Script **generated by TTT**

Title: Westermann: Informatik_1 (18.01.2012)

Date: Wed Jan 18 14:28:18 CET 2012

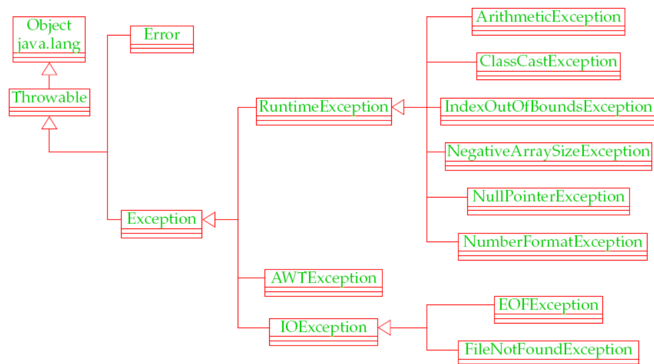
Duration: 62:49 min

Pages: 39

1 Fehler-Objekte: Werfen, Fangen, Behandeln

- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehler-Objekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehler-Objekt kann gefangen und geeignet behandelt werden.

Einige der vordefinierten Fehler-Klassen:



Die direkten Unterklassen von `Throwable` sind:

- `Error` – für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` – für bewältigbare Fehler oder Ausnahmen.

Ausnahmen der Klasse `Exception`, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden !!!

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programm-Ausführung ab.

Beispiel:

```
public class Zero {
   public static main(String[] args) {
      int x = 10;
      int y = 0;
      System.out.println(x/y);
   }  // end of main()
}     // end of class Zero
```


Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: “`/ by zero`”.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Rekursions-Stack.

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: “`/ by zero`”.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Rekursions-Stack.

8

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: “`/ by zero`”.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Rekursions-Stack.

8

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: `NumberFormatException`

```
import java.io.*;
public class Adding {
   private static BufferedReader stdin = new BufferedReader
         (new InputStreamReader(System.in));
   public static void main(String[] args) {
      int x = getInt("1. Zahl:\t");
      int y = getInt("2. Zahl:\t");
      System.out.println("Summe:\t\t"+ (x+y));
   }  // end of main()
   public static int getInt(String str) {
      ...
```


- Das Programm liest zwei `int`-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - ... weil keine syntaktisch korrekte Zahl eingegeben wird;
  - ... weil sonstige unvorhersehbare Ereignisse eintreffen
- Die Behandlung dieser Fehler ist in der Funktion `getInt()` verborgen ...

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: `NumberFormatException`

```
import java.io.*;
public class Adding {
   private static BufferedReader stdin = new BufferedReader
         (new InputStreamReader(System.in));
   public static void main(String[] args) {
      int x = getInt("1. Zahl:\t");
      int y = getInt("2. Zahl:\t");
      System.out.println("Summe:\t\t"+ (x+y));
   }  // end of main()
   public static int getInt(String str) {
      ...
```

9

```
      while (true) {
         System.out.print(str);
         System.out.flush();
         try {
            return Integer.parseInt(stdin.readLine());
         } catch (NumberFormatException e) {
            System.out.println("Falsche Eingabe! ...");
         } catch (IOException e) {
            System.out.println("Eingabeproblem: Ende ...");
            System.exit(0);
         }
      }  // end of while
   }     // end of getInt()
}        // end of class Adding
```

```
      while (true) {
         System.out.print(str);
         System.out.flush();
         try {
            return Integer.parseInt(stdin.readLine());
         } catch (NumberFormatException e) {
            System.out.println("Falsche Eingabe! ...");
         } catch (IOException e) {
            System.out.println("Eingabeproblem: Ende ...");
            System.exit(0);
         }
      }  // end of while
   }     // end of getInt()
}        // end of class Adding
```



```
      while (true) {
         System.out.print(str);
         System.out.flush();
         try {
            return Integer.parseInt(stdin.readLine());
         } catch (NumberFormatException e) {
            System.out.println("Falsche Eingabe! ...");
         } catch (IOException e) {
            System.out.println("Eingabeproblem: Ende ...");
            System.exit(0);
         }
      }  // end of while
   }     // end of getInt()
}        // end of class Adding
```

11

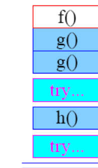
- Jede `catch`-Regel ist von der Form: `catch (Exc e) {...}`
  wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehler-Objekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

- Ein Exception-Handler besteht aus einem `try`-Block
  `try` { ss }
  in dem der Fehler möglicherweise auftritt gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

Die direkten Unterklassen von `Throwable` sind:

- `Error` – für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` – für bewältigbare Fehler oder Ausnahmen.

Ausnahmen der Klasse `Exception`, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden !!!



- Ein Exception-Handler besteht aus einem `try`-Block
  `try {` ss `}`
  in dem der Fehler möglicherweise auftritt gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge ss kein Fehler-Objekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

- Jede `catch`-Regel ist von der Form: `catch (Exc e) {...}`
  wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehler-Objekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

Beispiel: `NullPointerException`

```
public class Kill {
   public static void kill() {
      Object x = null; x.hashCode ();
   }
   public static void main(String[] args) {
      try { kill();
      } catch (ClassCastException b) {
         System.out.println("Falsche Klasse!!!");
      } finally {
         System.out.println("Leider nix gefangen ...");
      }
   }  // end of main()
}     // end of class Kill
```


- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden !
- Die Klasse `Exception` verfügt über die Konstruktoren `public Exception();  public Exception(String str);` (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc   löst den Fehler exc aus – sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen
- Ausgabe:
  ```
  Killed It!
  Killed
  Null
  ```


- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden !
- Die Klasse `Exception` verfügt über die Konstruktoren `public Exception();  public Exception(String str);` (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc   löst den Fehler exc aus – sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen
- Ausgabe:
  ```
  Killed It!
  Killed
  Null
  ```


- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden !
- Die Klasse `Exception` verfügt über die Konstruktoren `public Exception();  public Exception(String str);` (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc   löst den Fehler exc aus – sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen
- Ausgabe:
  ```
  Killed It!
  Killed
  Null
  ```

```
  public static void main(String[] args) {
    try {
      kill();
    } catch (RuntimeException r) {
      System.out.println("RunTimeException "+ r +"\n");
    } catch (Killed b) {
      System.out.println("Killed It!");
      System.out.println(b);
      System.out.println(b.getMessage());
    }
  } // end of main
}   // end of class Kill
```


Fazit:

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Die vordefinierten Fehlerarten reichen oft aus.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Warnung:

- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Fehler sollten dort behandelt werden, wo sie auftreten
- Es ist besser spezifischere Fehler zu fangen als allgemeine – z.B. mit `catch (Exception e) { }`


```
  public static void main(String[] args) {
    try {
      kill();
    } catch (RuntimeException r) {
      System.out.println("RunTimeException "+ r +"\n");
    } catch (Killed b) {
      System.out.println("Killed It!");
      System.out.println(b);
      System.out.println(b.getMessage());
    }
  } // end of main
}   // end of class Kill
```

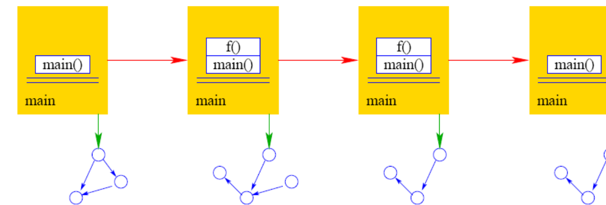
Warnung:

- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Fehler sollten dort behandelt werden, wo sie auftreten
- Es ist besser spezifischere Fehler zu fangen als allgemeine – z.B. mit `catch (Exception e) { }`

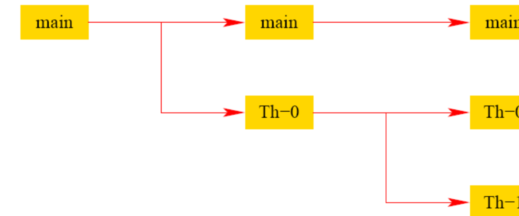

```
    public static void main(String[] args) {
        try {
            kill();
        } catch (RuntimeException r) {
            System.out.println("RunTimeException "+ r +"\n");
        } catch (Killed b) {
            System.out.println("Killed It!");
            System.out.println(b);
            System.out.println(b.getMessage());
        }
    } // end of main
}   // end of class Kill
```


2 Threads

- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungs-Strang.
- Der Aufruf eines Programms startet einen Thread `main`, der die Methode `main()` des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programm-Ausführung zur Verfügung stellen.



- Mehrere Threads sind auch nützlich, um
  - ... mehrere Eingabe-Quellen zu überwachen (z.B. Mouse-Klicks und Tastatur-Eingaben) ↑Graphik;
  - ... während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ... die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt Java die Klasse `Thread` und das Interface `Runnable` bereit.



Beispiel:

```
public class MyThread extends Thread {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyThread
```


Beispiel:

```
public class MyThread extends Thread {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyThread
```

33

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- Ist t ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für t wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Beispiel:

```
public class MyThread extends Thread {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyThread
```


- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode `public void run();` implementieren.
- Ist t ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für t wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Beispiel:

```
public class MyRunnable implements Runnable {
   public void hello(String s) {
      System.out.println(s);
   }
   public void run() {
      hello("I'm running ...");
   }  // end of run()
   public static void main(String[] args) {
      Thread t = new Thread(new MyRunnable());
      t.start();
      System.out.println("Thread has been started ...");
   }  // end of main()
}     // end of class MyRunnable
```