Lösungen zu den Denkaufgaben

Übung 5

Was haben sich die Designer von Java dabei gedacht, als sie verboten haben, eine Methode gleichzeitig `abstract` und `static` zu deklarieren?

Antwort: Zu dieser Aufgabe gibt es zwei gute Antworten. Ich erkläre beide anhand der folgenden (hypothetischen, syntaktisch inkorrekten) Klassen.

```
abstract class A {
  abstract void meth();
  abstract static void staticMeth();   // (das ist syntaktisch falsch)
}

class B extends A {
  void meth() { ... }
  static void meth() { ... }
}

A obj = new B();
```

1. Die Sprache Java muss so entworfen sein, dass der Compiler feststellen kann, dass jeder Methodenaufruf-Ausdruck tatsächlich implementierte Methoden verwendet. Für den Ausdruck `obj.meth()` ist das dadurch garantiert, dass das Objekt, auf das `obj` referiert, nur durch Instanziierung einer konkreten Klasse erzeugt werden konnte. Der Compiler weiß nicht, welche konkrete Unterklasse von `A` das Objekt wirklich hat; er weiß aber, dass es eine konkrete Unterklasse ist, in der `meth()` eine Implementierung hat – sonst hätte das Objekt gar nicht erzeugt werden können. Für statische Methoden gibt es dieses Sicherheitsnetz nicht: Man könnte Ausdrücke wie `A.staticMeth()` schreiben und damit versuchen, abstrakte Methoden aufzurufen.

2. Stellen wir uns vor, der Programmierer verwendet weiter unten im Programm den Ausdruck `obj.staticMeth()`. Weil der Compilezeit-Typ von `obj` `A` ist, würde dieser Ausdruck versuchen, die Implementierung von `staticMeth()` aus der Klasse `A` aufzurufen – die natürlich nicht existiert. Hier ist also der Unterschied zwischen Überschreiben (im Falle der nichtstatischen Methode `meth()`) und Verstecken (im Falle der statischen Methode `staticMeth()`) essenziell.

Übung 7

Warum ist es für einen Java-Compiler im allgemeinen nicht möglich, Restrekursion zu optimieren? Mit welchem Sprachkonstrukt aus der heutigen Vorlesung könnte man es ihm doch ermöglichen?

Antwort: Betrachte das folgende Programmfragment:

```
class A {
  void meth() {
    ...
    meth();
  }
}

class B extends A {
  void meth() {
   ...
   super.meth();
   ...
  }
}
```

Die Antwort, mit der ich gerechnet hatte, ging in etwa so. Der Aufruf von `meth()` in der `A`-Implementierung der Methode sieht zwar rekursiv aus, ist es aber im Allgemeinen nicht. Stellt Euch vor, Ihr habt ein Objekt `obj` der Klasse `B`, und Ihr ruft die Methode `obj.meth()` auf. Die Implementierung aus `B` ruft die Implementierung aus `A` auf – aber sobald diese zum Aufruf-Ausdruck `meth()` kommt, wird sie die Implementierung von `B` aufrufen, weil `this` auf ein Objekt der Klasse `B` referiert und die überschreibende Implementierung von `meth()` verwendet wird.

Das bedeutet, dass der Compiler zur Compilezeit grundsätzlich nicht erkennen kann, ob ein Methodenaufruf zur Laufzeit rekursiv sein wird oder nicht. Deshalb kann er auf jeden Fall nicht einfach Restrekursion syntaktisch durch eine Schleife ersetzen. Der einzige Fall, in dem das geht, ist, wenn die Methode oder die ganze Klasse `final` deklariert sind: In diesem Fall weiß der Compiler, dass die Methode nicht überschrieben werden kann.

Inzwischen hat mich Steffen aber darauf hingewiesen, dass der wesentliche Effekt einer Restrekursions-Optimierung – nämlich dass der Methodenaufruf durch einen Sprungbefehl ersetzt wird und der Kellerrahmen der aufrufenden Methodeninstanz (in dem lokale Variablen und Rücksprungadresse stehen) von der aufgerufenen Methodeninstanz direkt wieder verwendet werden kann – im Prinzip doch erzielt werden könnte. Man könnte diese Optimierung nämlich grundsätzlich bei allen Methodenaufruf-Ausdrücken anwenden, deren Werte unmittelbar Rückgabewerte der aufrufenden Methode sind; wir müssen ja in

der aufrufenden Methode nichts mehr erledigen. Das heißt, man kann restrekursions-artig optimieren, ohne dass die Methode überhaupt rekursiv ist. Wir kennen aber den Java-Bytecode nicht genau genug, um zu entscheiden, ob man darin solche Tricks hinschreiben könnte.

Übrigens kann der Just-in-time-Compiler in der JVM auf jeden Fall Restrekursion optimieren, denn zur Laufzeit sind die wahren Typen der Objekte bekannt. Außerdem kann der JIT-kompilierte Code in Sonderfällen (z.B. Exceptions) auf den unoptimierten Bytecode zurückgehen. Die Sun-JVM 1.4.2 optimiert Restrekursion unter Windows, aber nicht unter Linux. Die IBM-Implementierung der JVM unter Linux optimiert Restrekursion ebenfalls.

Übung 8

Schreibe eine Version der Fakultätsfunktion, die Exceptions in dieser Art zweckentfremdet. Ist das guter Programmierstil oder nicht?

Antwort: Die folgende Implementierung tut es:

```
int fak(int n) throws Exception {
  try {
    evilFak(n, 1);
  } catch(Exception e) {
    return Integer.parseInt(e.getMessage());
  }

  throw new Exception( "evilFak hat keine Exception geworfen!" );
}

void evilFak(int n, int akku) throws Exception {
  if( n <= 1 )
    throw new Exception(Integer.toString(akku));
  else
    evilFak(n-1, n*akku);
}
```

Das aktuelle Zwischenergebnis der Fakultät wird im Argument `akku` "nach unten" weitergereicht; es gilt die Invariante, dass $\texttt{n}! \cdot \texttt{akku}$ für jeden Aufruf von `evilFak()` den gleichen Wert hat. Sobald `n` den Wert 1 oder 0 hat, wird eine Exception geworfen, die sofort alle Aufrufinstanzen von `evilFak()` beendet und dann in der einzigen Instanz von `fak()` gefangen wird. Der Exception wird als Argument eine Stringdarstellung des Ergebnisses mitgegeben, die in `fak()` wieder in einen `int`-Wert umgewandelt wird.

Dies ist natürlich grauenvoller Programmierstil. Exceptions sind nicht dafür da, Funktionswerte zurückzugeben, sondern um Fehler zu signalisieren. Das Programm ist zwar korrekt (d.h. es berechnet die richtigen Funktionswerte), aber ein normaler Mensch wird große Schwierigkeiten haben, zu verstehen, was Ihr da macht. Wie verdreht die Verwendung von Exceptions hier ist, merkt man auch daran, dass die Methode `fak()` eine Exception werfen muss, um einen Fehler zu melden, wenn die innere Methode *keine* Exception geworfen hat.

Übung 9

Was haben sich die Designer von Java dabei gedacht, als sie innerhalb einer generischen Klasse mit Typ-Parameter `E` Ausdrücke der Form `new E()` verboten haben?

Antwort: Zunächst mal wissen wir nicht, ob die Klasse, die für `E` eingesetzt wurde, einen Konstruktor ohne Argumente hat.

Es kommt aber noch schlimmer. Für den Typ-Parameter kann ein Programmierer jeden beliebigen Referenz-Datentyp einsetzen, also insbesondere Interfaces und abstrakte Klassen. Von diesen Typen können nicht direkt Instanzen erzeugt werden – aber wieder weiß der Compiler zur Compilezeit der generischen Klasse nicht, was für ein Typ in irgendeiner anderen Klasse hier eingesetzt wurde.