen.

public static String toString(*type* val)

Diese Methode liefert eine Zeichendarstellung des gegebenen Basisdatentyps bzw. Datentyps *type*.
Beispiel:
Integer.toString(66) liefert die Zeichenkette "66".
Diese Methode wird auch aufgerufen, falls Sie beispielsweise Zeichenketten in der Ausgabe verknüpfen: "i="+66.

Zusätzlich besitzen die Integer und Long Klassen die folgenden Methoden, wobei *type* entweder int oder long sein muss:

public static String toString(*type* val, int radix)

Diese Methode liefert eine Darstellung des gegebenen Werts (val) zur gegebenen Basis (radix). Falls radix fehlt, wird einfach die Basis 10 angenommen.

public static String toBinaryString(*type* val)

Diese Methode liefert eine Zeichenkettendarstellung des Zweierkomplements des gegebenen Wertes (val).
Falls der Wert positiv ist, entspricht dies der toString(value, 2).
Falls der Wert negativ ist, wird der Wert als Binärzahl, mit allen Einsen dargestellt.
Beispiel:
Integer.toBinaryString(-10) liefert "11111111111111111111111111110110".
Diese Darstellung kann in der parse*Type* Methode jedoch nicht eingesetzt werden; es würde der MAX_VALUE der Typen überschritten!

public static String toOctalString(*type* val)

Liefert eine Zeichenkettendarstellung zum gegebenen Wert in einer vorzeichenlosen Basis 8 Darstellung.
Beispiel:
Integer.toOctalString(10) liefert die Zeichenkette "12".
Negative Werte werden wie bei der toBinaryString() Methode beschrieben behandelt.

Zu beachten ist, dass Sie keine führende Zeichen "0" für oktal oder "0x" für hexadezimale Zahlen liefert. Falls Sie die Radix Information in er anschliessen Bearbeitung der Zeichenkette benötigen, liegt es an Ihnen, diese weiterzugeben.

11.1.6. Die Floating-Point Wrapper Klassen

Die Float und Double Klassen erweitern Number, um float und double Datentypen darzustellen. Mit wenigen Ausnahmen sind die Namen der Methoden und Konstanten die selben für beide Basisdatentypen. Die folgende Tabelle enthält die Informationen für den Float Datentyp. Sie können aber Float und float durch Double und double austauschen.

Zusätzlich zu den normalen Methoden der Number Klasse verfügt Float über die Methoden und Konstanten:

public final static float POSITIVE_INFINITY
: Der Wert für +unendlich.

public final static float NEGATIVE_INFINITY
: Der Wert für -unendlich.

public static float NaN
: Not-a-Number : diese Konstante kann eingesetzt werden, um einen NaN Wert zu erhalten. Sie können damit *nicht* testen, ob ein Zeichen eine Zahl darstellt!
Falls Sie diese Konstante in Vergleichen einsetzen, liefert der Vergleich immer false!

public static boolean isNaN(float val)
: Liefert true, falls val ein NaN Wert ist.

public boolean isNaN()
: Liefert true, falls der Wert des Objekts einen NaN Wert besitzt.

public boolean isInfinite()
: Liefert true, falls der Wert dieses Objekts positiv oder negativ unendlich ist.

Zusätzlich zu den üblichen Konstruktoren besitzt Float einen Konstruktor mit einer double als Argument. Dieses Argument wird in eine float Zahl umgewandelt und anschliessend zur Initialisierung verwendet.

Float besitzt auch Methoden, mit denen die Bits innerhalb der float Zahl manipuliert werden können. Damit lassen sich float in int und umgekehrt umwandeln.

Die Double Klasse stellt analoge Methoden für die Umwandlung von double in long Bit Muster zur Verfügung.

public static int floatToIntBits(float val)
: Liefert die Bit Darstellung des float Wertes als int, gemäss der IEEE-754 Floating Point "single precision" Darstellung. NaN wird immer gleich dargestellt.

public static float int floatToRawIntBits(float val)
: Äquivalent zu floatToIntBits ausser, dass das aktuelle Muster an Stelle des NaN Musters zurückgeliefert wird.

public static float intBitsToFloat(int bits)

Liefert die float Darstellung zu einer gegebenen Bit Darstellung, gemäss IEEE-754 Floating Point "single precision" Darstellung.

Wir können auch float in Zeichenketten und Zeichenketten in float umwandeln, analog zur Integer Klasse.

public static *type* parse*Type*(String str)

Konvertiert die Zeichenkette str in einen nummerischen Wert in der *type* Basistypdarstellung.
Falls str nicht darstellbar ist, wird eine NumberFormatException geworfen.
Beispiele:
Float.parseFloat("3.14") liefert den float Wert 3.14.
Wir könnten auch anders vorgehen:
valueOf(str) ist äquivalent zu Float.valueOf(str).floatValue().

public static String toString(*Type* val)

Liefert eine Zeichendarstellung zum gegebenen Basisdatentyp.
Beispiel:
Double.toString(0.3e2) liefert "30.0".
Diese Methode wird auch eingesetzt, falls Zeichenketten verknüpft werden. NaN liefert die Zeichenkette "NaN".

Selbsttestaufgabe 1

Schreiben Sie ein Programm, welches aus einer Datei Zeilen im Format "*type value"* liest. Type ist dabei einer der Datentypen (Boolean, Character, ...) und value ist eine Zeichenkette, welche als Typ darstellbar ist.
Lesen Sie alle Zeilen, konvertieren Sie die Zeichenkette in den Datentyp und speichern Sie diese in einer ArrayListe.
Geben Sie diese Liste nach der Umwandlung aller Zeilen aus.

11.2. Reflection

Das Package java.lang.reflect enthält das *Reflection* Package. Mit diesen Klassen kann man die Datentypen (Klassen, Objekte) im Detail überprüfen. Sie können mit Hilfe dieser Klassen einen vollständigen Datentyp Browser entwickeln. Aber Sie können auch eine Applikation schreiben, welche andere Klassen analysiert und interpretiert, also Objekte kreieren und Methoden ausführen. Die folgenden Klassen und Methoden stammen alle aus dem Package Reflection, mit Ausnahme der Klassen Class und Package, welche zum Package java.lang gehören.

Reflection startet mit der Class Klasse oder einem Class Objekt. Aus dem Class Objekt können Sie eine vollständige Liste der Members der Klasse erhalten, also alle Datenfelder und Methoden und deren Parameter, Parametertypen und Rückgabewertetypen. Aber Sie erhalten auch die Information darüber, welche Klasse die Oberklasse ist oder welche Interfaces implementiert werden, zudem alle Modifiers (public, static, abstract, final zum Beispiel). An Stelle von Reflection spricht man auch von Introspection. Beide Ausdrücke (Reflection und Introspection) wollen veranschaulichen, dass die Klasse, der Datentyp, die Fähigkeit hat, sich selbst anzuschauen und etwas über sich selbst zu berichten. Diese Möglichkeit können Sie beispielsweise einsetzen, um in Browsern Informationen über die Klassen herauszufinden und eventuell grafisch darzustellen. Zudem stellen die Informationen einen ersten Schritt in Richtung dynamischer Generierung von Objekten dar.

Reflection gestattet es Ihnen Programme zu schreiben, welche Sie leichter direkt ausführen können. Falls Sie den Namen der Klasse kennen, können Sie mit einem Class Objekt eine neue Instanz dieser Klasse bilden. Mit diesen Objekten können Sie dann genauso kommunizieren, wie mit traditionell erzeugten Objekten und deren Methoden. Methodenaufrufe sind allerdings wesentlich komplexer, als bei Objekten, die Sie mit new kreiert haben. Daher sollten Sie Reflection nur einsetzen, falls die klassischen Methoden nicht zur Verfügung stehen. Dumm wäre es, beispielsweise Methodenobjekte als Methodenpointer im Sinne von C/C++ einzusetzen. Den selben Effekt erreichen Sie mit Interfaces und abstrakten Klassen einfacher und eleganter. Falls Sie aber beispielsweise fremden Code visualisieren müssen, bleibt Ihnen nicht viel anderes als Reflection übrig.

11.2.1. Die Klasse Class

Für jeden Datentyp, für jede Klasse, existiert ein Class Objekt (analog zu den Metadaten in einer Datenbank: diese beschreiben die Tabellen, Datenelemente und deren Datentyp). Dies gilt für alle Klassen, Basisdatentypen, Interfaces und Arrays. Zudem gibt es ein spezielles Class Objekt für das Schlüsselwort void. Mit Hilfe dieser Objekte kann man Informationen über den Typus erhalten und im Falle von referenzierbaren Klassen, auch neue Objekte dieses Typs kreieren.

Die Class Klasse ist der Ausgangspunkt für Reflection. Die Klasse stellt auch Werkzeuge für die Manipulation der Klassen zur Verfügung, primär, um neue Objekte dieses Typs zu kreieren oder zum Laden der Klassen, beispielsweise über das Netzwerk.

Sie können auf vier Arten zu einem Class Objekt gelangen:
1. Abfrage des Objekts (erfragen des Class Objekts) mit Hilfe der getClass() Methode
2. mittels eines Klassenliterals (der Name der Klasse gefolgt von ".class")
3. nachschlagen des Objekts mittels des voll qualifizierten Namens (mit allen Packages) und der Class.forName() Methode
4. mittels einer der vom Reflection Package zur Verfügung gestellten Methoden, als Rückgabewert und anschliessender Abfrage aller Klassen in der Hierarchie mittels Class.getClasses().

Die Klasse Class stellt Ihnen einige Methoden zur Verfügung, mit deren Hilfe Informationen über eine bestimmte Klasse ermittelt werden können. Einige davon liefern Informationen über den Datentyp der Klasse - welche Interfaces implementiert werden, welche Klassen erweitert werden - oder liefern Informationen über die Klassenmembers, inklusive der inneren Klassen und Interfaces. Sie können abfragen, ob eine Klasse einen Datentyp, ein Interface oder ein Array ist, oder ob ein Objekt eine Instanz einer bestimmten Klasse ist. Wir werden diese Methoden im Detail anschauen und in Programmen einüben.

Die grundlegenden Methoden der Klasse Class sind jene, welche Ihnen gestatten die Typenhierarchien zu durchlaufen, Informationen über Interfaces, welche implementiert werden, auszugeben, oder Klassen anzugeben, welche erweitert werden. Hier ein einfaches Beispiel:

```
package reflection;

public class TypenBeschreibung {

  public static void main(String[] args) {
    TypenBeschreibung beschreibung = new TypenBeschreibung();
    System.out.println("" +args[0]);
    for (int i=0; i<args.length; i++) {
      try {
        Class startClass = Class.forName(args[i]);
        beschreibung.printType(startClass, 0, basic);
      } catch(ClassNotFoundException e) {
        System.err.println(e);
      }
    }
  }
  // Standardausgabe
  private java.io.PrintStream out = System.out;

  // Hilfsarrays für die Ausgabe der DAtentypen
  private static String[ ]
    basic       = {"class",   "Interface"},
    superClass  = {"extends", "implements"},
    iFace       = {null,      "extends"};

  private void printType(Class type, int depth, String[] labels) {
    if (type == null) // abbrechen der Rekusrion : keine Supertypen
      return;

    // Ausgabe des Datentyps
    for (int i=0; i< depth; i++)
```

```
      out.print(" ");

    out.print(labels[type.isInterface() ? 1 : 0] + " ");
    out.println(type.getName() );

    // Ausgabe aller Interfaces, die diese Klasse implementiert
    Class[ ] interfaces = type.getInterfaces();

    for (int i=0; i<interfaces.length; i++)
      printType(interfaces[i],depth+1,type.isInterface()?iFace:
superClass);


    // Rekursion
    printType(type.getSuperclass(), depth + 1, superClass);
  }
}
```

Dieses Programm iteriert durch alle voll qualifizierten Klassennamen, die Sie auf der Kommandozeile eingeben. Mit diesem Namen wird ein Class Objekt erstellt und anschliessend die gesamte Informationen über die Klassenhierarchie ausgegeben. Die Ausgabe geschieht mit Hilfe der printType Methode. Der try ... catch Block ist nötig, weil eventuell keine Klasse zum angegebenen Namen existiert.

Testen Sie das Programm mit unterschiedlichen Klassennamen. Ein Beispiel:
falls Sie als Parameter die Klasse java.util.HashMap eingeben, erhalten Sie als Ausgabe:

```
java.util.HashMap
class java.util.HashMap
 implements java.util.Map
 implements java.lang.Cloneable
 implements java.io.Serializable
 extends java.util.AbstractMap
  implements java.util.Map
  extends java.lang.Object
```

Nach der main Methode steht die Deklaration des Ausgabestromes, standardmässig System.out. Die String Arrays haben eine aus dem Text erkennbare Bedeutung.

Die Methode printType() druckt die Beschreibung der Parameter und ruft sich dann rekursiv auf, um die Supertypen auszugeben. Der depth Parameter hält fest, wie weit wir die Hierarchie schon hinaufgeklettert sind. Dabei verschieben wir die Ausgabe jeweils um ein Zeichen. Die Variable wird auf jedem Hierarchielevel um Eins erhöht. Das Array labels[ ] beschreibt die Klassen - label[0] ist das Label für die Klasse; labels[1] sind Interfaces.

Drei Arrays werden definiert. basic enthält die Top Level Konstrukte ("class", "interface"); superClass wird immer dann eingesetzt, wenn es Oberklassen gibt ("extends", "implements"); iFace wird für übergeordnete Konstrukte eingesetzt (Interfaces, die Interfaces erweitern oder die Spitze). Mit getName() wird der Name des Datentyps ausgedruckt. Die Formattierung geschieht mit der etwas unüblichen Kontrollstruktur <a> ? <b> : <c>: falls <a>=true ist wird <b> ausgeführt, sonst <c>.

Die Methode printType ruft sich selber auf, bis schliesslich die Wurzel, Object erreicht wird.

Das Reflection Package kennt einige Methoden, mit deren Hilfe Klassenobjekte abgefragt werden können:

public boolean isInterface()
liefert true, falls diese Klasse ein Interface ist.

public boolean isArray()
liefert true, falls diese Klasse ein Array ist.

public boolean isPrimitive()
liefert true, falls diese Klasse eine der acht Basistypen darstellt ('primitive datatype').

Mit Hilfe anderer Methoden können Sie detailliertere Informationen über die Klasse und den Datentyp herausfinden:

public Class[ ] getInterfaces()
liefert ein Array von Class Objekten, eines für jedes Interface, das vom abgefragten Datentyp implementiert wird. Falls der Datentyp kein Interface implementiert, wird einfach ein Array der Länge null zurück gegeben.

public Class getSuperClass()
liefert die Superklasse des betrachteten Datentyps. Falls es sich um Object handelt wird null zurück geliefert, da es keine Superklasse / Oberklasse gibt.
Falls der Datentyp ein Array ist, wird das Class Objekt für Object zurückgeliefert.
Durch rekursiven Aufruf dieser Methode können Sie alle Superklassen bestimmen, schrittweise.

public int getModifiers()
liefert die Modifiers des Datentyps, allerdings in Form von Integer Werten. Diese Werte kann man mit Hilfe der Klasse Modifier entschlüsseln. Die Modifiers sind auf der einen Seite Zugriffsmodifier (public, protected, private) oder abstract, final und static. (Aus Gründen der Einfachheit wird auch die Tatsache, ob ein Datentyp ein Interface ist oder nicht, auf diese Art und Weise verschlüsselt). Die Basisdatentypen sind immer public und final; Arrays sind immer final und besitzen die selben Zugriffsmodifier wie die Datentypen, die sie enthalten.

public Class getComponentType()
liefert das Class Objekt, welches der Komponente des Arrays entspricht, also den Elementen in einem Array.
Beispiel:
falls ein Array int Zahlen enthält, liefert die Methode getClass ein Class Objekt, für das isArray true ist und bei dem die getComponent() Methode ein Objekt int.class zurück liefert.

public Class getDeclaringClass()
liefert das Class Objekt für den Datentyp, in dem dieser Typ deklariert wurde (die Klasse muss also eine innere Klasse sein). Falls es sich nicht um eine innere Klasse handelt, wird null zurück geliefert.

Selbsttestaufgabe 2

Modifizieren Sie das TypenBeschreibung's Beispiel oben so, dass die Informationen über Object weggelassen werden. Das macht Sinn, weil jede Klasse Object erweitert. Verwenden Sie das Class Objekt für die Object Klasse als Hilfsmittel.

11.2.2. Namensgebung für Klassen

Die Class Objekte im obigen Beispiel wurden mit der statischen Methode Class.forName bestimmt. Diese Methode benutzt eine Zeichenkette als Parameter und liefert ein Class Objekt, falls die Zeichenkette eine gültige Klasse beschreibt. Der Name (die Zeichenkette) muss dabei voll qualifiziert sein, beispielsweise java.util.HashMap oder java.lang.Object. Auf der andern Seite erhalten Sie den Namen der Klasse durch Anwendung der Methode getName() auf das Class Objekt, beispielsweise liefert Object.class.getName() "java.lang.Object".

Bei Arrays verwendet man eine spezielle Notation. Diese besteht aus einem Code, mit dem der Komponententyp des Arrays gekennzeichnet wird, vorangestellt das Zeichen [.

Die Kodierung der Komponententypen sieht folgendermassen aus:

B	byte
C	char
D	double
F	float
I	int
J	long
L*classname;*	class oder interface
S	short
Z	boolean

Beispiel:
ein Array mit ints wird als [I bezeichnet,
ein Array mit Objects als [java.lang.Object; (inklusive dem Semikolon am Schluss).

Bei mehrdimensionalen Datenfeldern handelt es sich um Arrays, deren Elemente Arrays sind.
Beispiel:
int [ ] [ ] wird als [[I bezeichnet, also als Array ([ ), dessen Komponenten [I (Array von Integern) sind.

Bei inneren oder verschachtelten Klassen wird auch eine spezielle Notation verwendet. Innerhalb der Programme selbst kann man auf innere Klassen mittels der "dot" Notation zugreifen, also Outer.Inner. Der externe Klassenname ist allerdings Outer$Inner, das Dollarzeichen wird als Trennzeichen verwendet.

Eine Ausnahme bilden lokale oder anonyme Klassen: diese können mit Reflection (noch) nicht instanziert werden!

Im Falle der Basisdatentypen kann man das Class Objekt nicht mittels forName() erhalten - Sie müssen mit den Klassenliteralen arbeiten, beispielsweise int.class oder mit dem TYPE

Feld der entsprechenden Wrapperklasse: Integer.TYPE. Falls Sie sich nicht daran halten und beispielsweise die Methode forName() auf einen Basisdatentyp anwenden, wird angenommen, dass es sich um einen Datentyp des Benutzers handelt. Vermutlich wird also nicht gefunden und auch nicht zurück gegeben.

Bisher haben wir lediglich die einfachste Form der forName() Methode verwendet. Diese sieht in ihrer komplexeren Form folgendermassen aus:

public Class forName(String name, boolean initialize, ClassLoader loader) throws ClassNotFoundException

> die Methode liefert ein Class Objekt zum gegebenen Namen und lädt diese Klasse mit Hilfe des angegebenen Klassenladers. Die Methode versucht bei gegebenem voll qualifizierten Namen der Klasse diese zu finden und zu laden, mit Hilfe des angegebenen Class Loaders. Falls der Class Loader null ist, wird der System Class Loader verwendet. Die Klasse wird nur initialisiert, falls die Variable initialize auf true steht und die Klasse bisher noch nicht initialisiert wurde.

Aus dieser Methodenbeschreibung geht hervor, dass zum Bestimmen des Class Objekts die Klasse geladen und gegebenenfalls gelinked und initialisiert werden muss - also insgesamt ein recht komplexes Unterfangen ist. Die einfache Form: Class.forName() verwendet einfach den System Class Loader, also this.getClass().getClassLoader(), und initialisiert die Klasse falls nötig. Falls die Klasse nicht gefunden werden kann, wird eine ClassNotFoundException geworfen. Unter Umständen enthält dieses (Exception) Objekt Informationen über verschachtelte Aufrufe. Sie können weitere Details über das Exception Objekt erhalten, indem Sie die Methode getException() aus das Exception Objekt anwenden. Da der gesamte Prozess recht komplex ist, könnte auch ein LinkageError oder ein ExceptionInitializerError geworfen werden.

11.2.3. Class Members untersuchen

Die Klasse Class enthält Methoden, mit deren Hilfe die Komponenten des speziellen Typs überprüft werden können. Sie erhalten Informationen über Datenfelder, Methoden, innere Klassen und Konstruktoren. Diese Kenngrössen einer Klasse werden in Form von Klassen dargestellt: Field Objekte für Datenfelder, Method Objekte für die Methoden, Constructor Objekte für Konstruktoren und Class Objekte für innere Klassen.

Diese Methoden zur Bestimmung der Kenngrössen einer Klasse kommen in unterschiedlichen Formen. Dies hängt damit zusammen, was Sie konkret über die Klasse erfragen möchten. Sie können beispielsweise sich lediglich für public Members interessieren oder für die Members der Oberklasse.

Alle public Members erhalten Sie mit folgenden Methoden:

```
public Constructor[ ] getConstructors()
public Field[ ] getFields()
public Method[ ] getMethods()
public Class[ ] getClasses()
```

Die Methode getConstructors() liefert lediglich die public Konstruktoren der betrachteten Klasse. Die Konstruktoren einer Oberklasse sind nicht erkennbar.

Sie können auch lediglich die Members abfragen, die in der betrachteten Klasse deklariert wurden, also keine vererbten. Diese Members müssen auch nicht unbedingt public sein.

```
public Constructor[ ] getDeclaredConstructors()
public Field[ ] getDeclaredFields()
public Method[ ] getDeclaredMethods()
public Class[ ] getDeclaredClasses()
```

In vielen Fällen werden die Arrays leer sein, weil die Klasse die entsprechenden Members nicht besitzt.

Die getClasses() Methode liefert sowohl innere Klassen als auch verschachtelte Interfaces.

Falls Sie ein spezielles Member verlangen, benötigen Sie Zusatzinformationen. Für innere Typen ist dies einfach der Name des inneren Typs. Dann liefert Class.forName() die gewünschte Information.

Falls Sie spezielle Felder abfragen möchten, sollten Sie die folgenden Methoden einsetzen:

```
public Field getField(String name)
public Field getDeclaredField(String name)
```

Mit der ersten Methode finden Sie ein public Feld, welches entweder lokal deklariert wurde oder aber geerbt wird. Die zweite Methode findet die Felder der Klasse, sofern diese in der Klassendeklaration enthalten sind, auch wenn diese nicht public sind. Falls das entsprechende Datenfeld nicht vorhanden ist, wird eine NoSuchFieldException geworfen. Die Länge des Arrays wird übrigens nicht geliefert, die length Information über das Rückgabe-Array fehlt also.

Sie können auch Methoden mittels einer bestimmten Signatur abfragen, also dem Namen der Methode und einem Array von Class Objekten, welche die Anzahl und die Datentypen der Parameter angeben.

```
public Method getMethod(String name, Class[ ] parameterTypes)
public Method getDeclaredMethod(String name, Class[ ] parameterTypes)
```

Konstruktoren werden genauso mittels einer Parameterliste identifiziert:

```
public Constructor getConstructor(Class[ ] parameterTypes)
public Constructor getDeclaredConstructor(Class[ ] parameterTypes)
```

In beiden Fällen wird, falls die Methode nicht existiert, eine NoSuchMethodException geworfen.

Alle obigen Methoden benötigen eine Sicherheitsüberprüfung bevor sie ausgeführt werden dürfen. Daher wird der Security Manager aufgerufen. Falls kein Security Managerinstalliert ist, sind alle Methodenaufrufe erlaubt. Ein Security Manager wird typischerweise den Zugriff auf public Informationen gestatten. DerZugriff auf nicht public Informationen werden aber in der Regel durch einen Security Manager abgeblockt oder eine SecurityException geworfen.

Das folgende Programm listet die öffentlichen Felder, Methoden und Konstruktoren einer vorgegebenen Klasse.

```
package reflection;

/**
 * Title:        Beispiele zum Reflection Package
 * Description:  Beispiele, mit deren Hilfe die Struktur einer Klasse,
einer Datentyps festgestellt werden kann.
 * Copyright:    Copyright (c) 2000
 * Company:      Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

 import java.lang.reflect.*;

public class KlassenBeschreibung {
  public static void main(String[] args) {
    try {
      Class c = Class.forName(args[0]);
      System.out.println(c);
      printMembers(c.getFields());
      printMembers(c.getConstructors());
      printMembers(c.getMethods());
    } catch(ClassNotFoundException e) {
      System.err.println("Unbekannte Klasse - "+args[0]);
    }
  }

  private static void printMembers(Member[ ] mems) {
    for (int i=0; i<mems.length; i++) {
      if (mems[i].getDeclaringClass() == Object.class)
        continue;
      String decl = mems[i].toString();
      System.out.print("      ");
      System.out.println(decl);
    }
  }
}
```

Wenn wir dieses Programm auf die generierte Klasse anwenden, erhalten wir folgende Ausgabe:

```
class reflection.KlassenBeschreibung
      public reflection.KlassenBeschreibung()
      public static void
reflection.KlassenBeschreibung.main(java.lang.String[])
```

Im Falle der HashMap Klasse :

```
class java.util.HashMap
      public java.util.HashMap(int)
      public java.util.HashMap()
      public java.util.HashMap(java.util.Map)
      public java.util.HashMap(int,float)
      public int java.util.AbstractMap.hashCode()
      public boolean java.util.AbstractMap.equals(java.lang.Object)
      public java.lang.String java.util.AbstractMap.toString()
```

```
      public java.lang.Object
java.util.HashMap.put(java.lang.Object,java.lang.Object)
      public java.lang.Object java.util.HashMap.clone()
      public java.lang.Object java.util.HashMap.get(java.lang.Object)
      public java.util.Collection java.util.HashMap.values()
      public int java.util.HashMap.size()
      public void java.util.HashMap.clear()
      public java.lang.Object java.util.HashMap.remove(java.lang.Object)
      public java.util.Set java.util.HashMap.keySet()
      public java.util.Set java.util.HashMap.entrySet()
      public boolean java.util.HashMap.isEmpty()
      public boolean java.util.HashMap.containsValue(java.lang.Object)
      public boolean java.util.HashMap.containsKey(java.lang.Object)
      public void java.util.HashMap.putAll(java.util.Map)
```

Die Klassen Field, Constructor und Method implementieren alle das Interface Member. Das Member Interface deklariert folgende Methoden:

Class getDeclaringClass()
: liefert das Class Objekt für die Klasse in der das Member deklariert ist.

String getName()
: liefert den Namen dieses Members

int getModifiers()
: liefert den Modifier als Integer Zahl verschlüsselt. Diese Zahl kann mittels der in der Klasse Modifier entschlüsselt werden.

Obschon eine Klasse oder ein Interface auch Member einer anderen Klasse sein kann, wurde aus historischen Gründen die Klasse Class so gebaut, dass sie Member nicht implementiert. Allerdings wurden in der Klasse Class Methoden mit dem selben Namen und der selben Funktion wie in Member definiert.

Die toString() Methode der Member Klasse enthält die volle Deklaration für die Members, also auch Modofiers und im Falle der Methoden und Konstruktoren, Sinaturinformationen, nicht einfach den Namen wie die getName() Methode.

Selbsttestaufgabe 3

Schreiben Sie ein Programm, welches alle Deklarationen einer Klasse liefert, aber ohne import Anweisung, Kommentare und dem Code für die Initialisierung, die Konstruktoren und Methoden.

11.2.4. Die Modifier Klasse

Die Modifier Klasse beschreibt alle Modifier von Typen und Members mittels Konstanten: ABSTRACT, FINAL, INTERFACE, NATIVE, PRIVATE, PROTECTED, PUBLIC, STATIC, STRICT, SYNCHRONIZED, TRANSIENT und VOLATILE.

Für jede dieser Konstanten existiert eine entsprechende Abfragemethode is*Mod*(int modifiers), welche true liefert, falls der Modifier *mod* vorkommt.

Schauen wir uns ein Beispiel an. Sei ein Beispiel Feld implementiert:

```
public static final int OAK = 0;
```

Dann erhalten wir als Rückgabe der getModifiers() Methode des entsprechenden Field Objekts:

```
Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL
```

Die Konstante STRICT entspricht dem strictfp Modifier. Dieser Modifier wird eingesetzt, um eine strikte Floating Point Berechnung zu erzwingen.

Falls Sie den Methodenaufruf verschachteln können Sie eine & Verknüpfung simulieren:

```
Modifier.isPrivate(field.getModifiers() )
```

ist äquivalent zu :

```
(field.getModifiers() & Modifier.PRIVATE) != 0
```

11.2.5. Die Field Klasse

Die Field Klasse definiert Methoden für die Abfrage des Datentyps eines Feldes und für das Setzen und Abfragen von Werten eines Datenfeldes. Zusammen mit den geerbten Member Methoden können Sie damit alles über die Datenfeld-Deklaration herausfinden und auch die Datenfelder einer spezifischen Klasse oder Instanz manipulieren.

Die getType() Methode liefert das Class Objekt für den Typ des Feldes.
Beispiel:
falls das Feld eine Zeichenkette (String) ist, werden wir String.class erhalten;
falls das Feld ein Basisdatentyp ist, wie beispielsweise long, wird long.class zurück geliefert.

Mit den get... und set... Methoden können Sie auch den Wert eines Feldes setzen oder bestimmen. Diese Methoden besitzen eine einfache Form, welche Object als Argument akzeptieren und ein Object liefern. Zudem gibt es spezifischere Methoden, welche direkt mit den Basisdatentypen umgehen können. Falls die Methode statisch ist, können Sie auch null als Objekt eingeben.

Schauen wir uns einige Beispiele an:

```
public static void printShortField(Object o, String name) throws
NoSuchFieldException, IllegalAccessException {
        Field field = o.getClass().getField(name);
        Short value = (Short)field.get(o);
        System.out.println(value);
}
```

Der Rückgabewert der get...() Methode ist ein Objekt des Typs, auf den das Feld referenziert. Falls das Feld ein Basisdatentyp ist, wird ein Wrapper Objekt des entsprechenden Datentyps zurück geliefert, also im Falle eines short Feldes ein Short Objekt mit dem entsprechenden Wert.

Die set...() Methode kann analog eingesetzt werden. Schauen wir uns ein konkretes Beispiel an, in dem wir den Wert eines short Feldes setzen:

```
public static void setShortField(Object o, String name, short nv) throws
NoSuchFieldException, IllegalAccessException {
        Field field = o.getClass().getField(name);
        field.set(o,new Short(nv) );
}
```

Wir müssen also einen Short Wrapper benutzen, um den Wert nv aufzunehmen, weil die get...() und set...() Methoden lediglich auf Objekten funktionieren.

Falls auf ein Feld eines spezifizierten Objekts nicht zugegriffen werden kann und eventuell ein Zugriffsschutz definiert wurde, wird eine IllegalAccessException geworfen.

Falls das Objekt, welches als Argument übergeben wird, von einem unterschiedlichen Typ ist, als das zugrundeliegende Feld, wird eine IllegalArgumentException geworfen.

Falls Sie auf ein statisches Feld zugreifen wollen, kann es sein, dass das Feld zuerst initialisiert werden muss. Falls dies nicht geschieht, wird eine ExceptionInitializerError geworfen.

Die Field Klasse besitzt auch spezifische Methoden, um Basisdatentypen zu bestimmen. Diese sind von der Form get*Basisdatentyp*() und set*Basisdatentyp*() und werden auf Field Objekte angewandt.
Beispiel:

```
short value = field.getShort(o);
```

Das Setzen der Variable kann auch wie in folgendem Beispiel geschehen:

```
field.setShort(o, nv);
```

Mit einigem Aufwand können Sie alle Datenfelder manipulieren. Aber das sollten Sie unterlassen! Die Programmiersprache fängt viele Unzulänglichkeiten der Programmierung bei der Übersetzung ab. Aber es liegt an Ihnen, ob Sie diese Checks mit einigen Tricks über die Reflection Klassen und Methoden umgehen wollen.

Selbsttestaufgabe 4

Kreieren Sie ein Interpret Programm, welches ein Objekt einer bestimmten Klasse generiert und es dem Benutzer gestattet das Objekt zu untersuchen und die Datenfelder des Objekts zu modifizieren.

11.2.6. Die Method Klasse

Die Method Klasse zusammen mit der Member Klasse gestattet es Ihnen vollständige Informationen über die Deklaration einer Methode zu gewinnen, und Sie gestatten es Ihnen auch, diese Methoden auszuführen, falls Sie ein entsprechendes Objekt zur Verfügung haben.

public Class getReturnType()
liefert das Class Objekt für den Datentyp dieser Methode. Falls die Methode als void deklariert wurde, liefert die Methode ein Objekt vom void.class Typ.

public Class[ ] getParameterTypes()
liefert ein Datenfeld mit Class Objekten, jeweils eines pro Parameter, in der selben Reihenfolge wie in der Methodendeklaration.
Falls die Methode keinen Parameter besitzt, wird ein leeres Array zurück geliefert.

public Class[ ] getExceptionTypes()
liefert ein Datenfeld mit Class Objekten, je eines pro deklarierte Exception in der throws Erweiterung der Klassendefinition. Falls keine Exceptions angegeben wurden, wird ein leeres Array geliefert.

public Object invoke(Object onThis, Object[ ] args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException
damit wird die Methode aufgerufen, welche im Method Objekt definiert ist, im onThis Objekt. Die Parameter werden aus dem Array args bestimmt.
Falls die Methode zu einer statischen Klasse gehört, kann an Stelle von onThis auch null angegeben werden.
Die Länge des args Datenfeldes muss mit der Anzahl Parameter der Methode übereinstimmen, sonst wird eine IllegalArgumentException geworfen.
Falls Sie keine Zugriffsrechte auf diese Methode haben, und trotzdem die Methode aufrufen wollen, wird die IllegalAccessException geworfen.
Falls onThis nicht ein Objekt der Klasse ist, zu der die Methode gehört, wird eine IllegalArgumentException geworfen.
Falls onThis null ist, wird eine NullPointerException geworfen.
Falls die Methode statisch ist, kann es sein, dass eine Instanz kreiert werden muss.
Falls dies nicht geschieht, wird ein ExceptionInitializerError geworfen.
Falls die Methode abgebrochen wird, wird eine InvocationTargetException geworfen.

Falls Sie die invoke Methode mit Basisdatentypen aufrufen, müssen diese als Objekte vorliegen, also als Wrapper Objekte mit den entsprechenden Werten. Auch die Rückgabewerte des Methodenaufrufes werden in Wrapper Klassen geliefert. Falls die Methode void liefert, wird null zurück gegeben.

Schauen wir uns ein Beispiel an: der Methodenaufruf

```
return strindexOf(".", 8);
```

kann auch viel komplexer auf folgende Art und Weise beschrieben werden:

```
Throwable failure;
try {
        Class StrClass = str.getClass();
        Method indexM = StrClass.getMethod("indexOf", new Class[ ] {
                String.class, int.class } );
        Object result = indexM.invoke(str, new Object[ ] {
                ".", new Integer(8) } );
        return ((Integer)result).intValue();
} catch (NoSuchMethodException e) {
        failure = e;
} catch (InvocationTargetException e) {
        failure = e.getTargetException() }
} catch (IllegalAccessException e) {
        failure = e;
}
throw failure;
```

Der Reflection basierte Programmcode umfasst auch alle Sicherheitschecks, auch wenn diese Tests erst zur Laufzeit überprüft werden können.

Sicher ist es Ihnen klar, dass Sie gar nicht erst versuchen sollten, solche komplexe Umschreibungen durchzuführen. Falls Sie ein Design Tool oder einen Debugger schreiben wollen, dann ist Reflection eine tolle Sache; sonst sollten Sie sich davor eher hüten!

Selbsttestaufgabe 5

Modifizieren Sie Ihr Interpreter Programm so, dass es auch Methoden aufrufen kann. Sie sollten auch die Ausnahmen, die geworfen werden können, korrekt wiedergeben.

11.2.7. Kreieren eines Objekts und die Constructor Klasse

Mit Hilfe der newInstance Methode des Class Objekts können Sie auch eine neue Instanz (Objekt) der Klasse generieren. Diese Methode führt den argumentlosen Konstruktor der Klasse aus und liefert eine Referenz auf ein neues Objekt. Diese Referenz wird als Object geliefert, muss also noch entsprechend dem erwarteten Datentyp gecastet werden.

Das Kreieren eines Objekts auf diese Art und Weise ist dann sinnvoll, wenn Sie generellen Programmcode schreiben wollen und die Klasse auf einfache Art und Weise durch den Benutzer angegeben werden kann.

Angenommen wir haben einen Sort Algorithmus implementiert und möchten unterschiedliche Implementationen testen. Dann erlauben wir die Angabe der Implementationsklasse als Argument, also als Parameter eines allgemeineren Programms.

Hier zuerst die spezifische Implementation:

```
package reflection;

abstract class SortDouble {
  private double[ ] values;
  private final SortMetrics curMetrics = new SortMetrics();

  // Sort Aufruf
  public final SortMetrics sort(double[ ] data) {
    values = data;
    curMetrics.init();
    doSort();
    return getMetrics();
  }

  public final SortMetrics getMetrics() {
    return (SortMetrics)curMetrics.clone();
  }

  //
  protected final int getDataLength() {
    return values.length;
  }

  //
  protected final double probe(int i) {
    curMetrics.probeCnt++;
    return values[i];
  }

  //
  protected final int compare(int i, int j) {
    curMetrics.compareCnt++;
    double d1 = values[i];
    double d2 = values[j];
    if (d1 == d2)
      return 0;
    else
      return (d1 < d2 ? -1 : 1);
  }
```

```
  //
  protected final void swap(int i, int j) {
    curMetrics.swapCnt++;
    double tmp = values[i];
    values[i] = values[j];
    values[j] = tmp;
  }

  //
  protected abstract void doSort();
}

package reflection;

final class SortMetrics implements Cloneable {
  public long probeCnt,   // Probedaten
              compareCnt, // vergleichen zweier Elemente
              swapCnt;    // swappen zweier Elemente

  public void init() {
    probeCnt = swapCnt = compareCnt = 0;
  }

  public String toString() {
    return  probeCnt + " probes " +
            compareCnt + " compares " +
            swapCnt + " swaps ";
  }

  //
  public Object clone() {
    try {
      return super.clone(); // Standard
    } catch(CloneNotSupportedException e) {
        // kann eigentlich nicht passieren: Object und this sind klonbar
        throw new InternalError(e.toString());
    }
  }
}

package reflection;


public class SimpleSortDouble extends SortDouble {
  protected void doSort() {
    for (int i=0; i < getDataLength(); i++) {
      for (int j=i+1; j<getDataLength(); j++) {
        if (compare(i,j) > 0)
          swap(i,j);
      }
    }
  }
}
```

```
package reflection;

// klassischer Tester

public class TestSort {
  static double[ ] testData = { 0.3, 1.3e-2, 7.9, 3.17, };

  public static void main(String[] args) {
    SortDouble bsort = new SimpleSortDouble();
    SortMetrics metrics = bsort.sort(testData);
    System.out.println("Metrics : "+metrics);
    for (int i=0; i<testData.length; i++)
      System.out.println("\t"+testData[i]);
  }
}
```

Und hier endlich die Reflection Version:

```
package reflection;

public class ReflectionTestSorter {
  static double[ ] testData = { 0.3, 1.3e-2, 7.9, 3.17, };

  public static void main(String[] args) {
    try {
      // jetzt ohne Datentypen mit einer Sortbeschreibung als args
      // Beispiel : reflection.SimpleDoubleSort
      for (int arg=0; arg < args.length; arg++) {
        String name = args[arg];
        Class classFor = Class.forName(name);
        SortDouble sorter = (SortDouble)classFor.newInstance();
        SortMetrics metrics = sorter.sort(testData);
        System.out.println(name+": "+metrics);
        for (int i=0; i<testData.length; i++)
          System.out.println("\t"+testData[i]);
      }
    } catch(Exception e) {
      System.err.print(e);
    }
  }
}
```

Die zwei Programme sehen sehr ähnlich aus. Aber im neuen Programm wurden alle konkreten Datentypen eliminiert (fast). Sie können einfach die implementierte Sort Klasse als Argument eingeben, voll qualifiziert!

Die Methode newInstance() kann unterschiedliche Exceptions werfen, falls die Methode falsch aufgerufen wird.
Falls die Klasse in Wahrheit ein Interface oder eine abstrakte Klasse ist, wird das Objekt nicht kreiert werden können, also eine InstantiatonException geworfen.
Falls Sie kein Zugriffsrecht auf den argumentlosen Konstruktor haben, wird eine IllegalAccessException geworfen.
Falls Ihre Security Policy das Kreieren neuer Objekte nicht gestattet, wird eine SecurityException geworfen.
Falls die Klasse bereits initialisiert sein muss, bevor Reflection Methoden adaruf angewandt werden, wird ein ExceptionInitializerError geworfen.

Die newInstance Methode kann lediglich den argumentlosen Konstruktor aufrufen. Falls Sie einen anderen Konstruktor aufrufen wollen, müssen Sie mit dem entsprechenden Class Objekt das relevante Constructor Objekt bestimmen und dann auf diesen Konstruktor die newInstance() Methode anwenden, mit den passenden Parametern.

Die Constructor Klasse zusammen mit den von der Member Klasse geerbten Methoden, gestattet es Ihnen, vollständige Informationen über die Deklaration eines Konstruktors zu erhalten und damit auch den Konstruktor zur Konstruktion neuer Objekte einzusetzen.

public Class[ ] getParameterTypes()
liefert ein Datenfeld von Class Objekten für jeden Parameter des Konstruktors, in der Reihenfolge, in der sie im Konstruktor definiert sind.
Falls der Konstruktor keine Argumente besitzt, wird ein leeres Datenfeld zurück geliefert.

public Class[ ] getExceptionTypes()
liefert ein Datenfeld von Class Objekten, je eines für jede definierte Ausnahme, die in der Definition des Konstruktors aufgeführt wurde und genau in der selben Reihenfolge.
Falls keine Ausnahmen geworfen werden können, wird ein leeres Datenfeld zurück geliefert.

Falls Sie eine neue Instanz einer Klasse benötigen, rufen Sie die newInstance() Methode des Konstruktor Objekts auf:

public Object newInstance(Object[ ] args)
throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException

mit dieser Methode wird eine neue Instanz kreiert, ein Objekt der Klasse, welche durch diesen Konstruktor konstruiert werden kann.
Es wird eine Referenz auf ein neues Objekt der entsprechenden Klasse zurück geliefert.
Die Länge des Argumente-Datenfeldes args muss mit der Anzahl Parameter für den entsprechenden Konstruktor übereinstimmen, sonst wird eine Exception geworfen.

Das Konstrukt Constructor.newInstance(...) ist ähnlich wie Method.invoke(...) . Auch hier müssen Basisdatentypen gewrappt werden, falls sie als Parameter verwendet werden sollen.

Innere Klassen sind speziell. Class.newInstance() kann nicht eingesetzt werden, um Objekte innerer Klassen zu kreieren.

Selbsttestaufgabe 6

Modifizieren Sie Ihr Interpreter Programm so, dass es Ihnen auch den Aufruf der Konstruktoren gestattet. Zeigen Sie zuerst den Aufbau der Konstruktoren inklusive Exceptions an. Falls ein Konstruktor erfolgreich benutzt wurde, dann wenden Sie das generierte Objekt an, um eine Methode aufzurufen.

11.2.8. Zugriffsprüfung und AccessibleObject

Die Klassen Field, Constructor und Method sind alle Unterklassen der Klasse AccessibleObject. Diese gestattet es Ihnen die Zugriffsmodifier einer Klasse zu überprüfen, beispielsweise public und private. In der Regel werden die Zugriffe auf Methoden durch bestimmte Zugriffsrechte und Modifier geregelt. Falls Sie auf ein Datenfeld nicht ohne Reflection zugreifen können, werden Sie dies auch mit Reflection nicht können.

Falls Sie allerdings die Methode setAccessible() auf true setzen, können Sie die Zugriffsrechte umgehen, falls die Policy Sie nicht daran hindert.

Die AccessibleObject Methoden:

public void setAccessible(boolean flag)
: Falls dieses Flag auf true steht, werden die sprachabhängigen Zugriffsmechanismen ausgeschaltet; sonst gelten die Zugriffsrechte gemäss Sprachdefinition.
Falls Sie versuchen auf die Zugriffsrechte zu ändern, aber nicht berechtigt sind, wird eine SecurityException geworfen.

public static void setAccessible(AccessibleObject[ ] array, boolean flag)
: Damit können Sie das Zugriffsflag für ein ganzes Datenfeld von Objekten setzen.
Diese Methode wurde einfach aus Bequemlichkeit definiert.
Auch diese Methode kann eine SecurityException werfen.

public boolean isAccessible()
: damit können Sie den aktuellen Status des Zugriffsflags abfragen.

11.2.9. Arrays

Ein Array ist ein Objekt, besitzt aber keine Members. Daher liefert eine Abfrage des Class Objekts eines Arrays lauter leere Datenfelder für Konstruktoren, Methoden oder Datenfelder.

Die Klasse Array liefert aber statische Methoden, mit deren Hilfe die Elemente des Arrays gesetzt und gelesen werden können. Auch das Kreieren eines Arrays mit der newInstance() Methode ist möglich:

public Object newInstance(Class componentType, int length)
: liefert eine Referenz auf ein neues Array der angegebenen Länge mit / für Komponenten des angegebenen Typs.

public Object newInstance(Class componentType, int[ ] dimensions)
: liefert eine Referenz auf ein mehrdimensionales Array, mir Dimensionen gemäss der Angabe in der Methode und Datentypen als Elemente gemäss dem componentType.
Falls das dimensions Array leer ist, oder nicht korrekt definiert ist (beispielsweise eine Dimension aufweist, die grösser als erlaubt ist), wird eine IllegalArgumentException geworfen.

Bei Basisdatentypen müssen Sie ein entsprechendes Class Objekt erhalten: byte.class liefert ein Class Objekt für ein Byte Array:

```
byte[ ] ba = (byte[ ])Array.newInstance(byte.class, 13);
```

ist äquivalent zu:

```
byte[ ] ba = new byte[13];
```

Und hier noch ein Beispiel für die zweite Methode:

```
int [ ] dims = {4, 4};
double[ ] [ ] matrix = (double[ ][ ])Array.newInstance(double.class, dims);
```

ist äquivalent zu:

```
double[ ][ ] matrix = new double[4][4];
```

Da der Komponententyp selber auch ein Array sein kann, kann eine Dimension resultieren, die wesentlich grösser ist, als in der Dimensionsangabe.

Auch die Array Klasse besitzt get...() und set...() Methoden, um einzelne Komponenten eines Arrays herauszulesen oder zu modifizieren.

Beispiel:

```
int[ ] xa = new int[10]; ...      int j= xa[i];
```

ist äquivalent zu

```
Array.get(xa,i); // bis auf das Wrapping
```

Falls Sie eine Komponente auf einen bestimmten Wert setzen möchten:

```
xa[i] = 23;
```

wird zu

```
Array.setInt(xa, i, 23);
```

Die getLength Methode der Array Klasse liefert die Anzahl Komponenten des Arrays.

Selbsttestaufgabe 7

Wie gehabt: modifizieren Sie Ihren Interpreter so, dass Sie auch Arrays definieren und initialisieren können und deren Werte bestimmen bzw. verändern können.

11.2.10. Packages

Die Methode getPackage() der Class Objekte liefert das dazugehörige Package. Die Package Klasse wird im java.lang Package definiert. Die Methode getPackage() ist static, daneben gibt es auch eine getPackages() Methode, die ebenfalls static ist. Diese Methode liefert in einem Array alle bekannten Packages des Systems.

Die getName() Methode liefert eine voll qualifizierte Darstellung des Package Namens. Package Objekte verhalten sich anders als die anderen Klassen im Reflection Package: Sie können keine Packages definieren. Aber Sie können Informationen über das Package erhalten, beispielsweise die Version und allgemeine Informationen über das Package.

Packages werden an anderer Stelle im Detail besprochen.

11.2.11. Die Proxy Klasse

Die Proxy Klasse gestattet es Ihnen Klassen zur Laufzeit zu kreieren, welche eines oder mehrere Interfaces implementieren. Dies ist ein selten eingesetztes Feature und sollte auch nur mit Vorsicht eingesetzt werden.

Falls Sie beispielsweise die Aufrufe an ein Objekt in ein Log eintragen möchten, um beispielsweise im Fehlerfall einer Transaktion diese rückgängig machen zu können, dann könnten Sie dies von Hand leicht implementieren. Allerdings müssen Sie und alle anderen Personen, die ein solches Problem lösen möchten, immer wieder das selbe implementieren.

Die Java Lösung sieht anders aus: Java liefert Ihnen ein allgemein einsetzbares Hilfsmittel, die Proxy Klasse. Das Proxy Modell besteht darin, dass Sie das Proxy Objekt bzw. dessen Methoden aufrufen, statt die Methoden des Zielobjekts. Falls Sie Proxy.getProxyClass( ) aufrufen, mit einem Class Loader und einem Array von Interfaces, erhalten Sie ein Class Objekt für den Proxy. Proxy Objekte besitzen einen Konstruktor, dem man ein InvocationHandler Objekt mitgeben kann. Das Constructor Objekt erhalten Sie aus dem Class Objekt. Der Aufruf der newInstance(...) dieses Objekts liefert ein neues Proxy Objekt. Dieses Objekt implementiert alle Interfaces, die man als Argumente dem Konstruktor mitgegeben hat.

Sie können auch mittels Proxy.newProxyInstance(...) mit einem Class Loader und einem Array von Interfaces sowie einem InvocationHandler als Parameter ein solches Proxy Objekt konstruieren.

Aufrufe der in den Interfaces deklarierten Methoden werden in den InvocationHandler umgeleitet und dort mittels der invoke(...) Methode ausgeführt.

Ein generischer Debugger könnte mit einem Proxy Objekt etwa folgendermassen aussehen:

```
public class DebugProxy implements InvocationHandler {
      private final Object obj;          // zugrunde liegendes Objekt
      private final List methods;        // benutzte Methoden
      private final List history;        // einfache History

      private DebugProxy(Object obj) {
            this.obj = obj;
            methods = new ArrayList();
            history = Collections.unmodifiableList(methods);
      }

      public synchrionized static Object proxyFor(Object obj) {
            Class objClass = obj.getClass();
            return Proxy.newProxyInstance(
                  objClass.getClassLoader(),
                  objClass.getInterfaces(),
                  new Debugproxy(obj) );
      }

      public Object invoke(Object proxy, Method method, Object[ ] args)
            throws Throwable {

            methods.add(method);
```

```
            return method.invoke(obj, args);
      }

      public List getHistory() {
            return history;
      }
}
```

Falls Sie nun einen Debug Proxy benötigen, können Sie diesen folgendermassen instanzieren:

```
Object proxyObj = DebugProxy.proxyFor(realObj);
```

Das Objekt proxyObj würde alle Interfaces implementieren, welche realObj implementiert, zusätzlich alle Methoden von Object. Ein Methodenaufruf auf dem Objekt proxyObj würden an Aufruf in realObj weitergeleitet.

Falls Sie die getProxyClass zweimal aufrufen, wird Ihnen jeweils eines und nur ein Class Objekt zurück geliefert. Falls Sie die Interfaces vertauschen, erhalten Sie unterschiedliche Objekte!

Sie können abfragen, ob ein Objekt ein Proxy Objekt ist, mit der Methode Proxy.isProxy.

11.3. Das Laden von Klassen

Das Java Laufzeitsystem lädt die Klassen immer dann wenn sie benötigt werden. Wie die Klassen im Einzelnen geladen werden, hängt letztlich von der Implementation ab. Aber meistens wird der *class path* Mechanismus verwendet, um die referenzierten Klassen aus Ihren Programmen zu finden und ins Laufzeitsystem zu laden. Der Klassenpfad ist einfach eine Sammlung von Stellen, ab denen Klassen geladen werden können. Dieser Mechanismus funktioniert für Standardanwendungen bestens. Aber Java bietet wesentlich mehr. Daher sollten auch Mechanismen zur Verfügung stehen, mit deren Hilfe Klassen von irgendwo, beispielsweise aus dem Internet geladen werden können. Dazu werden die Class Loader eingesetzt. Mit ihnen können Sie universelle oder spezifische Ladungsvorgänge definieren und Techniken wie beispielsweise eine Verschlüsselung implementieren.

Nehmen wir beispielsweise ein globales Computerspiel. In diesem werden Teile des Programms ab dem Server, andere aber lokal und für den Server remote vom Spieler her geladen.

Dies geschieht mittels derClassLoader Klasse, welche auch eine find(...) Methode besitzt. Diese muss jeweils bei Ihrer Implementation überschrieben werden, sofern Sie im Sinne haben, etwas eigenes zu implementieren:

protected Class findCass(String name)
throws ClassNotFoundException

Lokalisiert den Bytecode, welcher durch den Klassennamen bezeichnet wird und lädt diesen Bytecode in die Virtual Machine und liefert ein Class Objekt zurück, welches das generierte Objekt in der VM repräsentiert.

In unserem Beispiel mit dem weltweiten Computerspiel könnte unser Spielerloop folgendermassen aussehen:

```
public class Game {
      static public void main(String args[ ]) {
            String name; // der Klasse
            while ( (name=getNextPlayer() ) != null) {
                  try {
                        PlayerLoader loader = new PlayerLoader();
                        Class classOf = loader.loadClass(name);
                        Player player = (Player)classOf.newInstance();
                        Game game = new Game();
                        player.play(game);
                        game.reportScore(name);
                  } catch(Exception e) {
                        reportException(name, e);
                  }
            } // while neue Player
      } // main
} // Class Game
```

Jeder neue Spiel, jeder neue Spieler kreiert einen neuen PlayerLoader, um die Klasse zu laden, welche das Spiel spielt. Dies geschieht mit der loadClass(...) Methode. Diese liefert ein Class Objekt. Mit dem Class Objekt kann auf das Objekt zugegriffen werden.

Falls Sie bereits eine Klasse geladen haben, können Sie im Nachhinein feststellen, welcher ClassLoader diese Klasse in die JVM geladen hat. Dies geschieht mit der getClassLoader Methode. Systemklassen benötigen keinen ClassLoader: in diesem Fall liefert die getClassLoader Methode null.

ClassLoader definieren Namensräume, Name Spaces, welche die Klassen innerhalb einer Applikation aufteilt. Falls zwei Klassen unterschiedliche ClassLoader besitzen, werden sie als unterschiedlich betrachtet, selbst wenn die Class Bytecode Dateien identisch sind. Jede dieser Klassen hätte eigene statische Variablen und Modifikationen in einer Klasse hätten keinerlei Einfluss auf den Zustand der andern Klasse.

Auch jeder Thread besitzt seinen eigenen ClassLoader, mit dem seine Klassen geladen werden. Dieser Context Class Loader kann bei der Instanzierung des Threads angegeben werden. Falls kein Class Loader angegeben wird, wird der Standard Class Loader des übergeordneten Threads dafür verwendet. Der Class Loader für den ersten Thread wird typischerweise der Gleiche sein, wie jener, mit dem die Applikation geladen wird - der System Class Loader. Mit den zwei Methoden setContextClassLoader und getContextClassLoader kann der Class Loader des Threads abgefragt oder gesetzt werden, sofern Sie die entsprechenden Rechte haben.

11.3.1. Die ClassLoader Klasse

Ein Class Loader kann die Aufgabe Klassen zu laden auch an den übergeordneten Class Loader delegieren. Der übergeordnete Class Loader kann im Konstruktor für Class Loader spezifiziert werden:

protected ClassLoader()
: kreiert einen ClassLoader mit dem System Class Loader als implizit deklarierten Class Loader. Dieser System Class Loader kann mit getSystemClassLoader(...) bestimmt werden.

protected ClassLoader(ClassLoader parent)
: dieser Konstruktor kreiert einen Class Loader mit einem speziellen Parent Class Loader.

Falls Sie einen eigenen Class Loader definieren, sollten Sie beide Konstruktoren definiert werden, auch wenn Sie sie nicht explizit benötigen.

Der System Class Loader ist typischerweise jener Class Loader, der benutzt wird, um die Systemklassen zu laden. Sie erhalten Informationen über diesen ClassLoader mit der statischen getSystemClassLoader(...) Klasse.

Die Oberklasse (parent class) des Class Loaders kann mit der Methode getParent() erfragt werden. Falls diese Oberklasse bereits der System Class Loader ist, kann je nach Implementation auch einfach null zurück gegeben werden.

Class Loaders sind ein integraler Bestandteil der Sicherheitsarchitektur von Java. Daher kann bei nicht ausreichenden Privilegien eine SecurityException geworfen werden.

Die wichtigste Methode der ClassLoader Klassen ist die loadClass Methode:

public Class loadClass(String name) throws ClassNotFoundException
diese liefert ein Class Objekt für die Klasse mit dem angegebenen Namen, wobei falls nötig diese Klasse geladen wird. Falls die Klasse nicht gefunden werden kann, wird eine ClassNotFoundException geworfen.

Die Standardmethode loadClass, welche in der Regel von Ihnen in Ihrem Class Loader überschrieben werden muss, funktioniert etwa folgendermassen:

1. es wird geprüft, ob die Klasse bereits geladen wurde, nach Aufruf der Methode findLoadedClass. Der ClassLoader unterhält eine Tabelle der Class Objekte für alle Klassen, die vom aktuellen Klassenlader geladen wurden. Falls eine Klasse bereits geladen wurd wird findLoaderClass das bereits vorhandene Class Objekt zurück liefern.

2. falls die Klasse noch nicht geladen wurde, wird die Methode loadClass der Oberklasse des Class Loaders ausgeführt. Falls der Klassenlader keine Oberklasse besitzt, dann wird der System Klassenlader aktiv.

3. falls die Klasse immer noch nicht geladen werden konnte, wird findClass ausgeführt, um die Klasse zu suchen und zu laden.

Im obigen Beispiel erweitert die PlayerLoader Klasse die ClassLoader Klasse und überschreibt die findClass(...) Methode wiefolgt:

```
class PlayerLoader extends ClassLoader {
        public Class findClass(String name) throws ClassNotFoundException {
                try {
                        byte[ ] buf = bytesForClass(name);
                        return defineClass(name, buf, 0, buf.length);
                } catch(IOException e) {
                        throw new ClassNotFoundException(e.toString());
                }
        }

        // bytesForClass und weitere Methoden

}
```

Die findClass Methode hat in der Regel zwei Aufgaben:
1. sie muss den Bytecode für diese spezielle Klasse finden und in das ByteArray laden. Dies wird oben durch die bytesForClass(...) Methode erledigt.

2. mittels der defineClass(...) Hilfsklasse wird die Klasse mittels des ByteArrays in die JVM geladen und liefert ein Class Objekt für diese Klasse.

protected final Class defineClass(String name, byte[ ] data, int offset, int length)
throws ClassFormatError
diese Methode liefert ein Class Objekt für die Klasse mit dem Namen name. Sie wird im Byte Array data gespeichert ab offset bis offset + length. Falls dieser Teil des Datenarrays nicht einer legalen Java Klasse entspricht, wird ein ClassFormatError geworfen.
Die defineClass Methode wird zudem eingesetzt, um die Klasse mit dem Class Objekt in die Tabelle einzutragen, so dass sie von der findLoaderClass(...) Methode gefunden wird.

Es existiert eine überladene Form der defineClass(...) Methode, welche zusätzlich einen Parameter, das ProtectionDomain, enthält. Protection Domains werden eingesetzt, um bestimmte Sicherheitsbereiche zu definieren. Beide Formen der Methode können eine SecurityException werfen.

Die bytesForClass(...) Methode liest den Bytecode für die Klasse:

```
protected byte[ ] bytesForClass(String name) throws IOException,
ClassNotFoundException {
      FileInputStream in = null;
      try {
            in = streamFor(name + ".class");    // Definition fehlt
            int length = in.available();        // Anzahl Bytes
            if (length == 0) throw new ClassNotFoundException(name);
            byte[ ] buf = new byte[length];
            in.read(buf);
            return buf;
                                          // catch lassen wir weg
      } finally {
            if (in != null)
                  in.close();
      }
}
```

Diese Methode verwendet eine streamFor(...) Methode, um einen FileInputStream auf den Bytecode der Klasse zu erhalten, unter der Annahme, dass sich der Bytecode in einer Datei mit der Bezeichnung name+".class" befindet. Der rest der Methode ist denkbar einfach: wir lesen die Daten in einem Buffer und liefern diesen Buffer zurück.

Falls die Klasse erfolgreich geladen wurde, kann die findClass(...) Methode sie finden und ein neues Class Objekt (durch defineClass(...) erzeugt) für diese Klasse liefern. Klassen können Sie nicht explizit entladen, falls Sie die Klasse nicht mehr benötigt werden. Sie verwenden die Klasse einfach nicht mehr und starten eventuell den Garbage Collector.

Selbsttestaufgabe 8

Versuchen Sie ein einfaches Spiel, beispielsweise Tic-Tac-Toe zu implementieren und dynamisch zu laden. Hinweis: Sie finden dieses Spiel bei den Demobeispielen von Sun oder auf dem Internet.

11.3.2. Vorbereiten einer Klasse für deren Einsatz

Der Class Loader hilft nur im ersten Schritt des Ladens einer Klasse. Insgesamt werden drei Schritte durchlaufen:

1. Laden:
   dabei wird der Bytecode, der die Klasse implementiert, gelesen und ein Class Objekt kreiert.

2. Linking:
   dabei wird der Byte Code verifiziert und seine Konformität mit der JVM Spezifikation überprüft (Byte Code Verifier). In der JVM wird Speicherplatz reserviert, damit die statischen Datenfelder geladen werden können. Allfällige Referenzen werden aufgelöst und referenzierte Klassen geladen.

3. Initialisierung:
   die Oberklassen werden initialisiert (falls nötig werden diese geladen und gelinked). Dann werden alle statischen Initalizer und die statischen Initialisierungsblöcke der Klassen ausgeführt.

Das Class Objekt wird durch die defineClass(...) Methode geliefert, lediglich für geladene Klassen, also vor dem Linken. Das Linken kann explizit ausgeführt werden indem die resolveClass(...) Methode ausgeführt wird:

protected final void resolveClass(Class c)
    diese Methode linked die spezifizierte Klasse, falls diese noch nicht gelinked wurde.

Die loadClass(...) Methode löst die Referenzen nicht auf. Eine geschützte Version der loadClass(...) Methode, welche zusätzlich ein Flag besitzt, wird dann eingesetzt, wenn bekannt ist, ob die Referenzen bereits aufgelöst wurden oder nicht.

Eine Klasse muss initialisiert werden, bevor eine Instanz der Klasse kreiert werden kann oder eine statische Methode der Klasse eingesetzt wird oder ein nicht-finales Datenfeld der Klasse benutzt wird. Wann, zu welchem Zeitpunkt genau diese Initialisierung geschieht, kann von der Implementation der virtuellen Maschine abhängen.

11.3.3. Laden von Ressourcen

Klassen sind die wichtigsten Ressourcen, die ein Programm benutzt. Aber es gibt viele Programme, welche andere Ressourcen benötigen, beispielsweise Text, Bilder oder Tondateien. Class Loaders müssen auch diese Ressourcen finden und die selben Techniken wie beim Laden der Klassedateien einsetzen.

Wenn wir nochmals unser Spiel betrachten, dann könnte es beispielsweise sein, dass wir noch ein "Buch" laden müssen. Das folgende Programmfragment zeigt, wie so etwas aussehen könnte:

```
String book = "BoldPlayer.book";
InputStream in;
ClassLoader loader = this.getClass().getClassLoader();
if (loader != null)
      in = loader.getResourceAsStream(book);
else
      in = ClassLoader.getSystemResourceAsStream(book);
```

Systemressourcen gehören zu Systemklassen, also jene Klassen, welche keinen Class Loader benötigen. Die statische Methode getSystemResourceAsStream(...) liefert einen InputStream für die benannte Ressource. Im obigen Beispiel wird zuerst überprüft, ob ein Class Loader vorhanden ist. Falls dies nicht der Fall ist, also alles mit dem System Class Loader geladen wurde, dann wird mit der getResourceAsStream(...) Methode ein InputStream zur Ressource bestimmt. Falls ein spezieller Class Loader vorhanden ist (loader = null : der else Teil in der obigen Verzweigung), dann wird mit getSystemResourceAsStream(..) gearbeitet.

Wir könnten das obige Beispiel kompakter schreiben als:

```
String book = "BoldPlayer.book";
InputStream in = BoldPlayer.class.getResourceAsStream(book);
```

Der Standard Class Loader enthält eine Implementation der getResourceAsStream(...) Methode, welche null liefert. Sie müssen diese Methode überschreiben, falls Sie eigene Class Loader definieren. Sie können aber auch andere Techniken verwenden, um Ressourcen zu adressieren.

In unserem Spielebeispiel könnte die Methode folgendermassen implementiert werden:

```
public InputStream getResourceAsStream(String name) {
      try {
            return streamFor(name);
      } catch (IOException e) {
            return null;
      }
}
```

Dieser Code verwendet die selbe streamFor(name) Methode, die wir schon einmal benutzt haben.

Zwei weitere Ressource Methoden - getResource(...) und getSystemResource(...) - liefern ein URL Objekt, mit dem diese Ressource benannt wird.

Mit der getContent Methode des URL Objekts können Sie den Inhalt des URL Objekts abfragen.

Die getResources(...) Methode liefert eine Auflistung, Enumeration, von Objekten, alle mit der selben Adresse, falls es mehrere gibt.

public Enumeration findResources(String name)
liefert eine Aufstellung (Enumeration) aller Ressourcen zum gegebenen Namen.

public java.net.URL findResource(String name)
liefert ein java.net.URL Objekt. Der Standard Class Loader liefert einfach null.

Diese Methoden, mit deren Hilfe Ressourcen ermittelt werden können, fragen zuerst die Oberklassen des Class Loaders ab, oder den System Class Loader, falls der Class Loader keine Oberklasse besitzt.

Hier eine mögliche Implementation für unser Spiel:

```
public java.net.URL findResource(String name) {
      File f = fileFor(name);
      if (f.exists() )
            return null;
      try {
            return f.toURL();
      } catch(java.net.MalformedURLException e) {
            return null;      // sollte nicht passieren
      }
}
```

Die fileFor(...) Methode funktioniert analog zur streamFor(...) Methode. Sie liefert ein File Objekt, welches die Ressource im Dateisystem darstellt.

Selbsttestaufgabe 9

Modifizieren Sie Ihr Spiel so, dass Sie Strategien laden können und jeder Spielerstrategie weitere Ressourcen zugeteilt sind.