Grundzüge der Informatik

V11. Ausnahmebehandlung in Java

Prof. Dr. Mira Mezini
FG Softwaretechnik
TU-Darmstadt

Die Rolle der Fehlerbehandlung

- Ausnahme (Exception) - ein Ereignis, das während des Ablaufes eines Programms vorkommt und den normalen Kontrollfluss der Anweisungen stört.
- Als Ausnahme bezeichnet man alle Arten von Problemen,
  - angefangen von schweren Hardwarefehlern, wie Festplatten-Crash
  - bis hin zu einfachen Programmier-fehlern, wie z.B. der Versuch, außerhalb der Grenzen eines Arrays zu lesen.

Die Rolle der Fehlerbehandlung

- Die übliche Reaktion auf nicht abgefangene Ausnahmesituationen: Programmabsturz
- Komplexe Programme ohne jede Reaktion auf Ausnahmen sind nicht akzeptabel.
  - Telekommunikationssysteme
  - Steuerungssysteme, z.B. Raketen, Kernkraftwerke

Klassifikation von Fehlern

- schwerwiegende Fehler (Error)
  - Systemfehler, die prinzipiell nicht abgefangen werden können und direkt zum Programmabsturz führen.
    - Fehler im Java-Interpreter
    - Speichermangel
  - Laufzeitfehler, nach denen eine Programmfortsetzung nicht mehr möglich ist
    - Eine benötigte Klasse ist nicht vorhanden
    - Versuch, eine abstrakte Klasse oder ein Interface zu instantiieren

Klassifikation von Fehlern

- leichte Fehler (Ausnahme / Exception): Fehler, die innerhalb des Programms abgefangen werden können. Programmfortsetzung ist möglich.
  - Problem: Eingabe eines fehlerhaften Dateinamens durch Benutzer
    Behebung: Neueingabe verlangen
  - Problem: Fehlerhafte Daten in einer Datei, die ignoriert werden können, z.B. nicht interpretierbare Bild- oder Audiosignale
    Behebung: Ignorieren
  - Problem: Zusammenbruch einer Netzverbindung
    Behebung: Neuaufbau der Verbindung
  - Problem: Zugriff auf ein Bild, dessen URL (Uniform Resource Locator) im Internet nicht mehr vorhanden ist
    Behebung: ignorieren, wenn das Bild nur der Schönheit dient

Fehlerbehandlung ohne eigenständige Sprachkonstrukte

Sprache ohne eigenständige Fehlerbehandlung

Zwei Ansätze zur Fehlerbehandlung sind möglich:

- *Fehler werden* über unübliche Rückgabewerte von Prozeduren / Funktionen *signalisiert* (z.B. -1) und z.B. über Fallunterscheidungen behandelt
- *Programmabbruch (!)* Den Extremfall stellen Standardprozeduren / Funktionen einer Sprache dar, die überhaupt keine Fehler zurückmelden.
  - Z.B. Zugriffsversuch auf eine nicht vorhandene Datei endet in Pascal mit einem Programmabsturz.

Fehlerbehandlung ohne eigenständige Sprachkonstrukte

Sprache ohne eigenständige Fehlerbehandlung

- Schlußfolgerung: Konflikt zwischen Zuverlässigkeit und Übersichtlichkeit
  - werden Fehler behandelt, so entstehen unübersichtliche Programmstrukturen (z.B. viele Fallunterscheidungen)
  - werden Fehler ignoriert, so ist die Zuverlässigkeit des Programms nicht sichergestellt

Eine Fehlerbehandlung ohne eigenständige Sprachkonstrukte hat sich nicht bewährt! Java stellt solche Sprachkonstrukte zur Verfügung

Eigenständige Fehlerbehandlung in Java

Deklaration von Operationen, die Ausnahmen erzeugen

```
public class FileInputStream
extends InputStream {

  public FileInputStream(String dateiPfad)  {
      . . .
      // öffne die Datei in dateiPfad
      . . .
   }
}
```

Wird vom Übersetzer nicht akzeptiert !

Die Datei existiert möglicherweise gar nicht
→Ausnahme: FileNotFoundException
Der Konstruktor muss das explizit machen

Eigenständige Fehlerbehandlung in Java

Deklaration von Operationen, die Ausnahmen erzeugen

```
public class FileInputStream
extends InputStream {

  public FileInputStream(String dateiPfad)
                throws FileNotFoundException {
      . . .

      // öffne die Datei in dateiPfad

      . . .
   }
}
```

Wird vom Übersetzer noch nicht akzeptiert !

Möglicherweise liegt keine Berechtigung vor, die Datei zu öffnen
→Ausnahme: SecurityException
Die Operation muss das explizit machen

Eigenständige Fehlerbehandlung in Java

Deklaration von Operationen, die Ausnahmen erzeugen

```
public class FileInputStream
extends InputStream {

  public FileInputStream(String dateiPfad)
  throws SecurityException, FileNotFoundException {
      . . .

      // öffne die Datei in dateiPfad

      . . .
   }
}
```


Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

```
public class Bar {
  . . .
  public void dateiAusdruck (String dateiPfad)   {
     . . .
    FileInputStream in
               = new FileInputStream(dateiPfad);

     . . .

   }
}
```

Wird vom Übersetzer nicht akzeptiert !!!

Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

1. Möglichkeit: Die aufrufende Operation behandelt die von aufgerufenen Operationen geworfenen Ausnahmen

```
// public class Bar
   public void dateiAusdruck (String dateiPfad)   {
     . . .
    try {
       FileInputStream in
               = new FileInputStream(dateiPfad);
    }
  }
```

Alle Operationsaufrufe, die Ausnahmen werfen können, werden in einem **try**-Block eingeschlossen. Die aufrufende Operation signalisiert damit ihre Bereitschaft, die Ausnahmen abzufangen und zu behandeln.

Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

1. Möglichkeit: Die aufrufende Operation behandelt die von aufgerufenen Operationen geworfenen Ausnahmen

```
// public class Bar
   public void dateiAusdruck (String dateiPfad)  {
    . . .
    try {
        FileInputStream in
                = new FileInputStream(dateiPfad);
    }
    catch ( SecurityException se ) {
        // Ausnahmebehandlung
    }
  }
```

Das Abfangen und die Behandlung von Ausnahmen erfolgt in einem `catch`-Block.

Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

1. Möglichkeit: Die aufrufende Operation behandelt die von aufgerufenen Operationen geworfenen Ausnahmen

Schlüsselwort

```
catch ( SecurityException se ) {
   // Java-code für die Ausnahmebehandlung}
```

- Jeder `catch`-Block deklariert einen formalen Parameter
  - legt den Ausnahmetyp fest, welchen der `catch`-Block abfängt und behandelt (`SecurityException`)
  - Deklariert eine lokale Variable, die innerhalb des Blocks benutzt wird, um auf das zu behandelte Ausnahme-Objekt zu verweisen (`se`)

Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

1. Möglichkeit: Die aufrufende Operation behandelt die von aufgerufenen Operationen geworfenen Ausnahmen

```
// public class Bar
   public void dateiAusdruck (String dateiPfad)  {
    . . .
    try {
       FileInputStream in
                = new FileInputStream(dateiPfad);
    }
    catch (SecurityException se) {
       // Ausnahmebehandlung
    }
    catch (FileNotFoundException e) {
       // Ausnahmebehandlung
    }
     . . .
  }
```

Verschiedene catchs für verschiedene Ausnahme-Typen

Eigenständige Fehlerbehandlung in Java

Aufruf von Operationen, die Ausnahmen werfen

2. Möglichkeit: Die aufrufende Operation gibt Ausnahmen weiter entlang der Aufrufkette

```
public void dateiAusdruck (String dateiPfad)
throws SecurityException, FileNotFoundException {
    . . .
    FileInputStream in
               = new FileInputStream(dateiPfad);
}
```

Eigenständige Fehlerbehandlung in Java

Das Auslösen einer Ausnahme

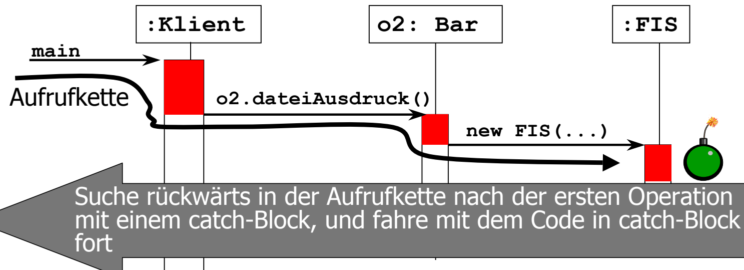
```
public class FileInputStream extends InputStream {
    public FileInputStream(String dateiName)
    throws SecurityException, FileNotFoundException {
       . . .
     /*
       if ( doesNotExist(dateiName) )
         throw new
           FileNotFoundException("File" + dateiName
                       + "doesn't exist");
         if ( noAccessRights(dateiName) )
          throw new
            SecurityException("No access rights");
         öffne die Datei in dateiPfad;
      */
          . . .
     }
}
```

Eigenständige Fehlerbehandlung in Java

- Wenn ein Problem während der Ausführung einer Operation auftritt:
  - erzeugt diese Operation ein Ausnahme-Objekt
  - das Ausnahme-Objekt enthält Information über den Typ des Problems, den Status des Programms, als der Fehler passierte, usw.
  - Eine Ausnahme wird ausgelöst → die Kontrolle und das erzeugte Ausnahme-Objekt wird dem Laufzeitsystem übergeben
  - Das Laufzeitsystem ist dafür zuständig, Code zu finden, der den Fehler behandeln kann.
    - Kandidaten dafür sind Operation in der Aufrufkette der Operation, in der der Fehler auftrat. Die Aufrufkette wird rückwärts gesucht.

Eigenständige Fehlerbehandlung in Java

```
public class Klient {
   public static void main(String[] args) {
      Bar o2 = new Bar();
      o2.dateiAusdruck();
   }
}
```



Vorteile einer eigenständigen Fehlerbehandlung

- Trennung der normalen Verarbeitung von der Fehlerbehandlung
- Weitergabe von Fehlern entlang der dynamischen Aufrufkette
- Unterscheidung und Gruppierung verschiedener Fehlertypen
- Kontrolle durch den Compiler, dass bestimmte Fehlertypen auf jeden Fall behandelt werden

Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

- Angenommen wir haben eine Funktion, die eine ganze Datei von der Festplatte im Hauptspeicher liest. In Pseudo-Code:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```


Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

- Auf dem ersten Blick scheint die Funktion sehr simpel zu sein. Aber sie ignoriert alle möglichen Fehler:
  - Die Datei kann nicht geöffnet werden
  - Die Länge der Datei kann nicht festgestellt werden
  - Nicht genug Platz im Hauptspeicher vorhanden
  - Lesen von der Datei nicht möglich
  - Datei kann nicht geschlossen werden
- Um diese Fälle zu behandeln, müssen wir eine Menge Code hinzufügen, wie die folgende Implementierung zeigt.

Vorteile einer eigenständigen Fehlerbehandlung

```
errorCodeType readFile {
  initialize errorCode = 0;
  open the file;
  if (theFileIsOpen) {
   determine the length of the file;
   if (gotTheFileLength) {
      allocate that much memory;
      if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) { errorCode = -1;
        }
      } else {
        errorCode = -2;
       }
   } else { errorCode = -3; }
   close the file;
   if (theFileDidntClose && errorCode == 0){
    errorCode = -4;
   } else { errorCode = errorCode and -4; }
  }
  else { errorCode = -5; }
  return errorCode;
}
```


Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

- Mit der Fehlerbehandlung bekommen wir 29 anstelle von 7 Zeilen Code - ein Faktor von fast 400%!
- der Ursprungscode geht in dem Code für die Entdeckung, Signalisierung und Behandlung von Fehlern völlig verloren

V11 - 25

Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

- der logische Fluss des Codes ist verloren gegangen, was die Beurteilung, ob der Code das Richtige macht, sehr erschwert
  Wird die Datei tatsächlich geschlossen in dem Fall, dass es nicht genügend Speicherplatz gibt?
- noch schwieriger wird das, wenn die Funktion später modifiziert wird!

V11 - 26

Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

- Java's Ausnahme-Objekte und die eigenständigen Konstrukte für Ausnahmeentdeckung, -Signalisierung und -Behandlung ermöglichen die Trennung des normalen Programmflusses von der Fehlerbehandlung, wie es in der folgenden Folie zu sehen ist.

V11 - 27

Vorteile einer eigenständigen Fehlerbehandlung

Trennung der normalen Verarbeitung von der Fehlerbehandlung

```
void readFile() {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    }
    catch (fileOpenFailed) { doSomething; }
    catch (sizeDeterminationFailed) { doSomething; }
    catch (memoryAllocationFailed) { doSomething; }
    catch (readFailed) { doSomething; }
    catch (fileCloseFailed) { doSomething; }
}
```

V11 - 28

Vorteile einer eigenständigen Fehlerbehandlung

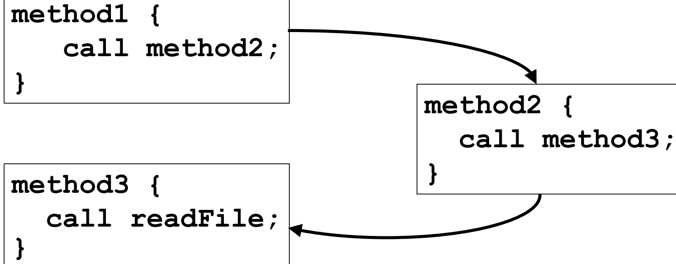
Trennung der normalen Verarbeitung von der Fehlerbehandlung

- Man muss bemerken, dass Ausnahmebehandlung durch eigenständige Sprachkonstrukte die Arbeit zur Ausnahmeentdeckung, Signalisierung und Behandlung nicht erspart.
- Der Vorteil liegt in der Trennung.

Vorteile einer eigenständigen Fehlerbehandlung

Fortpflanzung der Ausnahmen entlang der Aufrufkette

- Angenommen **readFile** ist die vierte Methode in einer Kette von Methodenaufrufen:



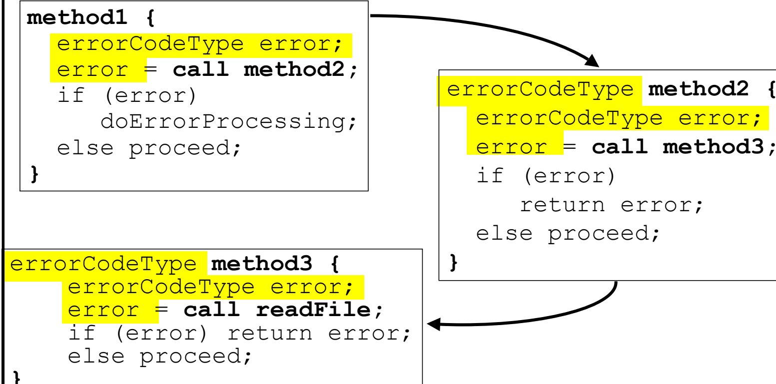
Vorteile einer eigenständigen Fehlerbehandlung

Fortpflanzung der Ausnahmen entlang der Aufrufkette

- Nehmen wir an, nur **method1** ist daran interessiert, die Fehler von **readFile** zu behandeln.
- In traditionellen Sprachen müssen **method2** und **method3** die Fehlerkodierungen, die von **readFile** zurückgegeben werden, weiterleiten bis sie **method1** erreichen

Vorteile einer eigenständigen Fehlerbehandlung

Fortpflanzung der Ausnahmen entlang der Aufrufkette



Vorteile einer eigenständigen Fehlerbehandlung

Fortpflanzung der Ausnahmen entlang der Aufrufkette

Im Gegensatz dazu sucht das Laufzeitsystem von Java rückwärts in der Aufrufkette nach Methoden, die an der Behandlung der Fehler interessiert sind.

```
method1 {
  try {
    call method2;
  } catch (exception) {
    doErrorProcessing;
  }
}
```

```
method2 throws
exception
  { call method3; }
```

```
method3 throws
exception
  { call readFile; }
```

Vorteile einer eigenständigen Fehlerbehandlung

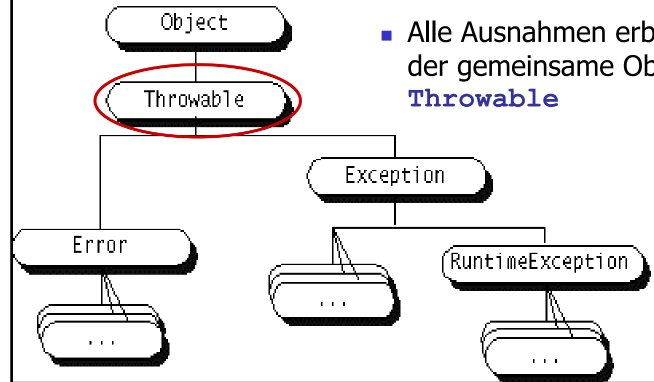
Kontrolle durch den Compiler, dass bestimmte Ausnahmetypen auf alle Fälle behandelt werden

- Eine Operation muss für bestimmte Ausnahmetypen (geprüfte Ausnahmen)
  - Entweder eine Behandlung definieren (catch),
  - oder alle Ausnahmen dieser Typen, die innerhalb des Methoden-Bereichs der Operation vorkommen, weitergeben, indem sie in einem throws-Ausdruck deklariert werden.

Der Bereich einer Operation O ist nicht nur der eigene Code, sondern auch der Code von Operationen, die von O aufgerufen werden. Diese Definition ist rekursiv

Gruppierung von Ausnahmetypen

- Ausnahmen sind ganz normale Java-Objekte, die als solche in Klassen definiert sind



- Alle Ausnahmen erben von der gemeinsame Oberklasse **`Throwable`**

Gruppierung von Ausnahmetypen

Die Klasse *`Throwable`*

`Throwable`

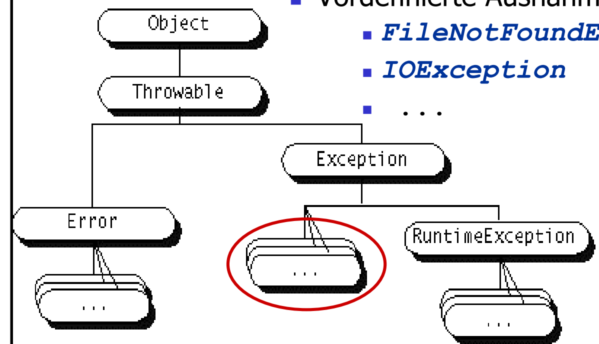
`Throwable()`
`Throwable(String)`
`getMessage()`
`printStackTrace()`
`printStackTrace(PrintStream)`

`...`

- `Throwable()` / `Throwable(String)`: Erzeugt ein *`Throwable`*-Objekt mit einer spezifischen Error-Message
- `getMessage()`: Gibt Error-Message zurück
- `printStackTrace()`: Gibt die Aufrufkette beim Auslösen der Ausnahme aus

Gruppierung von Ausnahmetypen

- Vordefinierte Ausnahmeklassen
  - *`FileNotFoundException`*
  - *`IOException`*
  - ...



Anwendungsspezifische Ausnahmen können als direkte oder indirekte Erben von *Exception* vom Programmierer definiert

Gruppierung von Ausnahmetypen

- Es wird zwischen geprüften und ungeprüften Ausnahmen unterschieden
- Geprüfte Ausnahmen:
  - Der Compiler kontrolliert, dass diese Ausnahmen auf jeden Fall behandelt werden
  - Alle benutzer-definierten Ausnahmen sind geprüfte Ausnahmen
- Ungeprüfte Ausnahmen:
  - Der Compiler erzwingt nicht die Behandlung von solchen Ausnahmen.
  - Deren Auftreten führt zu Programmabsturz

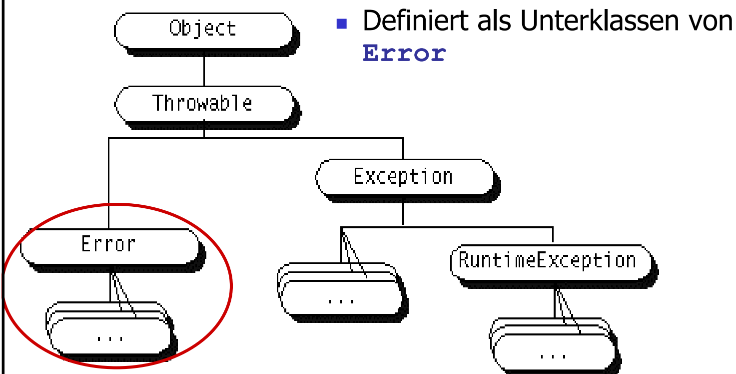
Gruppierung von Ausnahmetypen

Ungeprüfte Ausnahmen → Error

- Error → Schwerwiegende Fehler.
- Das Programm kann nach deren Auftreten nicht mehr fortgeführt werden, z.B. eine Klasse ist nicht vorhanden.
- Es macht keinen Sinn sie abzufangen und zu behandeln: Compiler erzwingt nicht die Behandlung von diesen Fehlern.
- Sie führen zum Programmabsturz

Gruppierung von Ausnahmetypen

Ungeprüfte Ausnahmen → Error



- Definiert als Unterklassen von `Error`

Gruppierung von Ausnahmetypen

Ungeprüfte Ausnahmen → RuntimeException

- Laufzeit Ausnahmen (Runtime Exceptions) sind Fehler, die überall im Programm auftreten könnten und deren Auftreten abhängig von Laufzeitbedingungen ist.
- Beispiele:
  - der Versuch eine Operation an eine Variable mit einer null-Referenz aufzurufen,
  - der Versuch außerhalb der Grenzen eines Array zu lesen / schreiben, usw.

Gruppierung von Ausnahmetypen

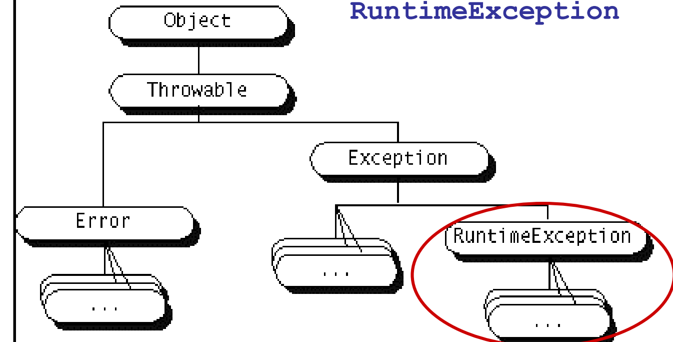
Ungeprüfte Ausnahmen → RuntimeException

- Laufzeit Ausnahmen können, müssen aber nicht abgefangen werden
- Der Grund: Die Behandlung würde das Programm unübersichtlich machen, da solche Fehler überall auftreten können.
  - Wenn NullPointerException z.B. eine geprüfte Ausnahme wäre, wäre ein catch-Block für jeden Operationsaufruf notwendig, auch wenn der Programmierer sicher ist, dass die Variable einen gültigen Verweis auf ein Objekt enthält! Der Compiler kann das aber statisch nicht testen.

Gruppierung von Ausnahmetypen

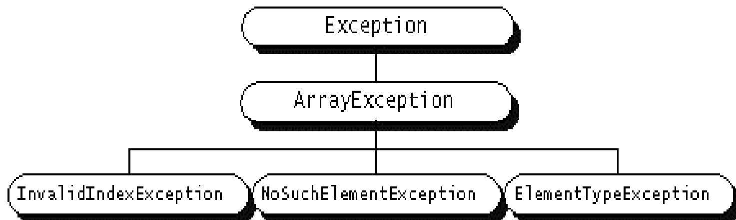
Ungeprüfte Ausnahmen → RuntimeException

- Definiert als Unterklassen von der Klasse **`RuntimeException`**



Gruppierung von Ausnahmetypen

- Man kann Ausnahmetypen durch Vererbung in Generalisierungs-/ Spezialisierungs-relationen zueinander setzen.
- *ArrayException* definiert z.B. das, was alle Ausnahmen, die während der Bearbeitung eines Arrays auftreten können, gemeinsam haben.



Gruppierung von Ausnahmetypen

Verschiedene Operationen können Interesse an die Behandlung von mehr oder weniger spezifischen Ausnahmen anmelden.

```
public void op1() {
    ...
    catch (InvalidIndexException e) {
      // do something with e
    }
    catch (NoSuchElementException e) {
      // do something with e
    }
    catch (ElementTypException e) {
      // do something with e
    }

}
```

Diese Version von *op1* behandelt Ausnahmen bei der Bearbeitung von Arrays unterschiedlich.

Gruppierung von Ausnahmetypen

Verschiedene Operationen können Interesse an die Behandlung von mehr oder weniger spezifischen Ausnahmen anmelden.

```
public void op1() {
   ...
   catch (ArrayException e) {
     // do something with e
   }
}
```

Dagegen werden Probleme bei der Bearbeitung von Arrays in dieser Version von *op1* uniform behandelt.

Gruppierung von Ausnahmetypen

Verschiedene Operationen können Interesse an die Behandlung von mehr oder weniger spezifischen Ausnahmen anmelden.

```
public void op1() {
   ...
   catch (Exception e) {
     // do something with e
   }
}
```

Man kann sogar alle Ausnahmen uniform behandeln. Wird nicht empfohlen!

Gruppierung von Ausnahmetypen

Verschiedene Operationen können Interesse an die Behandlung von mehr oder weniger spezifischen Ausnahmen anmelden.

```
public void op1() {
   ...
   catch (ArrayException e) {
     // do something with e
   }
}
```

Man kann durch Fallunterscheidung auch hier unterschiedlich behandeln, wenn das nötig ist. In diesem Fall ist aber die erste Version besser:

- Code ist besser dokumentiert
- Code ist leichter zu warten

Gruppierung von Ausnahmetypen

- Das Laufzeitsystem wählt den ersten catch-Block, der den Typ der ausgelösten Ausnahme oder ein Supertyp davon behandelt.
- Daher ist die Reihenfolge der catch-Blocks wichtig

```
public void op1() {
   ...
   catch (ArrayException e) {
     // do something with e
   }
   catch (InvalidIndexException e) {
     // do something with e
   }
}
```

Die spezielle Behandlung von Ausnahmen des Typs `InvalidIndexException` wird hier ignoriert ! Die umgekehrte Reihenfolge wäre korrekt.

Ausnahmebehandlung: Beispiel

```
public class ListOfNumbers {

   private Vector vector;
   private static final int size = 10;

   public ListOfNumbers () {
     vector = new Vector(size);
     for (int i = 0; i < size; i++)
       vector.addElement( new Integer(i) );
   }

   ...
```

:ListOfNumbers

0	1	2	...	9

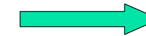
Ausnahmebehandlung: Beispiel

```
// ListOfNumbers

  public void writeList() {
     PrintWriter out = open("OutFile.txt");
     for (int i = 0; i < size; i++)
         out.println("Value at: " + i +
         " = "    + vector.elementAt(i));
     out.close();
  }
}
```

:ListOfNumbers

0	1	2	...	9



OutFile.txt

0
1
...
9

Ausnahmebehandlung: Beispiel

- Diese Klasse wird nicht vom Compiler akzeptiert. Die Methode *writeList* ruft zwei Methoden auf, die Ausnahmen auslösen könnten.
  - Die Methode *open* löst eine *IOException* aus, wenn die Datei nicht geöffnet werden kann: *out = open("OutFile.txt");*
  - Die Methode *elementAt* der Klasse *Vector* löst eine *ArrayIndexOutOfBoundsException* aus, wenn der Parameterwert größer ist als die Länge des Vectors: *out.println("Value at: " + i + " = " + vector.elementAt(i));*

Ausnahmebehandlung: Beispiel

- Der Compiler gibt eine Fehlermeldung nur für den ersten Fall.
  - Der Grund: *IOException* ist eine geprüfte Ausnahme
- *ArrayIndexOutOfBoundsException* eine Runtime-Ausnahme (nicht geprüft) ist.

Ausnahmebehandlung: Beispiel

Der *try*-Block

```
public void writeList() {
   PrintWriter out = null;

    try {
     System.out.println("Entering try block");
     out = open("OutFile.txt"));

     for (int i = 0; i < size; i++)
           out.println("Value at: " +
           i + " = " + vector.elementAt(i));
    }
}
```


Ausnahmebehandlung: Beispiel

Der *catch*-Teil

```
public void writeList() {
  PrintWriter out = null;
  try {
     . . .
  }
  catch (ArrayIndexOutOfBoundsException e) {
     System.err.println( "Caught
        ArrayIndexOutOfBoundsException: "
         + e.getMessage());
  } catch (IOException e) {
    System.err.println("Caught IOException: "
    + e.getMessage());
  }
}
```


Ausnahmebehandlung: Beispiel

Der *finally*-Teil

```
public void writeList() {
  PrintWriter out = null;
  try {
  . . .
  }
  catch (ArrayIndexOutOfBoundsException e) {
    ...
  } catch (IOException e) { ... }
  finally {
  ...
  }
}
```


Ausnahmebehandlung: Beispiel

Der *finally*-Teil

- Der *finally*-Block wird immer ausgeführt:
  - nach einem *try*-Block während dessen Ausführung keine Ausnahme vorkam,
  - nachdem, im Falle einer Ausnahme, alle relevanten *catch*-Blocks abgehandelt wurden.
- Dort wird in der Regel “sauber“ gemacht, bevor das Programm endet. Z.B. werden dort i.d.R. noch geöffnete Datenquellen, wie z.B. Dateien, Netzverbindungen, usw., geschlossen.

Ausnahmebehandlung: Beispiel

Der *finally*-Teil

- Idee: Bei Programmen mit Ausnahmebehandlung gibt es mehrere Möglichkeiten, das Programm zu verlassen.
- In *finally* werden Berechnungen ausgeführt, die unabhängig davon sind, welchem Kontrollfluss der Programmablauf gefolgt ist.

Ausnahmebehandlung: Beispiel

Der *finally*-Teil

```
public void writeList() {
  PrintWriter out = null;
  try { . . . }
  catch (ArrayIndexOutOfBoundsException e) {
    ...
  } catch (IOException e) { ... }
  finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else
         System.out.println("PrintWriter not open");
  }
}
```

Hier wird *out* geschlossen unabhängig von dem konkreten Kontrollfluss des Programms.

Ausnahmebehandlung: Beispiel

Der *finally*-Teil → Vorteile

- Ohne *finally* -Block müsste Code, der unbedingt ausgeführt werden soll, bevor das Programm verlassen wird (Schließen von noch geöffneten Datenquellen, wie z.B. Dateien, Netzverbindungen, usw.), im *try*- und allen *catch*-Blocks dupliziert werden.
- Es ist trotzdem nicht sicher, ob alle Datenquellen geschlossen wurden, da ungeprüfte Ausnahmen auftreten können.

Ausnahmebehandlung: Beispiel

Der *finally*-Teil → Vorteile

Code Duplizierung

```
public void writeList() {
   PrintWriter out = null;
   try { . . .
     out.close();
   } catch (ArrayIndexOutOfBoundsException e) {
     out.close();
     System.err.println("Caught
          ArrayIndexOutOfBoundsException: " +
                                       e.getMessage());
   } catch (IOException e) {
        System.err.println("Caught IOException: "
          + e.getMessage());
   }
}
```

Ausnahmebehandlung: Beispiel

Der *finally*-Teil → Vorteile

```
finally {
  if (out != null) {
   System.out.println("Closing PrintWriter");
   out.close();
  } else {
   System.out.println("PrintWriter not open");
 }
}
```

out wird geschlossen, auch wenn *readList* *ArrayIndexOutOfBoundsException* nicht behandeln würde !

Ausnahmebehandlung: Beispiel

- Alles zusammen:

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering try statement");
        out = new PrintWriter( new FileWriter("OutFile.txt"));
        for (int i = 0; i < size; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: " +
                           e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```


Ausnahmebehandlung: Beispiel

- **Scenario 1: IOException wird ausgelöst**

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

- **Scenario 2: ArrayIndexOutofBoundsException wird ausgelöst**

```
Entering try statement
Caught ArrayIndexOutOfBoundsException: 10>=10
Closing PrintWriter
```

- **Scenario 3: Der try-Block endet normal**

```
Entering try statement
Closing PrintWriter
```


Ablauf einer Ausnahmebehandlung