8 Ausnahmen

8.1 Grundsätzliches

- Wenn während der Ausführung eines Java-Programms bestimmte semantische Bedingungen verletzt werden (z. B. ganzzahlige Division durch 0, Dereferenzierung einer Nullreferenz, fehlerhafter Indexwert), wirft die Java Virtual Machine eine entsprechende *Ausnahme* (vgl. § 8.3).

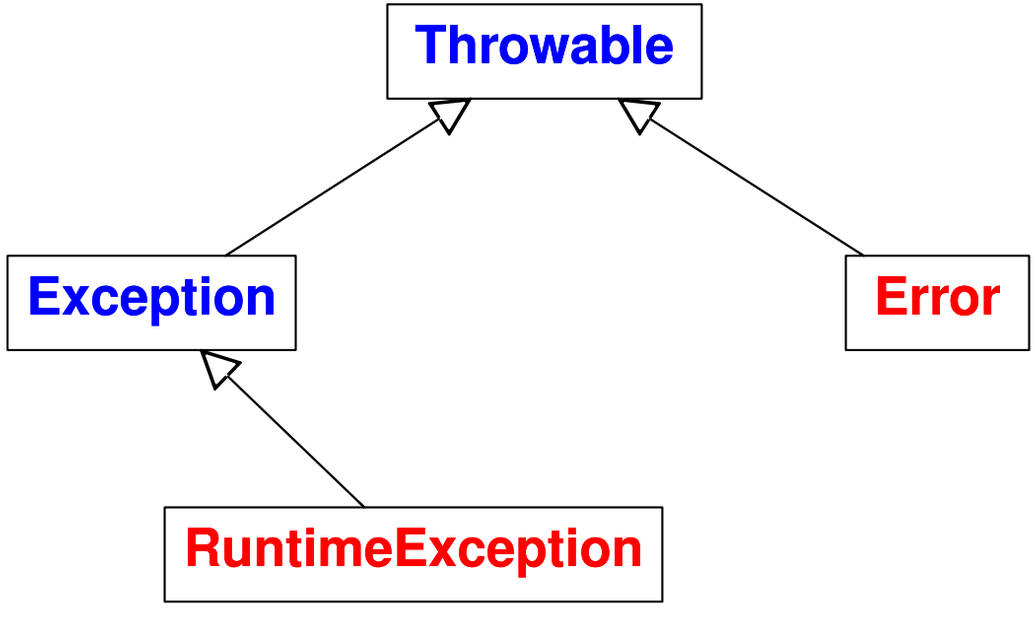
- Ebenso können Ausnahmen explizit mit Hilfe von `throw`-Anweisungen geworfen werden, um semantische Fehler (wie z. B. unzulässige Kontoüberziehungen) zu signalisieren (vgl. § 8.4 und § 8.5).

- Ausnahmen können mit `try`-Anweisungen „aufgefangen“ und behandelt werden, z. B. durch Ausgabe einer Fehlermeldung (vgl. § 8.6).

- Ausnahmen sind Objekte der Klasse `Throwable` oder einer ihrer (direkten oder indirekten) Unterklassen. Sie enthalten mindestens folgende Information:
  - eine Fehlermeldung (z. B. „/ by zero“ oder der konkrete fehlerhafte Indexwert);
  - die aktuelle Aufrufverschachtelung (stack trace).

8.2 Geprüfte und ungeprüfte Ausnahmen

- Es wird zwischen *geprüften* und *ungeprüften* Ausnahmen (checked/unchecked exceptions, in der Abbildung blau/rot) unterschieden:



  Unterklassen von `RuntimeException` und `Error` stellen ungeprüfte Ausnahmen dar, während alle anderen Unterklassen von `Throwable` geprüfte Ausnahmen darstellen.

- Wenn eine Methode oder ein Konstruktor eine geprüfte Ausnahme werfen kann, muss dies im Kopf *deklariert* werden (vgl. § 8.7).

8.3 Vordefinierte Ausnahmeklassen

- Bei der Ausführung bestimmter Operatoren können u. U. die folgenden ungeprüften Ausnahmen auftreten (vgl. § 3.3.4):

Operator	*Ausnahme*
`new`	`OutOfMemoryError`
`new []`	`NegativeArraySizeException` `OutOfMemoryError`
`.`	`NullPointerException`
`[]`	`NullPointerException` `ArrayIndexOutOfBoundsException`
`(type)`	`ClassCastException` (vgl. § 5.8)
`/` `%`	`ArithmeticException`
`=`	`ArrayStoreException` (vgl. § 5.11)

- Die Methode `clone` der Wurzelklasse `Object` kann eine geprüfte Ausnahme des Typs `CloneNotSupportedException` werfen (vgl. § 8.8).

- Darüber hinaus gibt es zahlreiche weitere geprüfte und ungeprüfte Ausnahmen, die von verschiedenen Methoden der Java-Standardbibliothek geworfen werden können.

8.4 Benutzerdefinierte Ausnahmeklassen

- Jede direkte oder indirekte Unterklasse von `Throwable` ist eine Ausnahmeklasse, d. h. ihre Objekte können mittels `throw` als Ausnahmen geworfen werden (vgl. § 8.5).

- Häufig bieten benutzerdefinierte Ausnahmeklassen analog zu `Throwable` zwei öffentliche Konstruktoren an, die jeweils via `super` den entsprechenden Oberklassenkonstruktor aufrufen:
  - einen parameterlosen Konstruktor;
  - einen Konstruktor mit einem Parameter `message` des Typs `String`.

- Die entsprechenden Konstruktoren von `Throwable` speichern in dem erzeugten Objekt einerseits die optionale Fehlermeldung (die mittels `getMessage` abgefragt werden kann) und andererseits die aktuelle Aufrufverschachtelung (die mittels `printStackTrace` ausgegeben werden kann).

- Je nach Anwendung können aber auch Konstruktoren mit anderen oder zusätzlichen Parametern sinnvoll sein.

Beispiel

```
// Geprüfte Ausnahme zur Signalisierung
// von unzulässigen Kontoüberziehungen.
class OverdrawException extends Exception {
  public final LimitedAccount account;  // Betroffenes Konto.
  public final int amount;              // Unzulässiger Betrag.

  // Konstruktor.
  public OverdrawException (LimitedAccount account, int amount) {
    super();
    this.account = account;
    this.amount = amount;
  }

  // Umwandlung in Zeichenkette.
  public String toString () {
    // super.toString() liefert den Namen der Ausnahmeklasse
    // (d. h. hier "OverdrawException").
    return super.toString() + " on account no. " + account.number();
  }
}
```

8.5 Werfen von Ausnahmen

- Die Anweisung `throw` `object` (vgl. § 3.4.3) wirft das Objekt `object` als Ausnahme.
- Dadurch werden nacheinander alle dynamisch umschließenden Anweisungen, Methoden, Konstruktoren etc. *abrupt* beendet, bis die Ausnahme von einer passenden `try`-Anweisung aufgefangen wird (vgl. § 8.6).
- Wenn es keine passende `try`-Anweisung gibt, wird das Programm (bzw. der aktuelle Thread) beendet.
- Der statische Typ von `object` muss ein (trivialer, direkter oder indirekter) Untertyp von `Throwable` sein.
- `object` kann prinzipiell ein beliebiger Ausdruck sein, der ein entsprechendes Objekt als Resultat liefert.
  In der Regel handelt es sich um einen Objekterzeugungsausdruck.
- Beispiel (vgl. § 5.7):

```
if (balance() - amount < -limit) {
  throw new OverdrawException(this, amount);
}
```

8.6 Auffangen von Ausnahmen

8.6.1 Syntax einer `try`-Anweisung

- Eine `try`-Anweisung (vgl. § 3.4.5) besteht aus einer `try`-Klausel mit oder ohne Ressourcen, beliebig vielen `catch`-Klauseln und eventuell einer `finally`-Klausel:

```
try (Resource1 r1 = ...; r2; Resource3 r3 = ...) {
  ......
}
catch (Exception1 e) {
  ......
}
catch (Exception2a | Exception2b e) {
  ......
}
catch (Exception3 e) {
  ......
}
finally {
  ......
}
```

- Es muss mindestens eine `catch`-Klausel oder die `finally`-Klausel oder eine Ressourcenliste vorhanden sein, weil eine `try`-Klausel allein nutzlos ist.
- Die Parametertypen der `catch`-Klauseln müssen Unterklassen von `Throwable` sein. Wenn eine dieser Klassen eine geprüfte Ausnahme bezeichnet, muss die `try`-Klausel tatsächlich eine Ausnahme dieses Typs (oder eines Untertyps) werfen können.
- Wenn eine dieser Klassen eine Unterklasse einer anderen ist, muss ihre `catch`-Klausel vor der `catch`-Klausel für die Oberklasse stehen.
- Wenn bei einer `catch`-Klausel mehrere Typen angegeben sind, ist der Typ des zugehörigen Parameters der speziellste gemeinsame Obertyp dieser Typen.
- Eine Ressourcen-Angabe ist
  - entweder eine Deklaration einer lokalen Variablen (die dann im `try`-Block sichtbar und implizit `final` ist) mit Initialisierungsausdruck
  - oder ein Ausdruck, der eine existierende Variable (Objektvariable, Klassenvariable, lokale Variable oder Parameter) bezeichnet, die (faktisch) unveränderlich ist. (Aus nicht nachvollziehbaren Gründen ist kein beliebiger Ausdruck erlaubt.)
- Die Typen aller Ressourcen müssen Untertypen der Schnittstelle `AutoCloseable` sein und dementsprechend eine parameterlose Methode `close` mit Resultattyp `void` implementieren.

8.6.2 Ausführung einer `try`-Anweisung

- Zunächst wird die `try`-Klausel ausgeführt, das heißt:
  - Wenn keine Ressourcenliste vorhanden ist, wird der `try`-Block ausgeführt.
  - Andernfalls wird die gesamte `try`-Klausel mit Ressourcen gemäß § 8.6.5 ausgeführt.

- Wenn die Ausführung der `try`-Klausel mit einer *Ausnahme* endet, wird der dynamische Typ des geworfenen Ausnahmeobjekts nacheinander mittels `instanceof` mit den Parametertypen der `catch`-Klauseln verglichen und der Block der ersten passenden Klausel ausgeführt; das Ausnahmeobjekt wird zuvor an den Parameter der `catch`-Klausel zugewiesen.
  Wenn keine passende `catch`-Klausel gefunden wird (insbesondere wenn keine `catch`-Klauseln vorhanden sind), wird die Ausnahme weitergeworfen.

- Wenn die Ausführung der `try`-Klausel *normal* endet oder durch eine *Sprunganweisung* (`break`, `continue` oder `return`) vorzeitig beendet wird, wird kein `catch`-Block ausgeführt.

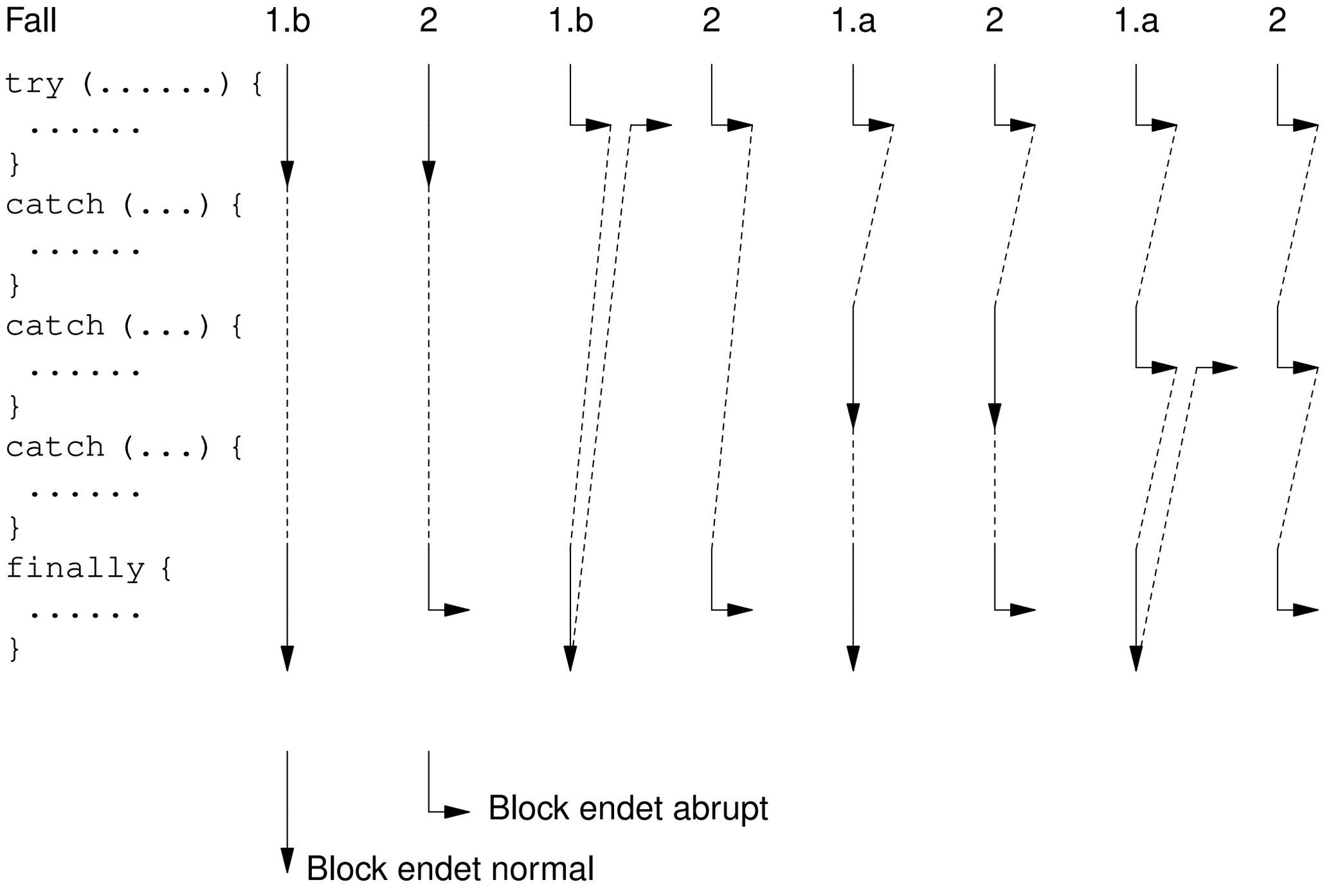
- Nach Ausführung der `try`-Klausel und ggf. eines `catch`-Blocks wird ein vorhandener `finally`-Block unter allen Umständen ausgeführt.

8.6.3 Ausgang einer `try`-Anweisung

❒ Ein Block endet *abrupt*, wenn er eine Ausnahme wirft oder durch eine Sprunganweisung vorzeitig beendet wird. Andernfalls endet der Block *normal*.

Damit lassen sich folgende Fälle unterscheiden:

1. Der `finally`-Block fehlt oder endet normal.
   a) Die `try`-Klausel endete mit einer Ausnahme, zu der es eine passende `catch`-Klausel gibt.
   In diesem Fall endet die gesamte `try`-Anweisung auf dieselbe Weise wie der ausgeführte `catch`-Block. (Insbesondere kann der `catch`-Block die aufgefangene Ausnahme weiterwerfen oder eine andere Ausnahme werfen, die dann von dieser `try`-Anweisung nicht mehr aufgefangen wird).
   b) Die `try`-Klausel endete normal oder vorzeitig durch eine Sprunganweisung oder mit einer Ausnahme ohne passende `catch`-Klausel.
   In diesen Fällen endet die gesamte `try`-Anweisung auf dieselbe Weise wie die `try`-Klausel.
2. Der `finally`-Block endet abrupt.
   In diesem Fall endet die gesamte `try`-Anweisung auf dieselbe Weise abrupt, egal wie die `try`- und eine ggf. ausgeführte `catch`-Klausel endeten. (Insbesondere kann die von der `try`- oder `catch`-Klausel eventuell geworfene Ausnahme verloren gehen.)


Fall
1.b
2
1.b
2
1.a
2
1.a
2
try (......) {
......
}
catch (...) {
......
}
catch (...) {
......
}
catch (...) {
......
}
finally {
......
}
Block endet abrupt
Block endet normal

8.6.4 Beispiel

```
try {
  // charAt wirft StringIndexOutOfBoundsException.
  String s = "abc";
  System.out.println(s.charAt(3));
}
catch (StringIndexOutOfBoundsException e) {
  // Die Aufrufverschachtelung von e wird auf System.out ausgegeben.
  e.printStackTrace(System.out);

  // parseInt wirft NumberFormatException, da msg nicht einfach die
  // Ziffernfolge "3" enthält, sdrn. "String index out of range: 3".
  String msg = e.getMessage();
  int i = Integer.parseInt(msg);
  System.out.println(i);
}
catch (NumberFormatException e) {
  // Dieser catch-Block wird NICHT mehr ausgeführt!
  System.out.println("NumberFormatException");
}

// Die gesamte try-Anweisung endet mit einer NumberFormatException.
```

8.6.5 Ausführung einer `try`-Klausel mit Ressourcen

Eine `try`-Klausel mit Ressourcen der Gestalt `try (.....) { T }` wird ähnlich wie eine Anweisung `try { I; T } finally { C }` ausgeführt:

- Im Teil `I` werden die in der Ressourcenliste angegebenen Ressourcen von links nach rechts initialisiert, das heißt:
  - Wenn eine Ressourcen-Angabe eine lokale Variable deklariert, wird ihr Initialisierungsausdruck ausgewertet.
  - Andernfalls ist die Initialisierung der Ressource leer.
    (Trotzdem ist es für die Ausführung von Teil `C` wichtig, ob die Ressource prinzipiell initialisiert wurde oder nicht.)
- Teil `T` ist der `try`-Block der auszuführenden `try`-Klausel.
- Im Teil `C` wird für jede bereits initialisierte Ressource, die nicht `null` ist, von rechts nach links jeweils die Methode `close` aufgerufen.
- Sobald im Teil `I` oder `T` eine Ausnahme auftritt, endet der Block `{ I; T }` wie gewohnt sofort abrupt mit dieser Ausnahme.
  Wenn im Teil `C` Ausnahmen auftreten, werden diese jedoch aufgefangen und gesammelt, d. h. es werden alle weiteren Aufrufe von `close` ausgeführt.
  Wenn in irgendeinem der drei Teile eine oder mehrere Ausnahmen aufgetreten sind, endet die gesamte `try`-Klausel mit Ressourcen mit der ersten von ihnen, die alle eventuellen weiteren als *unterdrückte* Ausnahmen enthält.

8.6.6 Beispiel

```
class FileCopy {
  // Inhalt der Datei from in die Datei to kopieren.
  public static boolean copy (String from, String to) {
    try (
      // Beide Dateien öffnen und am Ende automatisch schließen.
      var f = new java.io.FileInputStream(from);
      var t = new java.io.FileOutputStream(to);
    ) {
      // Zeichen für Zeichen bis zum Dateiende (-1) kopieren.
      int c;
      while ((c = f.read()) != -1) t.write(c);
    }
    catch (java.io.IOException e) {
      // Alle oben verwendeten Operationen werfen bei Fehlern
      // (z. B. beim Öffnen einer Datei) IOException.
      return false; // Dann Resultatwert false.
    }
    // Sonst Resultatwert true.
    return true;
  }
}
```

8.7 Deklaration von Ausnahmen

- Wenn eine Routine (d. h. eine Methode oder ein Konstruktor) eine geprüfte Ausnahme eines bestimmten Typs *werfen kann*, weil ihr Rumpf
  - eine entsprechende `throw`-Anweisung enthält
  - und/oder selbst eine Routine aufruft, die eine solche Ausnahme werfen kann

  und
  - die Ausnahme nicht von einer `try`-Anweisung aufgefangen wird,

  so muss dies im Kopf der Routine durch eine entsprechende `throws`-Klausel deklariert werden.

- Wenn ein Initialisierungsausdruck einer Objektvariablen oder ein Objektinitialisierer eine geprüfte Ausnahme werfen kann, muss diese Ausnahme von jedem Konstruktor der Klasse deklariert werden.
  (Da der implizit definierte Standardkonstruktor keine Ausnahmen deklariert, muss die Klasse mindestens einen expliziten Konstruktor besitzen.)

- Initialisierungsausdrücke von Klassenvariablen und Klasseninitialisierer dürfen keine geprüften Ausnahmen werfen, weil diese nirgends deklariert werden können.

- Wenn eine Methode eine geerbte Methode überschreibt (vgl. § 5.5), muss ihre `throws`-Klausel logisch eine (echte oder triviale) Teilmenge der `throws`-Klausel der ursprünglichen Methode sein, d. h. es dürfen keine zusätzlichen Ausnahmen deklariert werden.

Beispiel

- Auszug aus der Klasse `LimitedAccount`:

```
// Methode kann OverdrawException direkt via throw werfen.
private void check (int amount) throws OverdrawException {
  if (balance() - amount < -limit) {
    throw new OverdrawException(this, amount);
  }
}

// Methode kann OverdrawException
// indirekt durch Aufruf von check werfen.
public void withdraw (int amount) throws OverdrawException {
  check(amount);
  super.withdraw(amount);
}
```

- Damit die hier gezeigte Überschreibung der Methode `withdraw` in der Klasse `LimitedAccount` korrekt ist, muss bereits die ursprüngliche Methode in der Klasse `Account` mit `throws OverdrawException` deklariert werden, obwohl die Ausnahme von dieser Methode gar nicht geworfen werden kann.

- Begründung: Auch wenn `a` den statischen Typ `Account` besitzt, kann ein Methodenaufruf wie z. B. `a.withdraw(1000)` aufgrund dynamischen Bindens die o. g. Methode `withdraw` der Klasse `LimitedAccount` aufrufen und somit tatsächlich eine Ausnahme des Typs `OverdrawException` werfen.

8.8 Kopieren von Objekten

- Die Wurzelklasse `Object` (vgl. § 5.10) enthält eine Methode

  ```
  protected Object clone () throws CloneNotSupportedException
  ```

  die eine Kopie des aktuellen Objekts erstellt, sofern seine Klasse die Schnittstelle `Cloneable` implementiert; andernfalls erhält man eine geprüfte Ausnahme des Typs `CloneNotSupportedException`.

- `clone` erzeugt eine „flache Kopie“ eines Objekts, d. h. es werden nur die Werte seiner (eigenen und geerbten) Objektvariablen kopiert; referenzierte Objekte werden nicht rekursiv kopiert.

- Damit Klienten die Methode `clone` aufrufen können, überschreibt eine Klasse `C`, die `Cloneable` implementiert, diese Methode in der Regel wie folgt durch eine öffentliche Methode, die die Ausnahme `CloneNotSupportedException` nicht mehr wirft und als Resultattyp `C` statt `Object` besitzen kann:

  ```
  public C clone () {
    try { return (C)super.clone(); }
    catch (CloneNotSupportedException e) { return null; }
  }
  ```

- Gegebenenfalls kann das von `super.clone` erzeugte Objekt noch verändert werden, z. B. indem referenzierte Objekte rekursiv kopiert werden (vgl. § 4.13):

```
class List implements Cloneable {
  ......

  public List clone () {
    try {
      List c = (List)super.clone();
      if (tail != null) c.tail = tail.clone();
      return c;
    }
    catch (CloneNotSupportedException e) {
      return null;
    }
  }
}
```

Anmerkungen

- Die Methode `clone` wird nicht in der Schnittstelle `Cloneable`, sondern in der Klasse `Object` deklariert. Das Implementieren der leeren Schnittstelle dient nur als Kennzeichnung, dass `clone` Objekte der implementierenden Klasse kopieren darf.
- Wenn ein Objekt mittels `clone` erzeugt wird, finden keine der in § 4.9 genannten Initialisierungen statt; insbesondere wird kein Konstruktor der Klasse ausgeführt.
- Die Implementierung von `clone` in der Klasse `Object` kann ein Objekt auch dann kopieren, wenn es nicht-öffentliche geerbte Objektvariablen enthält, die man selbst nicht kopieren könnte.
- Auch wenn eine Klasse `Cloneable` implementiert und die von `Object` geerbte Methode `clone` daher zur Laufzeit keine Ausnahme werfen wird, kann der Aufruf `super.clone()` aus Sicht des Übersetzers trotzdem eine Ausnahme des Typs `CloneNotSupportedException` werfen, die abgefangen (oder im Methodenkopf deklariert) werden muss.
  Damit die Methode `clone` aus Sicht des Übersetzers unter allen Umständen einen Resultatwert liefert, muss der `catch`-Block eine Dummy-`return`-Anweisung enthalten.
- Jeder Reihentyp implementiert die Schnittstelle `Cloneable` und überschreibt die Methode `clone` so, dass sie öffentlich ist, keine Ausnahme werfen kann und den Reihentyp als Resultattyp besitzt.

8.9 Abschließende Bemerkungen

- Der Ansatz, Fehlersituationen durch das Werfen bzw. Auffangen von Ausnahmen zu signalisieren bzw. zu behandeln, besitzt wesentliche Vorteile gegenüber anderen Mechanismen (z. B. Rückgabe bestimmter Fehlercodes und entsprechende Überprüfungen von Resultatwerten):
  - Normaler Code und Fehlerbehandlung können sauber getrennt in `try`- bzw. `catch`-Blöcken formuliert werden.
  - Da eine nicht aufgefangene Ausnahme letztlich zu einem Programmabbruch führt, können Fehler nicht einfach übersehen oder ignoriert werden.
  - Da eine Ausnahme, die von einer Routine nicht aufgefangen wird, automatisch zu ihrem Aufrufer weitergeworfen wird, müssen Fehler nicht in jeder Routine überprüft und dann entweder behandelt oder explizit weitergeleitet werden.
- Da eine Routine, die eine Fehlersituation entdeckt, häufig nicht weiß, wie der Fehler geeignet behandelt werden soll (z. B. durch Ausgabe auf `System.out` oder durch Öffnen eines Dialogfensters), ist es sinnvoll, dass sie den Fehler nur durch Werfen einer Ausnahme signalisiert und die Behandlung einer anderen Routine überlässt.
- Das Konzept, dass geprüfte Ausnahmen in einer Routine entweder aufgefangen oder im Kopf deklariert werden müssen, ist zwar theoretisch überzeugend, erweist sich in der Praxis aber häufig als lästig – und führt gelegentlich zur Formulierung leerer `catch`-Blöcke, die Ausnahmen unbemerkt „verschlucken“ können und das Konzept damit konterkarieren.