Implikationen der Vererbung

Um die Implikation der Vererbung besser zu verstehen, wollen wir uns etwas vertieft mit der Speicherung der Datenstrukturen befassen.

Jedes Sprachkonzept einer Programmiersprache bewirkt in der Regel grössere und grosse Speicherbelastungen, aber auch Performance-Probleme.

Wie beeinflussen die modernen Sprachkonzepte generell die Struktur der Programmiersprachen?

- damit OO Konzepte effizient eingesetzt werden können, muss die Programmiersprache zum Beispiel polymorphe Variablen unterstützen. Eine polymorphe Variable ist eine Variable, die als Type X deklariert wurde, aber Werte vom Typ Y enthält.
- da wir zur Compilezeit nicht wissen, wieviel Speicherplatz wir zur Ausführzeit für diese polymorphen Variablen benötigen, werden diese Variablen auf dem Heap, nicht auf dem Stack gespeichert.
- da der Programmierer den Heap schlecht verwalten kann, benötigt man sinnvollerweise einen Garbage Collector. C als Assembler Ersatz und C++ als aufgemotztes C verfügen über keinen Garbage Collector und haben daher in der Regel Probleme mit der Speicherverwaltung, auf Assembler Ebene sicher kein echtes Problem, und viel mehr ist C / C+ nicht, zumindest aus Sicht der Erfinder!

1.1. Polymorphe Variablen

Wir werden uns im nächsten Kapitel ausführlich mit Polymorphismus befassen. Hier wollen wir uns eigentlich mit Fragen der Speicherorganisation "quälen".

Betrachten wir ein einführendes Beispiel:

```
class Figur {
        protected int x;
        protected int y;

        public Figur( int ix, int iy) { x = ix; x = iy; }

        public String beschreibe() {return "keine Ahnung was das wieder sein soll!";}
}

class Quadrat extends Figur {
        protected int seite;

        public Quadrat(int ix, int iy, int is) {super(ix, iy); seite = is; }
}

class Kreis extends Figure {
        protected int radius;

        public Kreis(int ix, int iy, int ir)  { super(ix, iy); radius = ir; }
        public String beschreibe() {return "DAS soll ein Kreis sein, mit Radius "+ir; }
}
```

Schreiben wir noch ein "Hauptprogramm", um diese Klasse testen zu können:

```
class FigurenTest {
        static public void main(String [ ] args) {
                Figur form = new Kreis(10,10,5);
                System.out.println("Beschreibung dieser Figur :"+form.describe());
        }
}
```

Welche Ausgabe liefert dieses Programm, sofern ich mich nicht vertippt habe?

1.2. Memory Layout

Jetzt wollen wir untersuchen, welche Implikationen der Einsatz polymorpher Variablen auf die Speicherverwaltung hat. Zuerst müssen wir beschreiben, wie die Speicherverwaltung in der Regel geschieht, bevor wir uns der eigentlichen Frage widmen!

Sie wissen sicher bereits, dass man zwischen *Heap basierter* und *Stack basierter* Speicherung unterscheiden muss, präziser:

- es gibt *Stack basierte* Speicherlokationen und
- es gibt *Heap basierte* Speicherwerte

Stack-basierte Speicherlokationen hängen mit Prozeduren Entries und Exits zusammen. Immer wenn eine Prozedur ausgeführt wird, wird Speicherplatz für lokale Variablen auf einem Laufzeitstack angelegt. Diese Werte auf dem Stack existieren so lange, wie wie Prozedur ausgeführt wird; anschliessend werden sie gelöscht und der Speicherplatz wird wieder frei gegeben.

Betrachten wir ein einfaches rekursives Programm:

```
class FakultaetTest {  // dummes, aber sehr beliebtes Beispiel
        static public void main (String [] args) {
                int f = Fakultaet(3);
                System.out.println("Fakultaet von 3 ist :"+f);
        }

        static public int Fakultaet(int n) {
                int c=n-1;
                int r;
                if (c>0)
                        r = n * Fakultaet(c);
                else
                        r = 1;
                return r;
        }
}
```

Und so sieht der Stack aus:

0	n : 1	dritte Aktivierung
4	r : 1	
8	c : 0	
0	n : 2	zweite Aktivierung
4	r : ?	
8	c : 1	
0	n : 3	dritte Aktivierung
4	r : ?	
8	c : 2	

Der Wert der Variable r wurde im ersten Aufruf auf 1 initialisiert; für die zwei weiteren Aufrufe muss der Wert noch berechnet werden.

Welche Vorteile besitzt eine Stack basierte Speicherverwaltung?
Die Variablen können als Block angelegt und auch wieder gelöscht werden. Diese Blöcke bezeichnet man in der Regel als *activation records*, Aktivierungssätze. Intern werden die Variablen relativ zu diesen Aktivierungssätzen adressiert, wie wir das in der obigen Skizze auch gemacht haben (0, 4, 8), jeweils für jeden Aktivierungssatz.

Stack basierte Speicherung hat aber auch deutliche Nachteile:

- die Adressen, die numerischen Startpunkte der Variablen, müssen zur Compilezeit berechnet werden, nicht zur Laufzeit. Daher muss der Compiler bereits zur Compilezeit wissen, wie viel Speicherplatz jede Variable im Einzelnen benötigt.
- im Falle polymorpher Variablen fehlt diese Information! Der Speicherplatz ergibt sich erst zur Ausführungszeit, zur Laufzeit. Der Speicherbedarf kann sich auch im Verlaufe der Ausführung ändern!
  Als extremes Beispiel : betrachten Sie die geometrische Figur. Da kann man zur Compilezeit unmöglich wissen, wieviel Speicherplatz konkret zur Laufzeit benötigt wird.

Java verwendet daher für solche Fälle den **Heap** als Speicher, an Stelle des Stacks.

Beim Heap wird Speicherplatz dann angelegt, wenn er benötigt wird. Der Operator **new** reserviert den Speicherplatz; allerdings wird dieser dadurch noch nicht in irgend einer Form

belegt. Es wird eigentlich nur dem Betriebssystem, oder im Falle von Java der Virtuellen Maschine, signalisiert, dass wir später Speicherplatz benötigen werden.

Falls wir den Speicherplatz nicht mehr benötigen, können wir entweder selber explizit frei geben, oder den Garbage Collector Aufräumen lassen.

Zur Laufzeit ist im Detail bekannt, wieviel Speicherplatz benötigt wird. Daher ist es kein (echtes) Problem, zu diesem Zeitpunkt den Speicher zu verwalten.

ALLERDINGS:
auch für Heap basierte Variablen müssen zur Compilezeit zumindest bekannt sein und deren Offsets in irgend einer Form kommuniziert werden. Die Lösung des Dilemmas besteht in einer indirekten Adressierung:

- zur Compilezeit werden Pointer auf dem Stack verwaltet
- diese zeigen auf den Heap, also nicht auf die Variable selbst
- dadurch muss nicht spezifiziert werden, wieviel Speicher echt benutzt wird, lediglich wo diese Information zu finden ist.
- die Pointer werden dann belegt, wenn die Objekte angelegt werden

Man sieht also eine Eigenart von Java:

- **Java ist eine für den Programmierer pointefreie Programmiersprache**
- **Java verwendet intern Pointer für die Verwaltung der Objekte, besteht also so gut wie ausschliesslich aus Pointern**

1.2.1. Alternativen

Ohne auf Details des Assemblerersatzes C/C++ einzugehen, sei doch erwähnt, dass C/C++ eine andere Lösung verwendet. Der Hauptgrund ist darin zu sehen, das C/C++ Pointer als "Datenstrukturen" kennt.

Das Dilemma ist Folgendes:

- da auch in C/C++ die Datenstruktur, zum Beispiel die geometrische Figur, nicht angelegt werden kann, wird sie einfach abgeschnitten: es wird nur die Grundform gespeichert. Der Effekt ist der, dass per Default die Methode "beschreibe()" der Grundform (also nicht die Methode, die wir eigentlich ausführen möchten) ausgeführt wird, nicht die Methode "beschreibe()" die wir anschliessend definiert haben.

1.3. Anweisung

Wie sieht nun die konkrete Anweisung in Java aus?
Da nur die Pointer auf dem Stack gespeichert werden, geschiet in der Anweisung folgendes:

- beide Seiten der Anweisung zeigen auf die gleiche Heaplokation

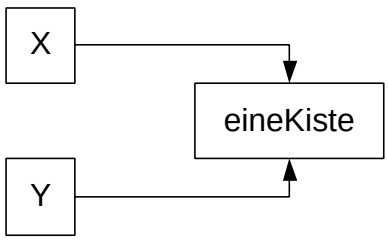
Beispiel:

```
public class Kiste {
        private int wert;

        public Kiste() { wert 0 0; }
        public void setzeDenWert(int v) {wert = v; }
        public int liesDenWert() { return wert; }
}
```

Das folgende Programm verdeutlicht die Stack basierte Zuweisung „ mit Hilfe der Pointer auf den Heap:

```
public class KistenTest {
        static public void main(String [] args) {
                Kiste x= new Kiste();
                x.setzeDenWert(7);    // na ja : geistreich ist anders; X hat jetzt den Wert 7

                Kiste y = x;              // ACHTUNG : nicht new Kiste();
                y.setzeDenWert(11);  // macht Spass

                System.out.println("Inhalt von x "+x.liesDenWert() );       // Ausgabe?
                System.out.println("Inhalt von y "+y.liesDenWert() );       // Ausgabe ?
        }
}
```

Schematisch:



Ich hoffe Ihnen ist klar, warum die Zeiger X und Y auf das gleiche Objekt weisen?

Wegen der Art der Referenzierung sagt man auch, Java verwende eine **Referenz-Semantik**.

1.3.1. Clones

Wahrscheinlich wollte der nicht übermüdete Programmierer des obigen Programmes eigentlich etwas anderes:

- er wollte vermutlich den Zustand, den Inhalt des Objektes kopieren!

Wenn wir das obige Programm leicht verändern, erreichen wir auch dieses Ziel:

```
Kiste y = new Kiste(x.liesDenWert() );
```

Jetzt haben wir eine saubere Kopie und dadurch auch zwei unabhängige Speicherbereiche!

Falls wir die Operation "copy" öfters brauchen, dann sollten wir sie explizit implementieren:

```
public class Kiste {

        ...
        public Kiste copy() {  // Methode copy liefert ein Objekt vom Typus Kiste
                Kiste b = new Kiste();
                b.setzeDenWert(liesDenWert() );
                return b;
        }
        ...
}
```

Dann könnten wir eine Kopie eines Objektes wie folgt erstellen:

```
....
        Kiste y = x.copy();     // ohne Argument, copy ist eine Methode die zum Objekt gehört
```

Jetzt erinnern wir uns an [Gosling] und die Besprechung von **Object**, und der Eigenschaft klonebar zu sein:

- **clone**() legt eine 1:1 Kopie, bitweise, des Objektes an
- das Interface **Cloneable** stellt Objekte dar, die klonebar sind

Hier (endlich) ein Beispiel eines klonebaren Objektes:

```
public class Kiste implements Cloneable {
        private int wert;

        public Kiste() {wert = 0;}
        public void setzeDenWert(int w) { wert = w; }
        public int liesDenWert() {return wert; }

        public Object clone() {
                Kiste k = new Kiste();
                k.setzeDenWert(liesDenWert() );
                return k;
        }
}
```

Die Clone Methode muss natürlich ein Ergebnis vom Typ **Objekt** liefern; das liegt an der Definition des Interfaces **Cloneable** und kann nicht überschrieben werden. Wir müssen also in der Regel das resultierende Objekt casten, um zum gewünschten Ergebnis zu gelangen.

Hier das Hauptprogramm:

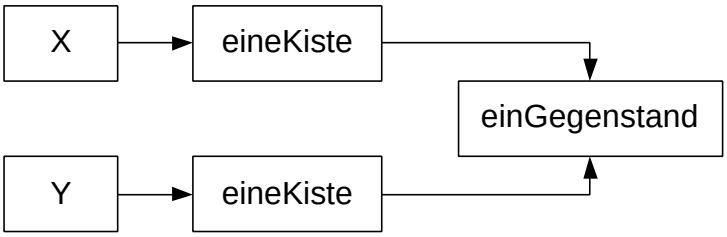
```
public class KistenTest {
        static public void main(String [ ] args) {

                Kiste x = new Kiste();
                x.setzeDenWert(7);

                Kiste y = (Kiste) x.clone(); // Zuweisung einer Kopie von X an Y

                y.setzeDenWert(11);  // einfach einen neuen Wert zuweisen, zum Testen

                System.out.println("Inhalt der Kiste x: "+x.liesDenWert() ); // Ausgabe?
                System.out.println("Inhalt der Kiste y: "+y.liesDenWert() ); // Ausgabe?
        }
}
```

Das Klonen enthält noch eine kleine Finesse:

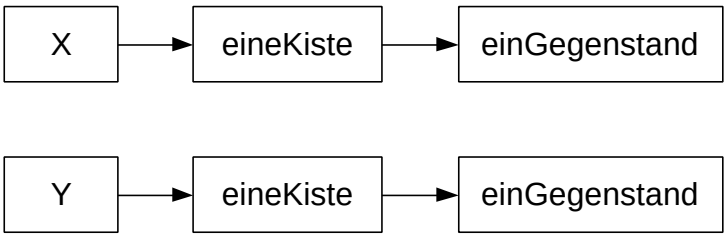
- falls die Objekte X und Y komplexe Objekte sind, sollen dann die gesamten Datenstrukturen aus denen die Objekte hergeleitet wurden, kopiert werden, oder lediglich die Werte der beiden Objekte, deren Zustand?
- in Java unterscheidet man zwei Arten des Klonens:
  - schwache oder untiefe Kopien: *shallow copy*
    dann zeigen X und Y auf ein und die selbe Grundstruktur
  - starke oder tiefe Kopie : *deep copy*
    dann zeigen X und Y auf völlig verschiedene Grundstrukturen : für y wird eine vollständige Kopie, inklusive abgeleiteter Strukturen, hergestellt.

Hier eine schematische Darstellung dieser Sachverhalte:

eine flache Kopie:



und hier eine tiefe Kopie:



Der Programmierer muss natürlich selber entscheiden wie tief er kopieren möchte.

1.3.2. Parameter als spezielle Form der Anweisung

Beim Methodenaufruf übergeben wir in der Regel irgend welche Parameter. In diesen Fällen geschieht eigentlich das selbe wie eben besprochen:
es werden Kopien eines Objektes angelegt, aber es liegt an uns zu entscheiden, wie tief diese Kopie sein soll.

Betrachten wir ein Beispiel:

```
public class KistenTest {
	static public void main(String [ ] args) {
		Kiste x = new Kiste();
		x.setzeDenWert(7);

		wasNun(x);

		System.out.println("Inhalt von x : "+x.liesDenWert() ); // Ausgabe?
	}

	static void wasNun(Kiste y) {
		y.setzeDenWert(11);  // ueberschreiben des Parameters
	}
}
```

1.4. Test auf Gleichheit

Für die Basistypen ist es ein Leichtes festzustellen, ob die Objekte gleich sind oder nicht.

Komplizierter wird's selbst bei den Basistypen, wenn diese nicht vom selben Typus sind.

Beispiel : ist 2 (integer) = 2.0 (floating-point)

In Java kann man natürlich auch die Default-Umwandlungen benutzen:
7 == (3+4);
'a' == '\141';
2 == 2.0;

Im Falle allgemeiner Objekte sieht das schon etwas anders aus!
Definition: Zwei Objekte sollen dann als gleich angesehen werden, wenn ihre Identität die gleiche ist.

Damit haben wir das Problem einfach verschoben: wie finde ich die Identität eines Objektes?

Wir haben, wie beim klonen, in [Gosling] gelesen, dass es eine Methode **equals** gibt, für **Object**.

Und hier ein Beispiel dafür:

```
String a = "abc";
String b = "abc";
Integer c = new Integer(7);
Integer d = new Integer(3+4);

if (a.equals(b) )
        System.out.println("Die Zeichenketten "+a+" und "+b+" sind gleich");

if (c.equals(d) )
        System.out.println("Die Zahlen "+c+" und "+d+" sind gleich");
```

Im Falle des Vergleiches:

```
if (a.equals(c) )
        System.out.println("Die Zeichenketten "+a+" und "+c+" sind gleich");
```

dürfte das Ergebnis klar sein!?!

Wir haben natürlich auch die Möglichkeit, die Methode **equals()** zu überschreiben. Dabei muss man aber einen typischen Fehler vermeiden:

- **equals**() hat folgende Eigenschaften:
  - falls x.**equals**(y), dann y.equals(x)
  - falls x.**equals**(y) und y.**equals**(z), dann x.**equals**(z)
- meistens vergisst man bei der Definition von **equals()** die Symmetrie:
  x wird mit y korrekt verglichen, aber y nicht mit x
  Die Transitivität wird in der Regel auch vergessen.

1.5. Garbage Collector

Wie wir gesehen haben, wird in Java speziell für polymorphe Objekte, mit Hilfe des Heap gearbeitet, nicht mit Stacks, Stacks sind aber von der Verwaltung her eher einfach, da ja die Lebensdauer eindeutig geregelt ist: sobald die Prozedur abgeschlossen ist, kann der temporäre Stack gelöscht werden und das Memory ist wieder clean, theoretisch wenigstens.

Der Garbage Collector in Java ist ein Segen! Alle Assembler oder C/C++ Programmierer, aber auch Objective-C, Object Pascal und ähnliche Sprachen, belasten den Programmierer mit dem Aufräumen. In der Regel funktioniert dies aber nicht.

Wir haben bereits früher Beispiele kennen gelernt, wie der Garbage Collector alte Datenstrukturen aus dem Memory löscht. Da der GC manuell gestartet werden kann, hat der Programmierer auch die Möglichkeit die Ausführung etwas zu kontrollieren. Aber es ist unbestritten, dass ein Garbage Collector aus Sicht der Programm-Performance einen Overhead produziert.

Insgesamt überwiegen aber die Vorteile:
- Programme werden übersichtlicher
- komplexe Aufräumaufgaben durch den Programmierer entfallen
- Programmierer können sich auf das Wesentliche konzentrieren

Also?

1.6. Zusammenfassung

Die Idee der polymorphen Variablen ist sehr mächtig, so mächtig, dass wir uns im Folgenden gleich nochmals diesen Thema, allgemeiner, widmen. Aber man muss sich der Subtilitäten des Polymorphismus bewusst sein. In diesem Kapitel ging es darum, einige dieser Finessen auf zu zeigen.

1.7. Selbsttestfrage

1 Was ist eine polymorphe Variable?
2 Aus Sicht des Sprachimplementierers, welche Arten Speicher stehen mir zur Verfügung?
3 Inwiefern besteht ein Konflikt zwischen polymorphen Variablen und der Berechnung des Speicherbedarfs zum Zeitpunkt der Übersetzung?
4 Erklären Sie, was man unter Referenz-Sematik versteht.
5 Was muss ein Programmierer vorsehen, damit eine Klasse **Cloneable** ist?
6 Erklären Sie den Unterschied zwischen flacher und tiefer Kopie.
7 Inwiefern besteht eine Parallele zwischen einer Zuweisung (X=Y;) und einer Parameterübergabe bei einem Methodenaufruf?
8 Was ist der Unterschied zwischen : X == Y und X.**equals**(Y) ?
9 Welche aufgaben hat der Garbage Collector?
10 Nennen Sie drei Vorteile einer Programmiersprache mit Garbage Collector im Vergleich zu einer Programmiersprache ohne.
Und was könnten bezw. sind Nachteile einer solchen Sprache?

1.8. Übungen

1 Schreiben Sie ein Programm, welches eine geometrische Figur klont, indem die Klasse das **Cloneable** Interface implementiert.

2 Schreiben Sie eine Klasse Kiste, oder Box, falls Sie schreibfaul sind (3 versus 5 Buchstaben), welche **Cloneable** ist, also das Interface **Cloneable** implementiert UND zudem eine tiefe Kopie herstellt.

1.9. Studienhinweise

15.1.1. Lernziele

Nach dem durcharbeiten dieses Kapitels sollten Sie in der Lage sein, zu erklären,

- wie in unterschiedlichen Programmiersprachen (Java versus C/C++) das Problem der polymorphen Variablen behandelt wird.
- welche Konflikte bestehen können zwischen dem Konzept der polymorphen Variablen und Konzepten wie
  Gleichheit von Objekten
  Zuweisungen
  Parameterübergabe

1.9.1. Generelle Hinweise

Dieses Kapitel dient der Vertiefung des Stoffes, den wir in sehr komprimierter Form bei [Gosling] kennen gelernt haben.

Lesen Sie nach dem Durcharbeiten dieses Kapitels das entsprechende Kapitel in [Gosling] noch einmal (oder mehrfach) durch. Sicher werden Ihnen einzelnen Konzepte klarer werden.

KONZEPTE DER OO PROGRAMMIERUNG