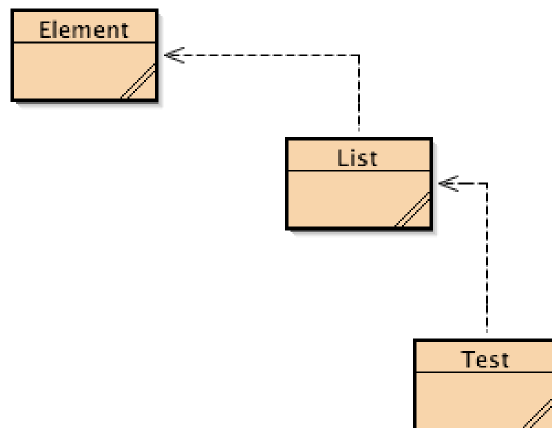
Folge 18 - Vererbung

Workshop

18.1 Ein einfacher Fall der Vererbung

Schritt 1 - Vorbereitungen

Besorgen Sie sich - vielleicht aus einer der Übungen der Folge 17 - ein fertiges und lauffähiges Listenprojekt, so ähnlich wie in der Abbildung 18-1:



***18-1** Ein typisches Listenprojekt*

Der Quelltext von **Element** könnte so aussehen:

```
public class Element
{
   public int value;
   public Element next, prev;

   public Element(int v)
   {
      value = v;
   }

   public void show()
   {
      System.out.println(value);
   }
}
```

Hier ein möglicher Quelltext von **List**:

```
public class List
{
   Element first, last;

   public List()
   {...}

   public boolean empty()
   {...}


   public void insertFirst(int p)
   {...}

   public void insertLast(int p)
   {...}

   public void deleteFirst()
   {...}

   public void deleteLast()
   {...}

   public int getFirst()
   {...}

   public int getLast()
   {...}

   public void show()
   {...}
}
```

Es sind nur die Methodenrümpfe angegeben, die Inhalte müssen Sie sich dazu denken. Es handelt sich um eine einfache Liste mit einem Zeiger **first** auf das erste Listenelement und einem Zeiger **last** auf das letzte Listenelement, also im Prinzip um eine Liste, wie wir sie im Workshop der Folge 17 erarbeitet haben. Wenn die Liste leer ist, existieren keine Listenelemente, **first** und **last** zeigen beide auf `null`. Dummy-Elemente wie bei der „NRW-Liste" sind also nicht vorhanden.

Kurze Methodenübersicht:

Manipulierende Methoden:

insertFirst() / **insertLast()**: Ein neues Element wird vorne / hinten in die Liste eingefügt.

deleteFirst() / **deleteLast()**: Das erste / letzte Element wird gelöscht.

Sondierende Methoden:

getFirst() / **getLast()**: Der Wert des ersten / letzten Elements wird zurück geliefert.

empty(): Es wird ermittelt, ob die Liste leer ist.

Anzeigemethoden:

show(): Die Liste wird in der Konsole angezeigt.

Ganz am Ende dieses Abschnitts wollen wir uns noch den Quelltext einer Testklasse ansehen:

```
public class Test
{
   List l;

   public Test()
   {
      l = new List();
      l.insertFirst(10);
      l.insertFirst(20);
      l.insertFirst(30);
      l.insertFirst(40);
      l.insertFirst(50);
      l.insertFirst(60);
      l.show();
   }
}
```

Achtung: Hier werden noch nicht *alle* Methoden getestet.

Übung 18.1 (9 Punkte)

Implementieren Sie alle Methoden der Klasse **List** und erweitern Sie die Testklasse **Test** so, dass alle Methoden gründlich getestet werden.

Konstruktor und empty(): je 1/2 Punkt; insertFirst(): 2 Punkte; insertLast(): 1 Punkt; deleteFirst(): 2 Punkte; deleteLast(): 1 Punkt; getFirst() und getLast(): je 1/2 Punkt; show(): 1 Punkt.

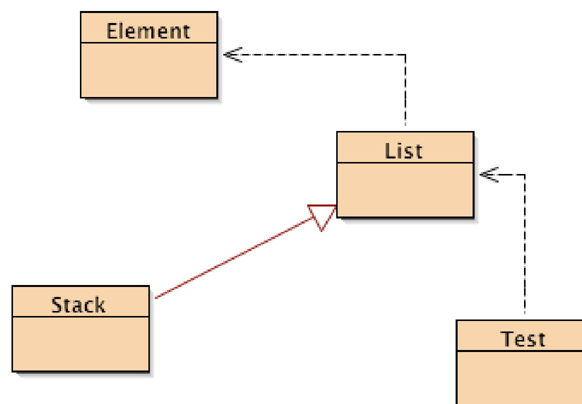
Schritt 2 - Jetzt wird's ernst

Erstellen Sie in Ihrem Projekt eine neue Klasse **Stack**, und zwar wie folgt:

```
public class Stack extends List
{

}
```

Kompilieren Sie diesen kurzen Quelltext und Sie erhalten folgendes UML-Diagramm im BlueJ-Fenster:



***18-2** Stack ist eine Tochterklasse von List.*

Durch das Schlüsselwort `extends` haben wir bewirkt, dass die Klasse **Stack** keine gewöhnliche Klasse ist, sondern eine **abgeleitete Klasse** von **List**. Umgekehrt kann man jetzt sagen, dass **List** die **Basisklasse** von **Stack** ist.

Basisklasse

Eine normale oder abstrakte Klasse, von der eine oder mehrere Klassen alle Attribute und Methoden geerbt haben. Statt „Basisklasse" sagt man auch häufig „Mutterklasse".

Abstrakte Klasse

Eine Basisklasse, von der keine eigenen Objekte erzeugt werden können. Abstrakte Klassen dienen einzig und allein dem Zweck, Vorlagen für abgeleitete Klassen zu sein.

Abgeleitete Klasse

Eine Klasse, die von einer Basisklasse alle Attribute und Methoden geerbt hat, die aber zusätzlich noch eigene Attribute und Methoden besitzen kann. Außerdem kann eine abgeleitete Klasse die geerbten Methoden modifizieren.

Statt „abgeleitete Klasse“ sagt man auch häufig „Tochterklasse“ oder „Kindklasse“.

Schritt 3 - Ein Experiment

IST-Beziehung

Die Beziehung zwischen einer Basisklasse und einer abgeleiteten Klasse, meist auch als „Vererbung“ bezeichnet. Eine abgeleitete Klasse hat alle Attribute und Methoden der Basisklasse und IST damit quasi die Basisklasse.

Die Beziehung zwischen den beiden Klassen **Stack** und **List** ist eine solche IST-Beziehung, denn alle **Stack**-Objekte sind gleichzeitig auch **List**-Objekte. Wir wollen diese Behauptung jetzt experimentell beweisen - schließlich gehört die Informatik zu den Naturwissenschaften. Schreiben Sie dazu die Klasse **Test** folgendermaßen um:

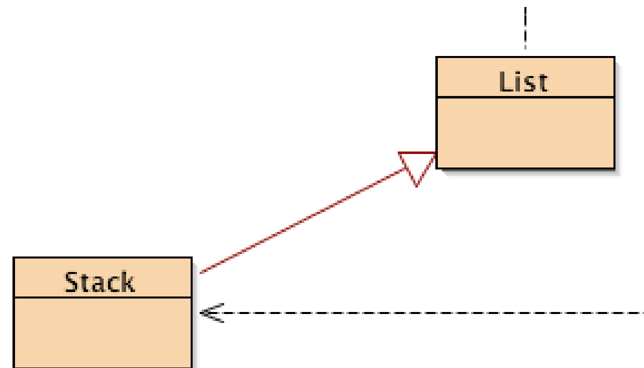
```
public class Test
{
   Stack l;

   public Test()
   {
      l = new Stack();
      l.insertFirst(10);
      l.insertFirst(20);
      l.insertFirst(30);
      l.insertFirst(40);
      l.insertFirst(50);
      l.insertFirst(60);
      l.show();
   }
}
```

Sie müssen nur an zwei Stellen das Wort **List** durch das Wort **Stack** ersetzen. Sie werden feststellen, dass das Testprogramm noch genau so funktioniert wie bisher. Dies ist der Beweis dafür, dass ein **Stack** eine **List** IST bzw. dass eine IST-Beziehung zwischen **Stack** und **List** herrscht.

Schritt 4 - Vererbung und IST-Beziehungen

Dass ein **Stack** eine **List** ist, haben wir eben experimentell bewiesen. Alle Objekte der Klasse **Stack** haben die gleichen Attribute und Methoden wie alle Objekte der Klasse **List**, somit *sind* **Stack**-Objekte auch gleichzeitig **List**-Objekte. Die Klasse **Stack** hat alle Attribute und Methoden von der Klasse **List** *geerbt*, daher spricht man allgemein auch von **Vererbung**, wenn man eine **IST-Beziehung** meint. In BlueJ wird eine solche IST-Beziehung zwischen zwei Klassen durch einen besonders gekennzeichneten Pfeil dargestellt:



***18-3** Darstellung einer IST-Beziehung im UML-Klassendiagramm von BlueJ (Ausschnitt)*

Schritt 5 - Noch ein Experiment

Betrachten Sie folgende Basisklasse sowie die davon abgeleitete Klasse:

```
public class Basisklasse
{
    public void pubAnzeigen()
    {
      System.out.println(„pubAnzeigen„);
    }

    private void privAnzeigen()
    {
      System.out.println(„privAnzeigen„);
    }
}
```

```
public class Tochterklasse
extends Basisklasse
{
    public int anzeigen()
    {
        pubAnzeigen();
        privAnzeigen();
    }
}
```

Der BlueJ-Compiler zeigt beim Übersetzen einen Fehler an: „cannot find symbol - method privAnzeigen()". Eine Tochterklasse erbt eben nicht *alle* Methoden und Attribute der Basisklasse, sondern nur die *öffentlichen*, durch das Schlüsselwort `public` gekennzeichneten. Die Methode **privAnzeigen()** war aber nicht als öffentlich deklariert, daher kennt die Tochterklasse diese Methode nicht.

public / private / protected

Attribute und Methoden einer Klasse **K** können mit einem der Schlüsselworte „public", „private" oder „protected" spezifiziert werden.

public: Eine Methode bzw. ein Attribut ist öffentlich. Alle Tochterklassen von **K** und alle Klassen, die Objekte **k** von **K** einbinden, können auf die Methode bzw. das Attribut zugreifen.

private: Eine Methode bzw. ein Attribut ist vollkommen geschützt. Weder Tochterklassen von **K** noch Klassen, die Objekte **k** von **K** einbinden, können auf die Methode bzw. das Attribut zugreifen.

protected: Tochterklassen von **K** können auf das Attribut bzw. die Methode zugreifen, nicht jedoch Klassen, die Objekte **k** von **K** einbinden.

Schritt 6 - Der Stack verhält sich wie ein Stack

Unser **Stack** aus den vorhergehenden Schritten erbt zwar alle öffentlichen Attribute und Methoden der Klasse **List**, damit verhält er sich aber auch *genau so* wie eine **List**. Ein echter Stack zeigt aber ein völlig anderes Verhalten, wie Sie wissen. Das Hinzufügen neuer Elemente geschieht mit der **Push**-Operation und nicht mit der InsertBefore-Operation, das Entfernen des zuletzt hinzugefügten Elementes mit der **Pop**-Operation und nicht mit der DeleteFirst-Operation. Den Wert des zuletzt hinzugefügten Elementes erhält man mithilfe der **Top**-Operation und nicht mit der GetFirst-Operation:

List	Stack
insertBefore()	**push()**
deleteFirst()	**pop()**
getFirst()	**top()**

18-4 *Vergleich der Methoden der Liste und des Stacks*

Wir müssen die Klasse **Stack** daher um die für einen Stack *typischen* Methoden ergänzen. Machen wir uns am Beispiel der **push()**-Methode klar, wie das geht:

```
public class Stack extends List
{
   public void push(int n)
   {
      insertFirst(n);
   }
}
```

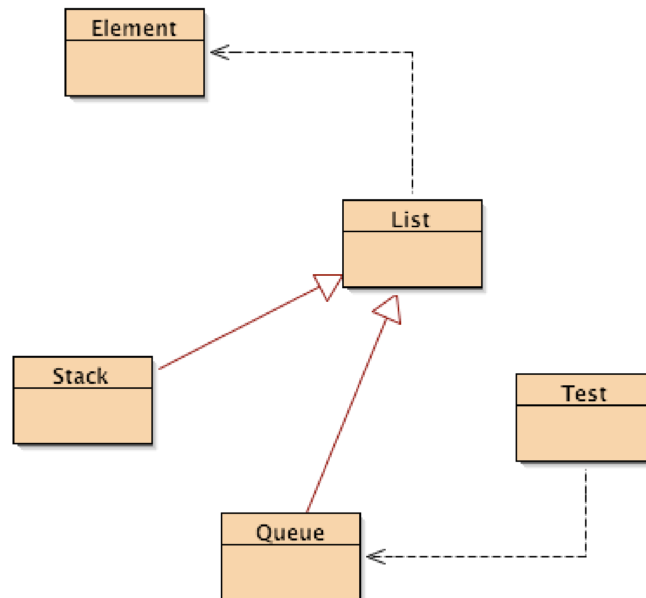
Das Grundprinzip ist ganz einfach: Wir überlegen uns, welche Methode der Basisklasse der **push()**-Methode am besten entspricht. Ganz eindeutig ist das **insertFirst()**. Daher besteht der Quelltext der **push()**-Methode tatsächlich nur aus dem Aufruf der **insertFirst()**-Methode.

Übung 18.2 (2 Punkte)

Ergänzen Sie die Klasse **Stack** auf ähnliche Weise um die noch fehlenden Methoden.

Schritt 7 - Eine Queue

Wir wollen nun eine Klasse **Queue** in unser Projekt aufnehmen und dabei genau so verfahren wie bei der Klasse **Stack**.



18-5 Das Klassendiagramm mit der Queue

Übung 18.3 (2 Punkte)

Erzeugen Sie die Klasse **Queue** und statten Sie sie mit allen typischen Methoden aus.

Die Klasse **List** ist nun die Mutterklasse von zwei Tochterklassen, **Stack** und **Queue**. Beide Tochterklassen haben Zugriff auf alle Methoden und Attribute von **List**, SIND also gleichzeitig Listen. Zusätzlich hat jede Tochterklassen eigene Methoden und bei Bedarf auch eigene Attribute.

Schritt 8 - Überschreiben von Methoden

Die Klasse **List** - die Basisklasse von **Stack** und **Queue** - stellt eine Methode ***show()*** zur Verfügung:

```
public void show()
{
   Element h = first;
   while (h != null)
   {
      h.show();
      h = h.next;
   }
}
```

Die Klassen **Stack** und **Queue** sollen die Elemente der Liste ebenfalls anzeigen, allerdings mit einer kleinen Veränderung. Bevor die Elemente angezeigt werden, soll der Benutzer erfahren, ob es sich um einen Stack oder eine Queue handelt. Wir können also die von **List** zur Verfügung gestellte **show()**-Methode nicht unverändert übernehmen, sondern müssen die **show()**-Methode jeweils modifizieren. Dies geschieht durch die Technik des **Überschreibens**.

Am Beispiel der **show()**-Methode für die Klasse **Stack** wollen wir sehen, wie das Überschreiben geht. Wir ergänzen also die Klasse **Stack** um eine eigene **show()**-Methode:

```
public void show()
{
   System.out.println(„Der Stack : „);
   super.show();
}
```

Zunächst wird die für die Klasse **Stack** typische Meldung erzeugt: „Der Stack : „. Und das war's auch eigentlich schon, an sich könnte jetzt die **show()**-Methode der Basisklasse **List** weitermachen. Und genau das wird durch das Schlüsselwort **super** erreicht: Die **show()**-Methode der Basisklasse wird ausgeführt.

Wir haben hier die **show()**-Methode der Mutterklasse **überschrieben**, also durch eine eigene, aber nicht unbedingt bessere **show()**-Methode ersetzt. Mit **super.show()** kann aber immer noch die **show()**-Methode der Basisklasse aufgerufen werden - allerdings nur komplett. Es gibt keine Möglichkeit, lediglich einen Teil der **show()**-Methode aufzurufen.

Überschreiben

Die Ersetzung einer Methode der Basisklasse durch eine entsprechende Methode der Tochterklasse. Die Basis-Methode kann mit Hilfe des Schlüsselworts **super** von der Tochter-Methode aufgerufen werden. Dies ist aber nicht zwingend notwendig; die Basisklasse kann auch komplett überschrieben werden.

Ist die Basisklasse eine abstrakte Klasse, so kann man abstrakte Methoden deklarieren, indem man ihnen das Schlüsselwort **`abstract`** voranstellt:

```
public abstract void test()
```

Eine solche Methode *muss* von den Tochterklassen überschrieben werden. Daneben können abstrakte Basisklassen aber auch ganz normale, nicht-abstrakte Methoden enthalten. Diese Methoden müssen dann von den Tochterklassen nicht überschrieben werden. Sollte es notwendig / erwünscht sein, können sie aber wie jede andere Methode einer Basisklasse überschrieben werden.

Schritt 9 - Konstruktoren

Die Tochterklassen **Stack** und **Queue** haben keinen Konstruktor - zumindest haben wir keinen programmiert. Was passiert jetzt, wenn Objekte von **Stack** und **Queue** erzeugt werden? In diesem Fall wird eine Art Standard-Konstruktor aufgerufen, der nichts Besonderes macht, außer eben die gewünschten Objekte zu erzeugen.

Wir wollen nun wieder ein Experiment machen, und dazu ergänzen wir die Klasse **Queue** einmal um einen Konstruktor:

```
public class Queue extends List
{
   public Queue()
   {
      System.out.println(„Ein neues Queue-Objekt
      wird erzeugt...„);
   }
```

Jetzt rufen wir wieder unser Testprogramm auf, in der Erwartung, dass nichts mehr funktioniert. Schließlich haben wir ja den Standard-Konstruktor von **Queue** durch einen eigenen ersetzt, der lediglich eine Meldung ausgibt. Und der Konstruktor der Mutterklasse **List** wird offensichtlich nicht aufgerufen.

Entgegen der Erwartungen funktioniert das Testprogramm einwandfrei. Ganz am Anfang, wenn die Zeile

```
s = new Queue();
```

ausgeführt wird, erscheint in der Konsole die Meldung

```
Ein neues Queue-Objekt wird erzeugt…
```

und dann werden munter die **enqueue()** und **dequeue()**-Methoden aufgerufen, ohne dass ein Fehler auftritt. Offensichtlich ist es völlig egal, ob man eine Tochterklasse mit einem Konstruktor ausstattet; der Konstruktor der Mutterklasse wird auf jeden Fall aufgerufen.

Ein weiteres Experiment soll jetzt die Frage klären, ob zuerst der Konstruktor der Tochterklasse oder zuerst der der Mutterklasse aufgerufen wird. Dazu ergänzen wir den Konstruktor der Mutterklasse um eine Meldung:

```
public List()
{
   first = last = null;
   System.out.println(„Konstruktor der Mutterklasse„);
}
```

Wenn wir uns nun die Ausgabe in der Konsole anschauen:

```
Konstruktor der Mutterklasse
Ein neues Queue-Objekt wird erzeugt..
…
```

haben wir die Frage beantwortet: Der Konstruktor der Mutterklasse wird zuerst aufgerufen, und dann werden die Anweisungen ausgeführt, die im Konstruktor der Tochterklasse stehen.