Teil II

Objektorientierte Programmierung und objektorientierter Entwurf

Kapitel 8

Robuste Programme durch Ausnahmebehandlung

Ziele:

- Lernen, robuste Programme zu schreiben
- Ausnahmen als Objekte verstehen lernen
- Bedeutung von Ausnahmen erkennen in der Signatur und im Rumpf einer Methode
- Lernen, Ausnahmebehandlung durchzuführen

Definition. Ein Programm heißt *robust*, wenn es für jede Eingabe eine wohldefinierte Ausgabe produziert. Siehe auch Beispiel 8.1.

8.1 Fehlerarten

Ein Programm kann aus vielerlei Gründen fehlerhaft sein. Man unterscheidet:

- **Spezifikationsfehler:** Die Spezifikation erfüllt nicht die informellen Anforderungen.
- **Entwurfsfehler:** Der Entwurf entspricht nicht der Spezifikation.
- **Programmierfehler:** Das Programm erfüllt nicht die Spezifikation.

Programmierfehler können auch unterschiedlicher Art sein:

- **Syntaxfehler:** Die kontextfreie Syntax des Programms ist nicht korrekt.
- **Typfehler:** Ein Ausdruck oder eine Anweisung des Programms hat einen falschen Typ.

Beispiel 8.1 Fakultät

Die folgende Implementierung der Fakultät ist nicht robust,

```
int fac(int n)
{
    if (n = 0)
        return 1;
    else
        return n * fac(n - 1);
}
```

da sie für $n < 0$ nicht terminiert. Die Implementierung

```
int fac1(int n)
{
    if (n <= 0)
        return 1;
    else
        return n * fac(n-1);
}
```

ist robust, liefert aber „überraschende“ Werte für $n < 0$, die den informellen Anforderungen nicht entsprechen. Besser ist es, hier Ausnahmen einzuführen.

- **Laufzeitfehler:** Ein Fehler, der während der Ausführung eines korrekt übersetzten Programms auftritt, wie z.B. Division durch Null, fehlende Datei oder Netzwerkfehler.

Bemerkung: Syntaxfehler und Typfehler werden zur Übersetzungszeit erkannt. Laufzeitfehler werden in Java durch das Laufzeitsystem dem Benutzer gemeldet. Üblicherweise terminiert ein Java-Programm „unnormal“ beim Auftreten eines Laufzeitfehlers. Darüberhinaus bietet Java eine benutzerdefinierte Behandlung von Laufzeitfehlern an, was zur Robustheit von Java-Programmen beiträgt.

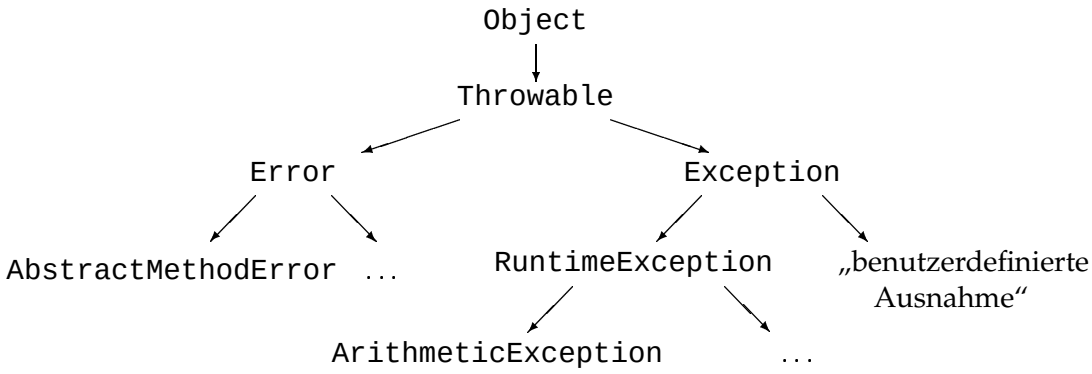
8.2 Ausnahmen und Fehler sind Objekte in Java

In Java sind auch (Laufzeit-)Fehler *Objekte*. Man unterscheidet zwischen

- Fehlern (Instanzen der Klasse `Error`)
- Ausnahmen (Instanzen der Klasse `Exception`)

Ausnahmen können vom Programmierer im Programm durch Ausnahmebehandlung abgefangen werden (und sind vom Programmierer definierbar), Fehler deuten auf schwerwiegende Probleme hin und sollten nie behandelt werden.

Java besitzt folgende Vererbungshierarchie von Fehlerklassen:



`AbstractMethodError` gibt an, daß eine abstrakte Methode aufgerufen wurde. `ArithmeticException` gibt einen arithmetischen Fehelr an, wie Division durch Null.

`Throwable` ist die Standardklasse für Ausnahmen (Laufzeitfehler).

Konstruktoren: `Throwable()`, `Throwable(String message)` konstruieren Fehlerobjekte, eventuell mit einer speziellen Nachricht.

Ein Fehlerobjekt enthält:

- einen Schnappschuß des Aufrufkellers zum Zeitpunkt der Erzeugung des Objekts
- eine Meldung zur Beschreibung des Fehlers

weitere Methoden:

- `String getMessage()`: gibt die Fehlermeldung zurück
- **`void`** `printStackTrace()`: gibt den momentanen Stand des Aufrufkellers aus

Ausnahmeobjekte werden vom Java-Laufzeitsystem automatisch erzeugt, wenn eine Fehlersituation auftritt.

Die folgenden zwei Klassen `Exc0` und `Exc1` illustrieren Laufzeitfehler der Klasse `ArithmeticException`. Sie sind in Beispiel 8.2 und 8.3 zu finden.

In der Klasse `Exc0` wird bei Ausführung der Division durch 0 die Ausführung des Codes gestoppt und ein Ausnahmeobjekt (der Klasse `ArithmeticException`) erzeugt.

Das Laufzeitsystem schleudert („throws“) die Ausnahme. (Man denke an eine „heiße Kartoffel“, die der Code abfangen muß.) Der Kontrollfluß wird unterbrochen und der aktuelle Aufrufkeller wird auf Ausnahmebehandlungsmöglichkeiten durchsucht.

In diesem Fall läuft die Standardausnahmebehandlung, die den `String`-Wert der Ausnahme (2. Zeile) und den Aufrufkeller (3. Zeile) ausgibt:

Beispiel 8.2 Die Klasse `Exc0`

```
/**
    Diese Klasse illustriert das Ausloesen einer Ausnahme.
    Bei der Division durch 0 wird eine ArithmeticException
    ausgeloest.
*/
public class Exc0
{
    /**
        Die Methode main loest wegen der Division durch 0
        eine ArithmeticException aus:
        "Exception in thread "main"
        java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:13)"
        d.h. der aktuelle Aufrufkeller enthaelt nur Exc.main
    */
    public static void main(String args[])
    {
        int d = 0;
        int a = 42/d;

        System.out.println("d = " + d); // nicht gedruckt
        System.out.println("a = " + a); // wegen der vorherigen Ausnahme
    }
}
```

Beispiel 8.3 Die Klasse `Exc1`

```
/**
    Diese Klasse illustriert - aehnlich wie Exc0 - das
    Ausloesen einer Ausnahme.
    Bei der Division durch 0 wird eine ArithmeticException
    ausgeloest. Anders als bei Exc0 wird die Ausnahme in der
    Methode subroutine augeloest, die in main aufgerufen wird.
*/
public class Exc1
{
    /**
        Die Methode subroutine loest wegen der Division durch 0
        eine ArithmeticException aus mit aktuellem Aufrufkeller:
        " at Exc1.subroutine(Exc1.java:19)
          at Exc1.main(Exc1.java:29)
    */
    public static void subroutine()
    {
        int d = 0;
        int a = 42/d;

        System.out.println("d =" + d);
        System.out.println("a =" + a);
    }

    /**
        Die Methode main ruft die Methode subroutine() auf, in der
        eine ArithmeticException ausgeloest wird.
    */
    public static void main(String args[])
    {
        Exc1.subroutine();  // Optionale zusaetzliche
                            // Angabe der Klasse Exc1
    }
}
```

```
> java Exc0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:19)
```

Die Klasse `Exc1` ist ähnlich zu `Exc0` mit dem Unterschied, daß die Division durch Null in der Methode `subroutine` auftritt. Bei der Fehlermeldung gibt das Java-Laufzeitsystem den zu diesem Zeitpunkt existierenden Laufzeitkeller an, der die beiden noch nicht beendeten Methodenaufrufe von `main` und `subroutine` enthält. Die Klasse ergibt folgende Ausgabe:

```
> java Exc1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exc1.subroutine(Exc1.java:19)
        at Exc1.main(Exc1.java:31)
```

8.3 Abfangen von Ausnahmen

Ausnahmebehandlung geschieht in Java mit Hilfe der „**`try`**“-Anweisung, die folgende Grundform hat:

```
try {
    // Block fuer "normalen" Code
}
catch (Exception1 e) {
    // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception1
}
catch (Exception2 e) {
    // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception2
}
finally {
    // Code, der in jedem Fall nach normalem Ende und nach
    // Ausnahmebehandlung ausgefuehrt werden soll.
}
```

In **`try`** wird der normale Code ausgeführt. Tritt eine Ausnahmesituation auf, so wird eine Ausnahme ausgelöst („**`throw`**“), die je nach Typ von einem der beiden Ausnahmebehandler („Handler“) abgefangen („**`catch`**“) wird. Falls die Handler nicht den passenden Typ haben, wird im umfaßenden Block nach einem Handler gesucht. Falls kein benutzerdefinierter Handler gefunden wird, wird die Standardausnahmebehandlung durchgeführt. Das „**`finally`**“-Konstrukt ist optional; darin stehender Code wird auf jeden Fall ausgeführt und zwar nach dem normalen Ende bzw. nach Ende der Ausnahmebehandlung.

Syntax

TryStatement ::=
*"**`try`**" BlockStatement*
*{ "**`catch`**" "(" ExceptionClass Name ")" BlockStatement }**
*["**`finally`**" BlockStatement]*

Mindestens ein **`catch`**- oder **`finally`**-Block muß vorkommen.

Bemerkung: Der Vorteil der „**`catch`**“-Anweisung ist, daß das System nicht mit einem Fehler „anormal“ terminiert, sondern normal weiterarbeiten kann, als ob der Fehler nie vorgekommen sei.

Die Beispielklasse `Exc2` zeigt, wie die Ausnahme bei Division durch 0 abgefangen werden kann. Das Programm arbeitet nach der **`try`**-Anweisung „normal“ weiter und gibt „Hurra!“ auf dem Bildschirm aus.

Die Klasse `Exc3` löst ähnlich wie `Exc1` eine `AritmeticException` aus, führt aber den Block von „**`finally`**“ aus, d.h. „Hallo!“ wird am Bildschirm ausgegeben. Da das Programm aber anormal terminiert, wird „Hurra!“ *nicht* gedruckt.

Die Klasse `Exc4` erweitert `Exc3` um ein Abfangen der Division durch 0. Deshalb wird sowohl „Hallo!“ als auch „Hurra!“ ausgegeben.

8.4 Ausnahmeklassen

Ausnahmeklassen sind *Subklassen von* `Exception` und werden wie normale Klassen mit Attributen und Konstruktoren deklariert. Meist benötigt man nur die Methoden von `Throwable`, die es erlauben, den Aufrufkeller auszugeben und die Ausnahme-Nachricht zu lesen.

Beispiel 8.7 auf Seite 51 zeigt eine Klasse von selbstdefinierten Ausnahmen, die zur Behandlung der Fakultätsmethode verwendet werden kann.

8.5 Auslösen von Ausnahmesituationen

Mittels der „**`throw`**“-Anweisung kann man eine kontrollierte Ausnahme auslösen.

Syntax

ThrowStatement ::=
*"***`throw`***" Expression;*

Der Ausdruck muß eine Instanz einer Subklasse von `Throwable` (d.h. eine Ausnahme oder ein Fehlerobjekt) bezeichnen.

Die Ausführung einer „**`throw`**“-Anweisung stoppt den Kontrollfluß des Programms. Die nächste Anweisung wird nicht mehr ausgeführt. Der umschließende „**`try`**“-Block wird auf die Existenz eines passenden Handlers untersucht. Falls dieser existiert, wird der zugehörige „**`catch`**“-Block ausgeführt. Falls nicht, wird im nächsten umschließenden „**`try`**“-Block gesucht, usw., bis der äußerste Ausnahmehandler erreicht und ausgeführt wird. Falls keine passende Ausnahmebehandlung gefunden wird, wird eine Ausnahme ausgelöst, die zu „unnormaler“ Terminierung führt.

In Java sind kontrolliert ausgelöste Ausnahmen genauso wichtig, wie normale Ergebniswerte. Deshalb wird ihr Typ im Kopf einer Methode angegeben (mit Ausnahme von Subklassen von `Error` und `RuntimeException`). Dies geschieht mittels „**`throws`**“.

Der Kopf einer Methode erhält folgende Form:

Beispiel 8.4 Die Klasse `Exc2`

```
/**
    Diese Klasse loest - aehnlich wie Exc1 - eine
    ArithmeticException aus, faengt diese aber durch "catch" ab.
*/
public class Exc2
{
    /**
        Die Methode loest wegen Division durch 0 eine
        ArithmeticException aus, faengt diese aber durch "catch" ab.
        Anschliessend wird "Hurra!" ausgegeben.
    */
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }
        catch (ArithmeticException e)
        {
            System.out.println("division by zero");
        }

        System.out.println("Hurra!");
    }

    /**
        Die Methode main ruft die Methode subroutine() auf, in der
        eine ArithmeticException ausgeloest wird.
    */
    public static void main(String args[])
    {
        Exc2.subroutine();
    }
}
```

Beispiel 8.5 Die Klasse `Exc3`

```
/**
    Diese Klasse loest - aehnlich wie Exc1 - eine
    ArithmeticException aus, fuehrt aber ausserdem
    ein "finally" aus.
*/
public class Exc3
{
    /**
        Die Methode loest wegen Division durch 0 eine
        ArithmeticException aus, faengt diese nicht ab.
        Aber es wird in "finally" ein Wert ausgegeben.
    */
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }
        finally
        {
            System.out.println("Hallo!");
        }

        System.out.println("Hurra!");
    }

    /**
        Die Methode main ruft die Methode subroutine() auf, in der
        eine ArithmeticException ausgeloest wird.
    */
    public static void main(String args[])
    {
        Exc3.subroutine();
    }
}
```

Beispiel 8.6 Die Klasse `Exc4`

```
/**
    Diese Klasse loest - aehnlich wie Exc3 - eine
    ArithmeticException aus, faengt diese aber ab
    und fuehrt ausserdem ein "finally" aus.
*/
public class Exc4
{
    /**
        Die Methode loest wegen Division durch 0 eine
        ArithmeticException aus, faengt diese ab und
        gibt in "finally" einen Wert aus.
    */
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }
        catch (ArithmeticException e)
        {
            System.out.println("division by zero");
        }
        finally
        {
            System.out.println("Hallo!");
        }

        System.out.println("Hurra!");
    }

    /**
        Die Methode main ruft die Methode subroutine() auf, in der
        eine ArithmeticException ausgeloest wird.
    */
    public static void main(String args[])
    {
        subroutine();
    }
}
```

Beispiel 8.7 Die Klasse `NegativeValueException`

```
public class NegativeValueException extends Exception
{
    String text;

    public NegativeValueException(String text)
    {
        this.text = text;
    }

    public String toString()
    {
        return "NegativeValueException[" + text + "]";
    }
}
```

*Accessmodifier Type MethodName "{" Paramlist "}" ["**throws**" Exceptionlist]*

wobei *Exceptionlist* die Form

Type {"," Type}

hat.

Die Typen der „**throw**“-Anweisung des Rumpfs *müssen* im Kopf der Methode angegeben werden.

Eine robuste Implementierung der Fakultät, die eine Ausnahme auslöst, wenn der aktuelle Parameter negativ ist, findet man in Beispiel 8.8 auf der nächsten Seite.

Bemerkung: Wenn man eine Methode aufruft, die einen Ausnahmetyp in der **throws**-Klausel (im Kopf) enthält, gibt es drei Möglichkeiten:

- Man fängt die Ausnahme mit **catch** ab und behandelt sie, um ein normales Ergebnis zu erhalten.
- Man fängt die Ausnahme mit **catch** ab und bildet sie auf eine Ausnahme (aus dem Kopf) der geeigneten Methode ab.
- Man deklariert die Ausnahme im Kopf der eigenen Methode.

8.6 Zusammenfassung

1. Ausnahmen sind Objekte. Siehe dazu auch Beispiel 8.7.
2. Methoden können Ausnahmen auslösen. Vgl. hierzu Methode `fac` aus Beispiel 8.8 auf der nächsten Seite.

Beispiel 8.8 Robuste Implementierung der Fakultät

```
/**
    Diese Klasse enthaelt eine statische Methode zur Berechnung der
    Fakultaet n!, die fuer negative aktuelle Parameter eine Ausnahme wirft.
*/
public class FacExc
{
    /**
        Diese Methode berechnet n! fuer nichtnegative Zahlen,
        aber terminiert NICHT fuer negative Zahlen.
    */
    public static int fac(int n) throws NegativeValueException
    {
        if (n == 0)
            return 1;
        else if (n > 0)
            return n * fac(n - 1);
        else
            throw new NegativeValueException("at FacExc.fac(" + n + ")");
    }

    /**
        Diese Methode testet FacExc.  Moegliche Ausnahmen werden
        in "try"-Anweisungen eingeschlossen.
    */
    public static void main(String[] args)
    {
        for(int i = -2; i < 5; i++)
        {
            try
            {
                System.out.println("fac(" + i + ") = " + fac(i));
            }
            catch (NegativeValueException e)
            {
                System.out.println(e);
            }
        }
    }
}
```

3. Ausnahmen können mit „**`catch`**“ behandelt werden. Siehe Methode `main` in Beispiel 8.8 auf der vorherigen Seite.
4. Werden Ausnahmen nicht behandelt, müssen sie in der Signatur erscheinen.
5. Robuste Programme terminieren immer – und zwar mit einem wohldefinierten Ergebnis.