Henning Schwentner

Project Valhalla: Value Types in Java

Da wird was wertvoll

Objekt und Wert sind zwei grundsätzliche Konzepte der Programmierung. Wer eine Programmiersprache entwirft, muss sich für eines der beiden entscheiden. Bei Java findet man bis einschließlich der kommenden Version 9 den Ansatz "Alles ist ein Objekt (bis auf die acht eingebauten Typen)". Mit dem Projekt Valhalla wird sich das ändern: Künftig lassen sich beide Konzepte gemeinsam und gleichberechtigt verwenden.

Java ist eine objektorientierte Programmiersprache. Objekte sind demnach das zentrale Konzept. Doch was kennzeichnet ein Objekt? Erstens hat es eine Identität. Zweitens existiert es in der Zeit. Das heißt, es wird erzeugt und gegebenenfalls irgendwann wieder vernichtet. Drittens hat es einen Zustand, der sich über seine Lebenszeit ändern kann. Ein Objekt ist also etwas Konkretes.

Betrachtet sei das am Objekttyp "Konto". Ein Objekt dieses Typs ist "Konto mit der Nummer 4711". Es wurde irgendwann angelegt (da wurde das Objekt erzeugt) und könnte gekündigt werden – dann würde das Objekt zerstört. Während seiner Lebenszeit kann sich der Kontostand erhöhen und verringern, also der Zustand ändern.

Was ist nun ein Wert? Er hat keine Identität, keinen Zustand und existiert unabhängig von der Zeit. Ein Wert ist demnach eine Abstraktion. Wenn man zum Beispiel alle Paare in der Welt (das sind Objekte) anschaut, könnte eine Abstraktion davon das Konzept (der Wert) "2" sein. Trennt sich ein Paar, hat das keinen Einfluss auf das Konzept. Selbst wenn sich alle Paare trennen sollten, beeinträchtigt das die Abstraktion (den Wert) "2" nicht. Die Idee "2" existiert außerhalb der Zeit, ändert sich nicht und hat auch keine Identität. "2" ist also ein typischer Wert.

Sowohl für Objekte als auch für Werte lassen sich Typen definieren. (Erinnerung an die Programmiersprachentheorie: Ein Datentyp ist eine Menge von Werten plus die darauf definierten Operationen.) Diese unterteilt man dann in Objekt- und Werttypen.

Soweit die allgemeine Sicht. Aber was bedeutet das für Java-Programmierer? Objekttypen (und dadurch Objekte) werden in Java mit Klassen definiert. Sie sind dort Referenztypen. Daraus folgt: Variablen können nie ein Objekt selbst, sondern immer nur eine Referenz auf

ein Objekt enthalten. Die Identität eines Objekts wird durch die Referenz (d. h. Speicheradresse) repräsentiert. Den Vergleich mit einer Identität implementiert man mit den Operatoren "==" und "*!=*". Die Prüfung auf Gleichheit lässt sich pro Typ durch Überschreiben der Methode *equals()* definieren.

Objekte leben auf dem Heap

Um herauszufinden, wo Objekte existieren, muss man sich anschauen, wie Programme ausgeführt werden. Ein Programm (auch Prozess genannt) besteht grob aus zwei Speicherbereichen – dem Call Stack und dem Heap. Der Stack funktioniert so: Ein laufendes Programm befindet sich immer an genau einer Stelle seines Codes (wenn man Multithreading außer Acht lässt). Bei Aufruf einer Methode werden ihre Parameter auf den Stack gelegt und Speicher für die lokalen Variablen reserviert. Im Anschluss führt es die Methode aus, letztere arbeitet also mit Daten. Ist die Methode abgearbeitet, werden ihre Parameter und lokalen Variablen abgeräumt und der Rückgabewert auf den Stack gelegt. Dann befindet man sich wieder in der aufrufenden Methode. Wie groß der (Stack-)Speicherbedarf eines Methodenaufrufs ist, lässt sich schon zur Übersetzungszeit feststellen.

Der Heap ist dagegen einfach ein ungeordneter Haufen Speicher, aus dem das Programm sich beliebig große Stücke reservieren kann. Wie viel Heap-Speicher es genau braucht, zeigt sich erst zur Laufzeit. Um das Abräumen ungenutzter Speicherbereiche müssen sich Entwickler (oder der Garbage Collector) im Gegensatz zum Call Stack selbst kümmern. Die Garbage Collection kostet wertvolle Performance.

Nun zurück zur Frage, wo Objekte "leben". Durch den Einsatz des Operators *new* legt das Programm sie auf dem Heap an.

Werte auf dem Call Stack

Und wie ist es bei Werten? Sie liegen direkt im Speicher der Variable – es sind eben keine Referenztypen. Zwei Variablen mit der gleichen Belegung zählen als gleich. Eine Identität gibt es bei Werten nicht. Werte liegen deshalb direkt auf dem Call Stack vor – oder direkt in den Objekten, zu denen sie gehören.

Als Werttypen gibt es in Java nur die acht vordefinierten (sog. eingebauten) Typen. Das sind *boolean*, *byte*, *char*, *int*, *short*, *long*, *float* und *double*. Variablen enthalten immer den Wert selbst. (In Java gibt es keine Zeiger oder Referenzen auf Werte.) Gleichheit wird durch den Vergleich des Variableninhalts geprüft. Die Operatoren "==" und "*!=*" implementieren darum bei Werten eine Prüfung auf Gleichheit.

Selbstdefinierte Werttypen gibt es in Java im Gegensatz zu beispielsweise C# bisher nicht – und zwar weder technische wie "unsigned" oder "complex" noch fachliche wie "Postleitzahl" oder "Geldbetrag". Eine Krücke ist dafür die Verwendung des Musters der "value-based classes". Solche Klassen haben nur finale Felder. Ihre Methoden dürfen nur erkunden und nicht den Zustand ändern. Außerdem muss die Methode *equals()* so implementiert sein, dass sie nur die Belegung der Felder vergleicht und nicht von der Identität abhängt.

Damit wurde zumindest die Werteigenschaft der Unveränderlichkeit abgebildet. Allerdings sind die Exemplare der "value-based classes" immer noch Objekte (wenn sie auch Werte abbilden). Das heißt, sie existieren auf dem Heap. Daraus folgen zwei Nachteile. Erstens haben sie immer noch eine Identität. Zwei Objekte, die gleich, aber nicht identisch sind, liegen an unterschiedlichen Adressen und haben also unterschiedliche Referenzen. Entwickler dürfen bei diesem Muster den Operator "==" nicht verwenden, weil die Variablen die Referenzen enthalten. Zweitens ist durch die Referenz mehr Speicher nötig. Das ist gerade bei kleinen Dingen spürbar.

Dramatisch wird es, wenn man nicht nur einen einzelnen Wert hat, sondern ein Array von Werten. Ein Beispiel: Bei einem *int*-Array liegen die Werte einfach hintereinander im Speicher (*int* ist hier der eingebaute Typ). Ein Array von *Integer* dagegen ist ein Array von Referenzen auf *Integer*. Eine *int*-Variable belegt 32 Bit, eine Referenz hat auf einem 64-Bit-System 64. Hier wird dreimal so viel Speicher (64 Bit für die Referenz plus 32 für den eigentlichen Wert) belegt, wie nötig. Außerdem liegen zwar die Referenzen in einer Reihe im Speicher, aber die referenzierten Objekte sind dort beliebig verteilt. Das und der Aufwand, der für das Dereferenzieren nötig ist, verlangsamen die Iteration über die Daten.

http://heise.de/-3115485