Kapitel 7

Programmieren im Großen

Die sinnvolle Strukturierung des zu entwickelnden Programmcodes ist eine der Grundvoraussetzungen für die erfolgreiche Durchführung größerer Softwareprojekte. Dieses Kapitel soll zeigen, wie in Java über Klassen und Vererbung hinaus die strukturierte Programmierung mit Hilfe von Schnittstellen und Paketen unterstüzt wird. Zur einheitlichen Fehlerbehandlung, ohne den Programmcode mit bedingten Abfragen zu überfrachten, gibt es eine spezielle Ausnahmebehandlung in Java, die wir ebenfalls kennenlernen werden.

7.1 Schnittstellen

Es sei die in Abbildung 7.1 dargestellte Klassenhierarchie gegeben (vgl. 5.14.6).

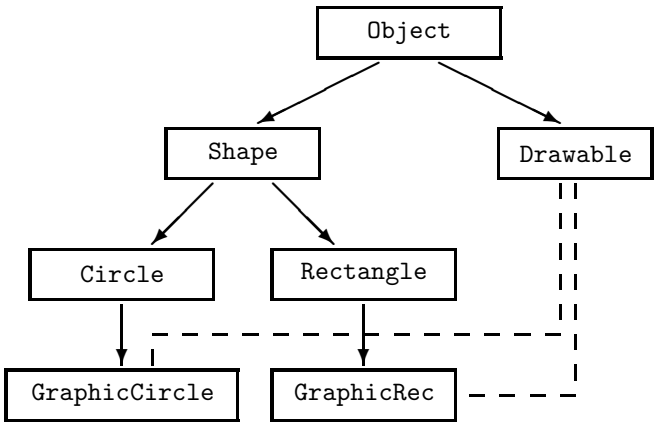


Abbildung 7.1: Eine Klassenhierarchie

Bei Instanzen der Klassen `GraphicCircle` und `GraphicRec` handelt es sich um Objekte, die man zeichnen kann. Um diese "zeichenbaren" Objekte einheitlich be-

handeln zu können, wäre es wünschenswert, eine abstrakte Oberklasse `Drawable` der beiden Klassen zu haben.

Problem: Es gibt nur Einfachvererbung in Java.

Lösung: Schnittstellen (engl. *Interfaces*).

Eine Schnittstelle kann man sich als abstrakte Klasse vorstellen, die nur abstrakte objektbezogene Methoden enthält. Während eine abstrakte Klasse auch nicht-abstrakte Methoden definieren darf, sind in einer Schnittstelle alle Methoden implizit abstrakte Methoden. Neben abstrakten Methoden darf eine Schnittstelle nur Konstanten enthalten.

Beispiel 7.1.1 Eine Schnittstellendeklaration:

```
import java.awt.Graphics;

public interface Drawable {
  public void draw(Graphics g);
}
```

Eine Klasse darf gleichzeitig eine andere Klasse erweitern und (evtl. mehrere) Schnittstellen implementieren, wie folgendes Beispiel (vgl. Abschnitt 5.9) zeigt.

Beispiel 7.1.2

```
  import java.awt.Color;
  import java.awt.Graphics;

  public class GraphicCircle extends Circle implements Drawable {
    protected Color outline;      // Farbe der Umrandung
    protected Color fill;         // Farbe des Inneren

    public GraphicCircle(int x,int y,int r,Color outline,Color fill) {
      super(x,y,r);
      this.outline = outline;
      this.fill = fill;
    }

    public void draw(Graphics g) {
      g.setColor(fill);
      g.fillOval(x-r, y-r, 2*r, 2*r);
      g.setColor(outline);
      g.drawOval(x-r, y-r, 2*r, 2*r);
    }
  }
```

Eine Schnittstelle ist ein Ausdruck reinen Entwurfs, wohingegen eine (abstrakte) Klasse eine Mischung aus Entwurf und Implementierung ist. Schnittstellen können auch mit Hilfe von `extends` erweitert werden. Im Gegensatz zu Klassen können sie mehr als eine Schnittstelle erweitern. Die Menge der *Obertypen* einer Klasse besteht aus der von ihr erweiterten Klasse und den von ihr implementierten Schnittstellen einschließlich der Obertypen dieser Klasse und dieser Schnittstellen. Der Typ eines Objektes ist also nicht nur seine Klasse, sondern auch jeder seiner Obertypen einschließlich der Schnittstellen. Das folgende Beispiel demonstriert dies.

Beispiel 7.1.3

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class DemoShape extends Applet {

  public void paint(Graphics g) {

    Shape[] shapes = new Shape[3];
    Drawable[] drawables = new Drawable[3];

    Drawable gc = new GraphicCircle(300,200,200,Color.red,
                                    Color.blue);
    Drawable gr1 = new GraphicRec(450,200,100,300,Color.green,
                                    Color.yellow);
    Drawable gr2 = new GraphicRec(50,400,300,100,Color.black,
                                    Color.magenta);

    shapes[0] = (Shape) gc;
    shapes[1] = (Shape) gr1;
    shapes[2] = (Shape) gr2;

    drawables[0] = gc;
    drawables[1] = gr1;
    drawables[2] = gr2;

    double totalArea = 0;
    for(int i=0; i<shapes.length; i++) {
      totalArea = totalArea+shapes[i].area();
      drawables[i].draw(g);
    }
```

```
    Double total = new Double(totalArea);
    String str = "Total area = "+total.toString();
    g.setColor(Color.black);
    g.drawString(str,100,550);
  }
}
```

7.1.1 Beispiel: Die vordefinierte Schnittstelle `Enumeration`

Die Schnittstelle `Enumeration` dient zur Aufzählung aller Elemente eines Datentyps. Sie deklariert zwei Methoden:

`public abstract boolean` **hasMoreElements**`()`
: liefert `true` zurück, wenn die Aufzählung noch mehr Elemente (als bisher aufgezählt) enthält. Sie darf auch mehr als einmal zwischen aufeinanderfolgenden Aufrufen von `nextElement()` aufgerufen werden.

`public abstract Object` **nextElement**`()`
: gibt das nächste Element der Aufzählung zurück. Aufrufe dieser Methode zählen aufeinanderfolgende Elemente auf. Wenn keine weiteren Elemente existieren, wird `NoSuchElementException` ausgelöst (vgl. Abschnitt 7.3).

Beispiel 7.1.4 Wir wollen alle Elemente eines Stacks aufzählen. Die Klasse Stack (vgl. Abschnitt 5.7) benutzt dazu die Klasse `StackEnum`.

```
import java.util.Enumeration;

class StackEnum implements Enumeration {
  private Stack st;
  private int pos;

  protected StackEnum(Stack stack) {
    st = stack;
    pos = stack.top;
  }

  public boolean hasMoreElements() {
    return (pos != -1);
  }

  public Object nextElement() {
```

```
    if(pos != -1)
      return st.stack[pos--];
    else
      return null;  // an exception would be better
  }
}
```

Man beachte, dass die Deklaration der Klasse `StackEnum` keinen Gültigkeitsmodifizierer enthält. Damit erhält diese Klasse automatisch (*default*) den Gültigkeitsbereich *package*. Desweiteren ist zu beachten, dass die Datenfelder `stack` und `top` der Klasse `Stack` im Abschnitt 5.7 als `private` deklariert wurden. D.h. sie sind in der Klasse `StackEnum` nicht zugreifbar. Aus diesem Grund müssen diese Datenfelder z.B. als `protected` deklariert werden.
Die Klasse `Stack` muss nun um die Methode `elements` erweitert werden.

```
public Enumeration elements() {
  return new StackEnum(this);
}
```

Die Anwendung der Methode erfolgt dann durch das Erstellen eines neuen Objektes vom Typ `Enumeration`.

```
Enumeration e = stack.elements();
while(e.hasMoreElements())
  System.out.println(e.nextElement());
```

Man beachte, dass die Implementierung von `elements()` völlig verborgen ist und der Typ `StackEnum` in der Klasse `Stack` überhaupt nicht auftaucht (stattdessen wird der Typ `Enumeration` der implementierten Schnittstelle benutzt).

Achtung: Die Schnittstelle `Enumeration` hat keine Schnappschussgarantie. Wird der Inhalt der Sammlung während der Aufzählung verändert, kann das die von den Methoden zurückgegebenen Werte beeinflussen. Ein Schnappschuss würde die Elemente so zurückgeben, wie sie waren, als das `Enumeration`-Objekt erzeugt wurde.

7.2 Pakete

Pakete und Gültigkeitsbereiche wurden schon kurz in Abschnitt 2.4 behandelt. Hier folgt nun eine ausführlichere Beschreibung.
Pakete enthalten inhaltlich zusammenhängende Klassen und Schnittstellen. Die Klassen und Schnittstellen können gebräuchliche öffentliche Namen (wie `get` und `put`) verwenden, denn durch Voranstellen des Paketnamens können eventuelle Namenskonflikte vermieden werden. Beispielsweise macht es Sinn, die Klassen `Stack` und `StackEnum` in ein Paket `stack` zu stecken.[1] Am Anfang jeder Quelltextdatei,

[1] Paketnamen werden nach Konvention klein geschrieben.

deren Klassen zu dem `stack`-Paket gehören sollen, muss die Paketdeklaration `package stack;` stehen.

```
package stack;

public class Stack { ... }
```

```
package stack;

class StackEnum { ... }
```

Der Paketname ist implizit jedem im Paket enthaltenen Typnamen vorangestellt. Benötigt außerhalb des Paketes definierter Code innerhalb des Paketes deklarierte Typen, kann er sich auf zwei Arten auf diese beziehen:

1. Voranstellen des Paketnames, z.B. `stack.Stack`.
2. Importieren (von Teilen) des Paketes durch

   ```
   import stack.Stack;
   ```

   oder auch durch Importieren aller zugreifbaren Klassen eines Paketes mit `*`:

   ```
   import stack.*;
   ```

Der Paket- und Importmechanismus ermöglicht die Kontrolle über möglicherweise in Konflikt geratene Namen. Es gibt z.B. schon eine vordefinierte Java-Klasse `Stack`. Diese befindet sich im Paket `java.util`. Sollen beide Klassen im selben Quelltext verwendet werden, so kann dies geschehen durch:

- Angabe der voll qualifizierten Namen (`stack.Stack` und `java.util.Stack`).
- Importieren von z.B. `java.util.Stack` oder `java.util.*` und verwenden von `Stack` für `java.util.Stack` sowie des vollen Namens von `stack.Stack` bzw. umgekehrt.

7.2.1 Paketinhalte

Enthält eine Quelltextdatei keine Paketdeklarationen, gehen die in ihr deklarierten Typen in ein unbenanntes Paket ein. Pakete sollen sorgfältig entworfen werden, damit sie nur von der Funktionalität zusammengehörige Klassen und Schnittstellen enthalten, denn Klassen in einem Paket können auf die nicht-privaten Attribute und Methoden anderer Klassen frei zugreifen. Pakete können ineinander geschachtelt werden (z.B. `java.util`). Die Schachtelung ermöglicht ein hierarchisches Benennungssystem, stellt aber keinen speziellen Zugriff zwischen Paketen zur Verfügung.

7.2.2 Paketbenennung

Paketnamen sollen einmalig sein. Folgende Konvention soll dies sicherstellen:

```
package DE.Uni-Bielefeld.TechFak.juser.stack;
```

Der Code für ein Paket muss sich in einem diesen Namen widerspiegelnden Verzeichnis befinden (auf Details wollen wir hier nicht eingehen).

7.3 Ausnahmen (Exceptions)

Java stellt einen umfangreichen Mechanismus zur Behandlung sog. Ausnahmen (z.B. Fehlermeldungen) bereit. Wir stellen die Vorteile von Ausnahmen gegenüber der traditionellen Fehlerbehandlung schlagwortartig dar.
Ausnahmen

- sind eine saubere Art, Fehlerprüfungen vorzunehmen, ohne den Quelltext zu überfrachten;
- vermeiden eine Überflutung des grundlegenden Ablaufs des Programms mit vielen Fehlerprüfungen;
- zeigen Fehler direkt an, statt Variablen oder Seiteneffekte auf Datenfelder zu nutzen, die anschließend zu prüfen sind;
- machen Fehlerbedingungen zum expliziten Bestandteil der Spezifikation einer Methode;
- sind daher für Programmierer sichtbar und bei einer Analyse überprüfbar.

Eine Ausnahme ist ein Signal, das auf eine *unerwartete* Fehlerbedingung hinweist. Eine Ausnahme wird ausgelöst durch das `throw`-Konstrukt. Die Ausnahme wird durch ein umgebendes Konstrukt irgendwo entlang der aktuell ablaufenden Methodenaufrufe abgefangen[2]. Wird die Ausnahme nicht abgefangen, tritt die standardmäßige Ausnahmebehandlung in Kraft.
Die Klassenhierarchie für Ausnahmen ist in Abbildung 7.2 dargestellt. Man unterscheidet zwischen *geprüften* und *ungeprüften* Ausnahmen. Es gilt:

- ungeprüfte Ausnahmen erweitern die Klasse `Error` und `RuntimeException` und werden nicht abgefangen;
- geprüfte Ausnahmen erweitern die Klasse `Exception` (nach allgemeiner Übereinkunft *nicht* die Klasse `Throwable`): der Übersetzer überprüft, dass Methoden nur die von ihnen deklarierten Ausnahmen auslösen.

[2]Dies bezeichnet man auch als *catch*.

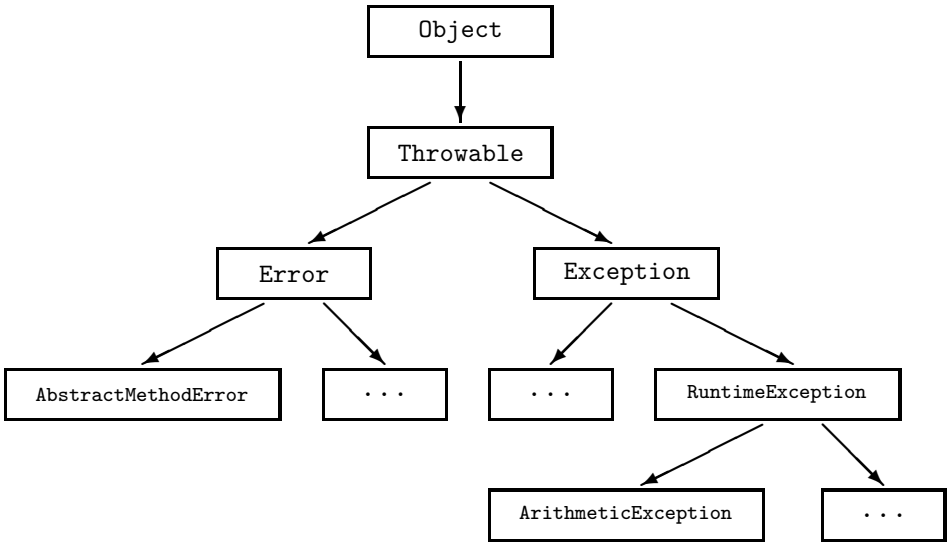
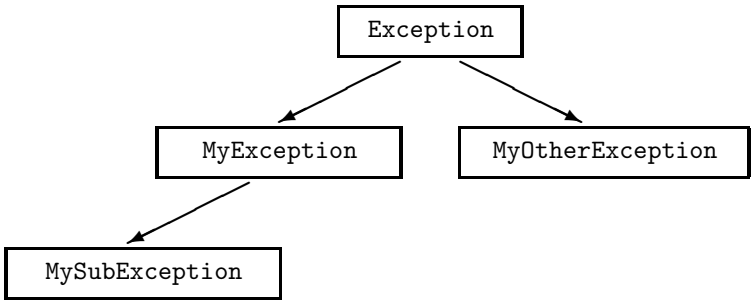


ABBILDUNG 7.2: Klassenhierarchie für Ausnahmen

Jedes Objekt vom Typ `Throwable` hat ein Datenfeld vom Typ `String`, welches mit der Methode `getMessage()` gelesen werden kann; dieser `String` enthält eine Fehlermeldung, die die Ausnahme beschreibt. Das Datenfeld wird bei der Erzeugung eines Ausnahmeobjekts mit Hilfe eines Konstruktors gesetzt.

Beispiel 7.3.1 Der folgende Programmcode implementiert die hier dargestellte Hierarchie von Ausnahmeklassen:




```
class MyException extends Exception {
  public MyException() {
    super();
  }
  public MyException(String s) {
    super(s);
  }
```

```
}

class MyOtherException extends Exception {
  public MyOtherException() {
    super();
  }
  public MyOtherException(String s) {
    super(s);
  }
}

class MySubException extends MyException {
  public MySubException() {
    super();
  }
  public MySubException(String s) {
    super(s);
  }
}
```

Es gibt zwei wesentliche Gründe für die Einführung neuer Ausnahmetypen:

- Hinzufügen nützlicher Informationen;
- der Typ selbst ist eine wichtige Information über die Ausnahmebedingung, da Ausnahmen aufgrund ihres Typs abgefangen werden.

7.3.1 `throw` und `throws`

Ausnahmen werden ausgelöst durch die ein Objekt als Parameter erhaltene `throw`-Anweisung.

Beispiel 7.3.2

```
import java.util.EmptyStackException;

public class Stack {

  ...

  public Object pop() {
    if(empty())
      throw new EmptyStackException();
```

```
    else
      return stack[top--];
  }

  public Object peek() {
    if(empty())
      throw new EmptyStackException();
    else
      return stack[top];
  }
}
```

Alle geprüften Ausnahmen, die in einer Methode ausgelöst und nicht in dieser abgefangen und behandelt werden, müssen deklariert werden durch `throws`. Das Fehlen von `throws` bedeutet, dass die Methode keine geprüfte Ausnahme auslöst. In der Tat ist `EmptyStackException` eine Unterklasse von `RuntimeException`. Daher musste `EmptyStackException` in Beispiel 7.3.2 nicht deklariert werden. Im folgenden Beispiel ist dies anders.

Beispiel 7.3.3 `MyException` und `MySubException` seien wie in Beispiel 7.3.1 definiert.

```
public static int c(int i) throws MyException {
  switch(i) {
    case 0: throw new MyException("input too low");
    case 1: throw new MySubException("input still too low");
    default: return i*i;
  }
}
```

Die Methode `c` kann die beiden Ausnahmen `MyException` und `MySubException` auslösen. Diese müssen daher deklariert werden. Da allerdings `MySubException` eine Unterklasse von `MyException` ist, reicht es, `MyException` zu deklarieren.

Ruft man eine Methode auf, die nach `throws` eine zu berücksichtigende Ausnahme aufführt, hat man die drei Möglichkeiten:

(a) die Ausnahme abzufangen und zu behandeln,

(b) die Ausnahme abzufangen und auf eine eigene Ausnahme abzubilden, die man dann selbst auslösen kann, und deren Typ im eigenen `throws`-Konstrukt deklariert ist,

(c) den Ausnahmetyp im eigenen `throws`-Konstrukt zu deklarieren und die Ausnahme ohne Behandlung die Methode passieren zu lassen (dabei kann ein eigenes `finally`-Konstrukt vorher noch zum Aufräumen aktiv werden).

7.3.2 `try`, `catch` und `finally`

Das allgemeine Muster zum Umgang mit Ausnahmen in Java ist die *try-catch-finally*-Sequenz: Man versucht etwas; wenn dieses eine Ausnahme auslöst, fängt man die Ausnahme ab; und schließlich räumt man nach Ende des normalen Ablaufs oder des Ausnahmeablaufs auf, was immer auch passiert ist. Die Syntax der *try-catch-finally*-Sequenz ist folgendermaßen:

```
try
  block1
catch(exception_type identifier)
  block2
catch(exception_type identifier)
  block3
...
finally
  blockL
```

Der Rumpf (*block1*) der `try`-Anweisung wird ausgeführt, bis entweder eine Ausnahme ausgelöst oder der Rumpf erfolgreich abgeschlossen wird. Wenn eine Ausnahme ausgelöst wurde, werden die `catch`-Konstrukte betrachtet, um darin die Klasse der Ausnahmen oder eine ihrer Oberklassen zu finden. Wird kein solches `catch` gefunden, so wird die Ausnahme über diese `try`-Anweisung hinaus an einen umgebenden `try`-Block zur Behandlung weitergeleitet. Die Zahl der `catch`-Konstrukte ist beliebig, sogar kein `catch` ist möglich. Wenn kein `catch` in einer Methode zur Behandlung einer Ausnahme in Frage kommt, wird die Ausnahme dem Aufrufer der Methode zur Behandlung weitergereicht.

Wenn ein `finally`-Konstrukt in einem `try` vorhanden ist, so wird dessen Block nach allen anderen Anweisungen in dem `try` ausgeführt. Dies erfolgt unabhängig davon, wie die vorherigen Anweisungen abgeschlossen wurden; sei es regulär, durch Ausnahmen oder durch den Ablauf beeinflussende Anweisungen wie `return`[3].

Die `catch`-Konstrukte werden der Reihe nach untersucht. Deshalb ist es ein Fehler, wenn in den `catch`-Konstrukten ein Ausnahmetyp vor einer Erweiterung desselben abgefangen wird (denn das `catch` mit dem Untertyp würde niemals erreicht werden). Dies wird in folgendem Beispiel deutlich.

Beispiel 7.3.4 (vgl. Arnold & Gosling [1], S. 155 ff.)
`MyException` und `MySubException` seien wie in Beispiel 7.3.1 definiert.

```
class BadCatch {
  public void goodTry() {
    try {
```

[3] Auch `break` ist so eine Anweisung, wir werden sie aber nicht genauer behandeln.

```
      throw new MySubException();
    }
    catch(MyException myRef) {
      // Abfangen von sowohl MyException als auch MySubException
    }
    catch(MySubException mySubRef) {
      // Dies wird nie erreicht
    }
  }
}
```

Nur eine Ausnahme wird in einem `try`-Konstrukt behandelt. Wenn eine weitere Ausnahme ausgelöst wird, werden die `catch`-Konstrukte des `try` nicht ein weiteres Mal aktiviert.

Das `finally`-Konstrukt in der `try`-Anweisung ermöglicht es – unabhängig davon, ob eine Ausnahme ausgelöst wurde – abschließend noch ein Programmstück auszuführen (z.B. das Schließen von offenen Dateien; so kann sparsam mit dieser begrenzten Ressource umgegangen werden).

Wenn der `finally`-Block durch `return` oder Auslösen einer Ausnahme verlassen wird, kann ein ursprünglicher Rückgabewert in Vergessenheit geraten.

Beispiel 7.3.5 (Arnold & Gosling [1], S. 157)

```
try {
  // ... irgendein Code ...
  return 1;
}
finally {
  return 2;
}
```

Es würde immer der Wert `2` zurückgegeben werden.

Beispiel 7.3.6 (Flanagan [3], S. 43 ff.)
`MyException`, `MyOtherException` und `MySubException` seien wie in Beispiel 7.3.1 definiert. Um das nachfolgende Programm verstehen zu können, muss man wissen, dass `instanceof` feststellt, ob ein Objekt von einem gegebenen Typ ist. Die `null`-Referenz ist keine Instanz irgendeines Typs, daher ist `false` der Rückgabewert von `null instanceof` *Typ*.

```
public class ThrowTest {
  public static void main(String[] args) {
    int i;
```

```
  try {
    i = Integer.parseInt(args[0]);
  }
  catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Must specify an argument");
    return;
  }
  catch(NumberFormatException e) {
    System.out.println("Must specify an integer argument");
    return;
  }
  a(i);
}

public static void a(int i) {
  try {
    b(i);
  }
  catch(MyException e) {
    if(e instanceof MySubException)
      System.out.print("MySubException:");
    else
      System.out.print("MyException:");
    System.out.println(e.getMessage());
    System.out.println("Handled at point 1");
  }
}

public static void b(int i) throws MyException {
  int result;
  try {
    System.out.println("i= "+i);
    result = c(i);
    System.out.println("c(i)= "+result);
  }
  catch(MyOtherException e) {
    System.out.println("MyOtherException: "+e.getMessage());
    System.out.println("Handled at point 2");
  }
  finally {
    System.out.println();
  }
}
```

```
  public static int c(int i) throws MyException,MyOtherException {
    switch(i) {
      case 0: throw new MyException("input too low");
      case 1: throw new MySubException("input still too low");
      case 99: throw new MyOtherException("input too high");
      default: return i*i;
    }
  }
}
```

```
> java ThrowTest hello
Must specify an integer argument

> java ThrowTest  0
i= 0
MyException:
input too low
Handled at point 1

> java ThrowTest 1
i= 1
MySubException:
input still too low
Handled at point 1

> java ThrowTest 2
i= 2
c(i)= 4

> java ThrowTest 99
i= 99
MyOtherException: input too high
Handled at point 2
```