LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung
Wintersemester 2018/19

Kapitel 12: Ausnahmen und Ausnahmebehandlung

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 16. Januar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- Fehlerquellen in Programmen und bei der Programmausführung verstehen
- Das Java-Konzept der Ausnahmen als Objekte kennenlernen
- Ausnahmen auslösen können
- Ausnahmen behandeln können

Fehlerquellen

Ein Programm kann aus vielen Gründen ein unerwünschtes Verhalten zeigen:

- **Logische Fehler beim Entwurf**, d.h. bei der Modellierung des Problems: Entwurf entspricht nicht den Anforderungen
- **Fehler bei der Umsetzung des Entwurfs in ein Programm**
  - Programm entspricht nicht dem Entwurf (logischer Programmierfehler)
  - Algorithmen falsch implementiert (logischer Programmierfehler)
  - Programm bricht ab wegen vermeidbarem Laufzeitfehler, z.B. Division durch 0, falscher Arrayzugriff, ...

Fehlerquellen (2)

- **Ungenügender Umgang mit außergewöhnlichen Situationen, die nicht vermieden werden können**
  - Fehlerhafte Benutzereingaben, z.B. Datum 31.11.2019
  - Fehlerhafter Dateizugriff, z.B. Datei existiert nicht
  - Abbruch der Netzwerkverbindung,
  - ...

Robuste Programme

- Wir beschäftigen uns **nicht** damit, wie man logische Fehler in einem Programm finden kann.
- Wir konzentrieren uns auf das Erkennen, Vermeiden und Behandeln von Ausnahmesituationen, die zu Laufzeitfehlern führen können.
- Unser Ziel ist es, **robuste Programme** zu schreiben.

Definition

Ein Programm heißt **robust**, falls es auch beim Auftreten von Fehlern reagiert.

Ausnahmesituationen erkennen (1)

Beispiel: Berechnung der Fakultät

```
class MyClass {
 public static int fact(int n) {
  if (n==0) {return 1;}
  else {return n*fact(n-1);}
 }
}

class Test {
 public static void main(String[] args) {
  int m = ...;
  int k = MyClass.fact(m);
  System.out.println("Fakulaet von " + m + " ist " + k);
  }
}
```

- Es kann zu Nichtterminierung kommen
- Diese Ausnahmesituation wird in der Methode `fact` **nicht** erkannt

Ausnahmesituationen erkennen (2)

```
class MyClass {
 public static int fact2(int n) {
  if (n<0) {return -1;}  // Ausnahmefall
  else {if (n==0) {return 1;}
        else {return n*fact2(n-1);}
       }
 }
}

class Test {
 public static void main(String[] args) {
  int m = ...;
  int k = MyClass.fact2(m);
  System.out.println("Fakulaet von " + m + " ist " + k);
  }
}
```

Die Ausnahmesituation `n < 0` wird in der Methode `fact2` zwar erkannt, aber nicht adäquat an den Aufrufer gemeldet.

Ausnahmesituationen erkennen und Ausnahme auslösen

```
class MyClass {
 public static int fact3(int n) {
  if (n<0) {throw new IllegalArgumentException(
              "Eingabe darf nicht kleiner als 0 sein.");}
  if (n==0) {return 1;}
   else {return n*fact3(n-1);}
 }
}

class Test {
 public static void main(String[] args) {
  int m = ...;
  int k = MyClass.fact3(m);
  System.out.println("Fakulaet von " + m + " ist " + k);
  }
}
```

Die Ausnahme wird in der Methode `fact3` erkannt und eine Ausnahme wird erzeugt („geworfen“).

```
...
    if (n<0) {throw new IllegalArgumentException(
                "Eingabe darf nicht kleiner als 0 sein.");}
...
```

- Die Ausnahme enthält eine individuelle Information, die die Fehlersituation beschreibt.
- Wird eine Ausnahme ausgelöst, dann werden die Methode und das Programm abgebrochen und die Fehlermeldung wird auf der Konsole gemeldet.
- Alternativ kann die Ausnahme auch individuell im main-Programm **behandelt** werden (vgl. später).

Vermeiden von Ausnahmen

```
class MyClass {
 public static int fact4(int n) {
  if (n<0) {throw new IllegalArgumentException(
              "Eingabe darf nicht kleiner als 0 sein.");}
  if (n==0) {return 1;} else {return n*fact4(n-1);} } }
class Test {
 public static void main(String[] args) {
  int m = ...;
  if (m >= 0) {int k = MyClass.fact4(m);
        System.out.println("Fakulaet von " + m + " ist " + k);}
                                          }}
```

- Die Ausnahme wird vermieden, indem in der main-Methode vor dem Aufruf von `fact4` getestet wird, ob das Argument größer gleich 0 ist.
- Es kann (diesbezüglich) keine Ausnahme mehr ausgelöst werden.
- Die Methode `fact4` beinhaltet jedoch weiterhin eine Ausnahmeerkennung mit Ausnahmeauslösung, da sie nicht sicher ist, ob sie korrekt aufgerufen wird („defensive Programmierung“).

Fehler- und Ausnahmenklassen in Java

In Java sind Laufzeitfehler ebenfalls Objekte.

Man unterscheidet:

- **Fehler** (Instanzen der Klasse `Error`)
- **Ausnahmen** (Instanzen der Klasse `Exception`)

Fehler (Instanzen der Klasse `Error`)

- Fehler sind schwerwiegende Probleme und sollten nie behandelt werden, z.B. `VirtualMachineError`
- Das Auftreten solcher Fehler sollte, wo möglich, vermieden werden.

Fehler- und Ausnahmenklassen in Java (2)

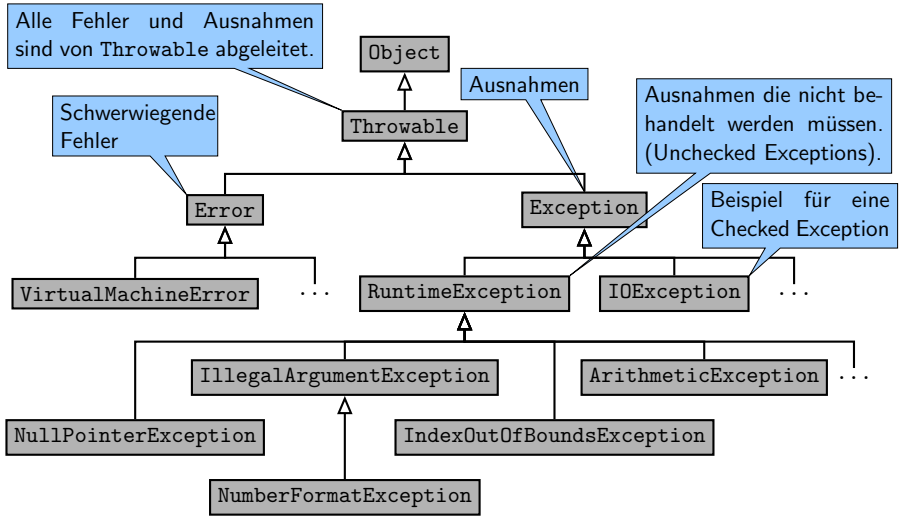
Ausnahmen (Instanzen der Klasse `Exception`)

- **Unchecked Exceptions**: Ausnahmen, die nicht unbedingt vom Programmierer behandelt werden müssen (Instanzen der Klasse `RuntimeException`)
- **Checked Exceptions**: Ausnahmen, die vom Programmierer behandelt werden müssen: (alle anderen Instanzen von `Exception`)

Bemerkungen:

- Ausnahmen können vom Programmierer im Programm durch Ausnahmebehandlung abgefangen werden (und sind vom Programmierer definierbar).
- Ausnahmeobjekte werden vom Java-Laufzeitsystem automatisch erzeugt, wenn eine Fehlersituation auftritt.

Klassenhierarchie der Fehlerklassen



Die Klasse `Error` und ihre direkten Subklassen

java.lang

Class Error

java.lang.Object
  java.lang.Throwable
    java.lang.Error

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationFormatError, AssertionError, AWTError, CoderMalfunctionError, FactoryConfigurationError, FactoryConfigurationError, IOError, LinkageError, SchemaFactoryConfigurationError, ServiceConfigurationError, ThreadDeath, TransformerFactoryConfigurationError, VirtualMachineError

Die Klasse `Exception` und ihre direkten Subklassen

Class Exception

java.lang.Object
  java.lang.Throwable
    java.lang.Exception

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, IOException, JAXBException, JMException, KeySelectorException, LambdaConversionException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Klasse `RuntimeException` und direkten Subklassen

Class RuntimeException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, CompletionException, ConcurrentModificationException, DataBindingException, DateTimeException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MalformedParameterizedTypeException, MalformedParametersException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, ProviderNotFoundException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException

Die Klasse `Throwable`

- Ausnahme- und Fehler-Objekte enthalten Informationen über Ursprung und Ursache des Fehlers.
- Die Klasse `Throwable`, von der alle Fehler- und Ausnahmeklassen abgeleitet sind, verwaltet solche Informationen und stellt zur Verfügung:
  - eine **Nachricht** zur Beschreibung des aufgetretenen Fehlers
  - einen **Schnappschuss** des Aufrufstacks zum Zeitpunkt der Erzeugung des Objekts.
- Nützliche Methoden der Klasse `Throwable`:
  - `String getMessage()`: gibt die Fehlermeldung zurück
  - `void printStackTrace()`: gibt den Aufrufstack des Fehlers aus.

Aufrufstack bei Exception

```
public class Div0 {
/* Die Methode m2 loest wegen der Division durch 0
   eine ArithmeticException aus */
  public static void m2() {
   int d = 0;
   int a = 42 / 0;
   System.out.println("d=" + d + ",a=" + a);
  }
  public static void m1() { m2(); }
  public static void main(String[] args) { m1(); }
}
```

Aufruf mit Ausgabe des Aufrufstacks:

```
> java Div0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Div0.m2(Div0.java:7)
        at Div0.m1(Div0.java:11)
        at Div0.main(Div0.java:14)
```


Kontrolliertes Auslösen von Ausnahmen

- Mittels der `throw`-Anweisung kann man eine Ausnahme auslösen.
- Syntax: `throw exp;`
- Der Ausdruck `exp` muss eine Instanz einer von `Throwable` abgeleiteten Klasse (d.h. eine Ausnahme oder ein Fehlerobjekt) bezeichnen.
- Beispiel: `throw new IllegalArgumentException("...");`
- Die Ausführung einer `throw`-Anweisung stoppt den Kontrollfluss der Methode und löst die von `exp` definierte Ausnahme aus. Die nachfolgenden Anweisungen im Rumpf der Methode werden nicht mehr ausgeführt (wie bei `return`)
- Es kommt zu einem Abbruch des Programms, wenn die Ausnahme nicht in einer übergeordneten Methode abgefangen und behandelt wird.

Geprüfte Ausnahmen (Checked Exceptions)

- Geprüfte Ausnahmen sind in Java alle Instanzen der Klasse `Exception`, die **nicht** Instanzen der Klasse `RuntimeException` sind.
- Kann während der Ausführung einer Methode eine geprüfte Ausnahme auftreten, dann muss diese entweder in der Methode behandelt werden (vlg. später) oder es muss im Methodenkopf mit `throws Ausnahmetyp` explizit darauf hingewiesen werden (wird vom Compiler geprüft).

Checked Exceptions: Beispiel

```
import java.io.IOException;
import java.io.File;
class CheckedExceptionTest {
 public static void m(String filePath) throws IOException {
   File f = new File(filePath);
   if (f.isFile()) {
    System.out.println("File will be deleted");
    f.delete();
   }
   else {throw new IOException(filePath + " is not a file");}

 }
 public static void main(String[] args) throws IOException{
   m("hallo");
 }
}
```

Man **muss** deklarieren, dass in dieser Methode die Ausnahme `IOException` auftreten kann.

Da die `IOException`, die beim Aufruf von `m` auftreten kann, hier nicht behandelt wird, muss die Methode `main` selbst wieder die `throws`-Klausel deklarieren.

Ungeprüfte Ausnahmen (Unchecked Exceptions)

- Ungeprüfte Ausnahmen sind genau die Instanzen von `RuntimeException`
- Beispiele: `ArithmeticException`, `NullPointerException`
- Ungeprüfte Ausnahmen müssen nicht im Methodenkopf explizit deklariert werden (sie können es aber).
- Ungeprüfte Ausnahmen müssen nicht behandelt werden (sie können es aber)

Benutzerdefinierte Ausnahmen

Mittels Vererbung kann man eigene Ausnahmeklassen definieren

Beispiel:

- Klassen `BankKonto` und `SparKonto`.
- Es soll nicht möglich sein, ein `SparKonto` zu überziehen
- Wir definieren eine checked Exception, die beim Versuch, das `SparKonto` zu überziehen, geworfen werden soll.

Benutzerdefinierte Ausnahmen

```
public class KontoUngedecktException extends Exception {
  private double abhebungsbetrag;

  public KontoUngedecktException(String msg, double
      abhebungsbetrag) {
    super(msg); // Konstruktor von Exception nimmt Nachricht
    this.abhebungsbetrag = abhebungsbetrag;
  }

  public double getAbhebungsbetrag() {
    return this.abhebungsbetrag;
  }
}
```

Beispiel für Benutzerdefinierte Ausnahme

```
public class BankKonto {
  ...
  public void abheben(double x) throws KontoUngedecktException {
     this.kontoStand -= x;
  }

public class SparKonto extends BankKonto {
  ...
  public void abheben(double x) throws KontoUngedecktException {
   if (this.getKontoStand() < x) {
     throw new KontoUngedecktException("Sparkonten duerfen nicht
         ueberzogen werden.", x);
   }
   super.abheben(x);
 }
}
```


Beispiel für Benutzerdefinierte Ausnahme (2)

```
public class KontoTest {
 public static void main(String[] args)
                    throws KontoUngedecktException {
    SparKonto k = new SparKonto(400,1.0);
    k.abheben(200);
    k.abheben(100);
    k.abheben(160);
 }
}
```

Ausführung:

```
> java KontoTest
Exception in thread "main" KontoUngedecktException:
Sparkonten duerfen nicht ueberzogen werden.
        at SparKonto.abheben(SparKonto.java:20)
        at KontoTest.main(KontoTest.java:7)
```


Behandlung von Ausnahmen: try und catch

Behandlung von Ausnahmen geschieht in Java mit der `try`-Anweisung. Damit ist es möglich Ausnahmen **abzufangen**

```
try {
 // Block fuer normalen Code
} catch (Exception1 e) {
  /* Ausnahmebehandlung falls Ausnahme vom Typ Exception1
      aufgetreten ist */
} catch (Exception2 e) {
  /* Ausnahmebehandlung falls Ausnahme vom Typ Exception2
      aufgetreten ist */
}
```


- Zunächst wird der `try`-Block normal ausgeführt.
- Tritt im `try`-Block keine Ausnahmesituation auf, so werden die beiden Blöcke zur Ausnahmebehandlung ignoriert.
- Tritt im `try`-Block eine Ausnahmesituation auf, so wird die Berechnung dieses Blocks abgebrochen.
  - Ist die Ausnahme vom Typ `Exception1` oder `Exception2`, so wird der Block nach dem jeweiligen `catch` ausgeführt, und eine lokale Variable `e` für die Exception angelegt.
  - Ansonsten ist die Ausnahme unbehandelt.

Beispiel

```
public class KontoTest2 {
 public static void main(String[] args) {
  SparKonto k = new SparKonto(400,1.0);
  try {k.abheben(200);
  } catch (KontoUngedecktException e) {
   System.out.println(e.getMessage());
   System.out.println("Der Abhebungsbetrag " + e.getAbhebungsbetrag() + " war zu
        hoch.");
  }
  try {k.abheben(100);
  } catch (KontoUngedecktException e) {
   System.out.println(e.getMessage());
   System.out.println("Der Abhebungsbetrag " + e.getAbhebungsbetrag() + " war zu
        hoch.");
  }
  try {k.abheben(160);
  } catch (KontoUngedecktException e) {
   System.out.println(e.getMessage());
   System.out.println("Der Abhebungsbetrag " + e.getAbhebungsbetrag() + " war zu
        hoch.");
  }
 }
}
```


Behandlung von Ausnahmen: `finally`

Manchmal möchte man Code nach der Ausführung des `try`-Blocks ausführen, **unabhängig davon** ob eine Ausnahme aufgetreten ist.

- Beispiel: Schließen einer im `try`-Block geöffneten Datei
- Das kann man mit einem `finally`-Block erreichen, der in jedem Fall nach dem `try`-Block un der Ausnahmebehandlung ausgeführt wird.

```
try {
 // Block fuer normalen Code
} catch (Exception1 e) {
   /* Ausnahmebehandlung falls Ausnahme vom Typ Exception1
        aufgetreten ist */
} catch (Exception2 e) {
   /* Ausnahmebehandlung falls Ausnahme vom Typ Exception2
        aufgetreten ist */
} finally {
   /* Code der in jedem Fall ausgefuehrt werden soll (nach
        normalem, Ende, nach Ausnahmebehandlung und nach
        Ausloesung einer nicht behandelten Ausnahme) */
}
```


Beispiel mit finally

```
public class KontoTest3 {
 public static void main(String[] args) {
  SparKonto k = new SparKonto(100,1.0);
  try {
   k.abheben(160);
  } catch (KontoUngedecktException e) {
   System.out.println(e.getMessage());
   System.out.println("Der Abhebungsbetrag " +
        e.getAbhebungsbetrag() + " war zu hoch.");
  } finally {
   System.out.println("Bitte entnehmen Sie Ihre Karte");
  }
 }
}
```


Beispiel: Ausnahmebehandlung bei einer GUI

```
...
public void actionPerformed(ActionEvent e) {
Object source = e.getSource();
// Pruefe, welcher Knopf gedrueckt wurde
if (source == this.einzahlenButton) {
    // Einlesen des Betrags
     try {
       double betrag = Double.parseDouble(this.betragField.getText());
       // Einzahlen auf Konto
       this.konto.einzahlen(betrag);
       // Ausgabe neuer Kontostand
       this.kontoStandField.setText(
                    Double.toString(this.konto.getKontoStand()));
     }
     catch (NumberFormatException ex) {
       this.kontoStandField.setText("Eingabefehler");
     }
    }
...
```

Zusammenfassung

- Ausnahmen werden in Java durch Objekte dargestellt.
- Methoden können Ausnahmen auslösen implizit durch einen Laufzeitfehler oder explizit mit `throw` und damit „abrupt“ terminieren.
- Ausnahmen können mit `catch` behandelt werden, so dass sie nicht zu einem Abbruch des Programms führen.
- Wir unterscheiden geprüfte und ungeprüfte Ausnahmen.
- Geprüfte Ausnahmen müssen abgefangen werden oder im Kopf der Methode wiederum deklariert werden.
- In jedem Fall ist es am Besten, Ausnahmen zu vermeiden, wenn es in der Macht des Programmierers liegt.
- Defensive Programme sehen auch für vermeidbare Ausnahmesituationen das Werfen von Ausnahmen vor (was dann hoffentlich nie nötig ist).