9. Exceptions

Harald Gall, Prof. Dr.

Institut für Informatik
Universität Zürich

http://seal.ifi.uzh.ch/info1

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Lernziele

- Reagieren auf *Unerwartetes*
- Was ist eine Exception?
- Exceptions werfen und abfangen
- Fehlerbehandlung mit Exceptions

Erwartetes & Unerwartetes

- Wir erwarten einen Erfolg für
  - `int numberInStock = kb.nextInt();`
- Aber wir erwarten nicht, dass
  - ein File auf einer Diskette repräsentiert, die irrtümlicherweise entfernt worden ist
  - eine Netzwerk-Verbindung repräsentiert, die plötzlich ausgefallen ist
  - ein File auf einer Festplatte repräsentiert, die wegen Defekts ausgefallen ist

Mehr Unerwartetes vom Anwender

- Wenn wir eine Zahl als Input erwarten, aber wenn
  - ein „w“ statt einer „2“ getippt wird...
  - 21-mal eine „3“ getippt wird...
- Der Kontext, in dem Software ausgeführt wird, ist nicht so vorhersehbar wie wir es uns wünschen.
- Gute Software ist so designed, dass sie diese unerwarteten Ereignisse und Eingaben berücksichtigt und korrekt darauf reagiert.
- Reagieren auf *Unerwartetes* ist mindestens so essentiell wie die Problemstellung per se.

Reagieren auf Unerwartetes

- Beispiel: `parseInt()` mit Argument String; was wenn?

```
System.out.print("Hi, parseInt here. I have a" +
                        "bad numeric argument");
System.out.print( arguments go here );
System.out.print("Please type in a proper value: ");
... = ...readLine();  // Read new String from keyboard
```

  - kein User, da File-Verarbeitung oder via Netzwerk-Verbindung
  - `parseInt()` hat keine Information über den Kontext!!!
  - Es ist nicht die Verantwortlichkeit von `parseInt()` den Fehler zu korrigieren oder dessen ultimative Quelle herauszufinden!
  - Verantwortlich ist der Aufrufer von `parseInt()` - `parseInt()` muss nur seinen Aufrufer über das aufgetretene Problem informieren

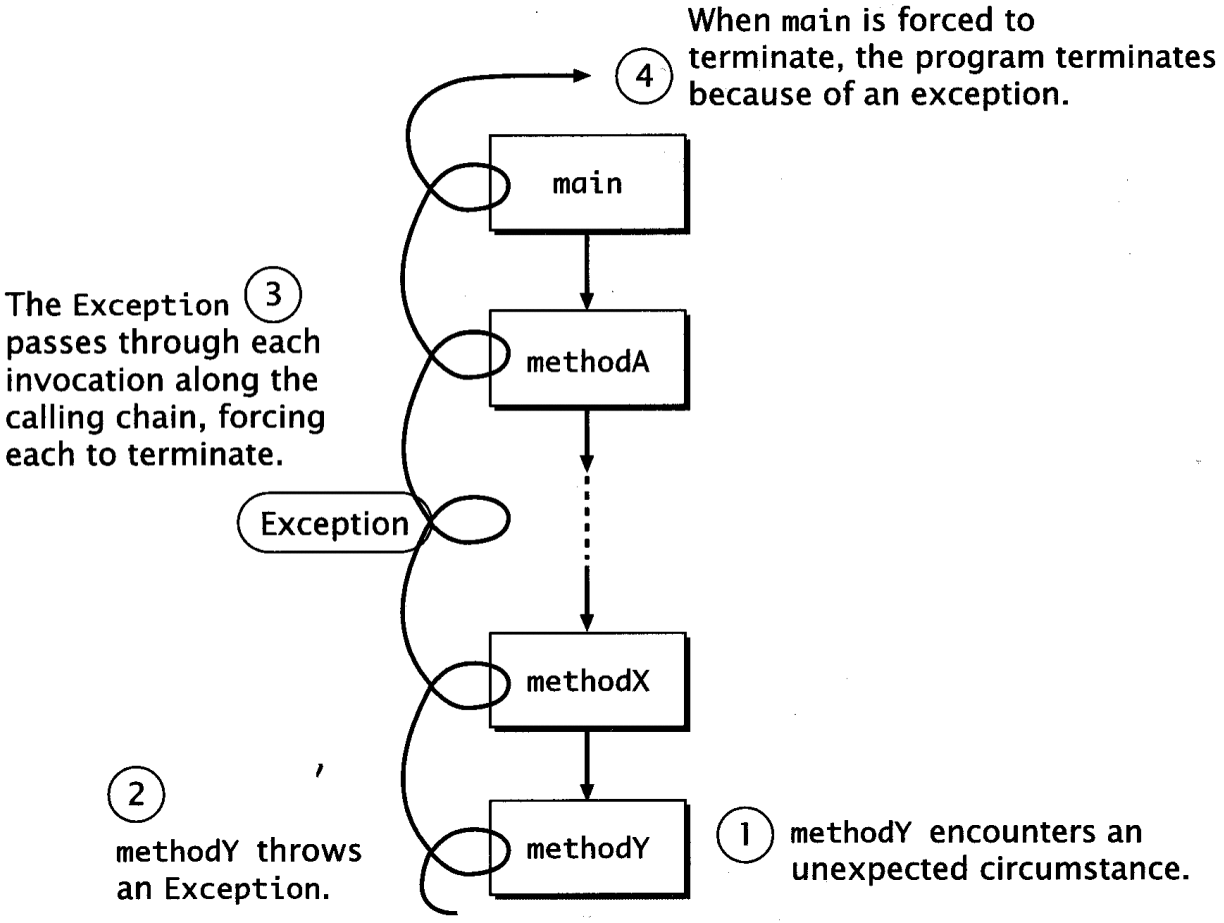
Information zum Aufrufer

- Normalerweise über einen Return-Wert
- unerwartete Situationen könnten über einen besonderen Return-Wert an den Aufrufer kommuniziert werden
- bei `parseInt()` z.B. -999 als Return-Wert?
  - `parseInt()` retourniert aufgrund des Prototypen immer einen Return-Wert...
  - warum und wann funktioniert das also nicht?

Werfen einer Exception

- Java stellt einen eigenen Mechanismus zur Verfügung, damit Methoden auf unerwartete Situationen reagieren können:
  - `throw reference`
    - `reference` ist eine Referenz auf ein Objekt einer Subklasse von Exception
    - Exception ist eine Klasse, die unerwartete Situationen repräsentiert
  - `throw new Exception-class(String-Argument);`
  - Führt eine Methode das `throw` Statement aus, ***wirft diese eine Exception***.

Kette von Methodenaufrufen



Exception werfen

- Das Werfen einer *Exception* bewirkt, dass die ausführende Methode sofort terminiert.
- Das `throw` Statement liefert jedoch weder einen Return-Wert, noch kann der Aufrufer dort fortsetzen, wo der Aufruf erfolgte.
- Die geworfene *Exception* wird entlang der Aufrufkette weitergereicht und bewirkt, dass alle Methoden entlang dieser Kette umgehend terminieren.
- Für jede Methode entlang dieser Kette erscheint es, als ob die jeweils von ihr aufgerufene Methode die *Exception* geworfen hätte.
- Dies erfolgt bis zurück zur `main()` Methode.

Beispiel: `parseInt()`

- Werfen einer Exception:

```
public static int parseInt(String s)
                throws NumberFormatException {
    // Code der Methode

    if (...) {

        throw new NumberFormatException();

    }
    // ...
}
```

Anzeige der Exception (error stack trace)

- Anzeige der geworfenen *Exception*:

```
SomeException
    at TryThrow.method2(TryThrow.java:18)
    at TryThrow.method1(TryThrow.java:15)
    at TryThrow.main(TryThrow.java:12)
```

  - In Zeile 12 von `main()` wurde `method1` aufgerufen,
    in Zeile 15 von `method1` wurde `method2` aufgerufen,
    in Zeile 18 von `method2` wurde die Exception geworfen.

Vorteile von Exceptions

- Separating Error-Handling Code from “Regular” Code

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

- potential errors:
  - What happens if the file can’t be opened?
  - What happens if the length of the file can’t be determined?
  - What happens if enough memory can’t be allocated?
  - What happens if the read fails?
  - What happens if the file can’t be closed?

Code for error detection, reporting, handling

```
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;
}
```

Error handling with Exceptions

- Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere:

```
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;
    }
}
```

Propagating Errors Up the Call Stack

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

without exceptions

Propagating Errors Up the Call Stack

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

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

with exceptions

Error Types: Grouping & Differentiating

- All exceptions thrown within a program are objects, grouping of exceptions is a natural outcome of the class hierarchy
- A method can write specific handlers that can handle a

```
catch (FileNotFoundException e) {
    ...
}
```

- For example, to catch all I/O exceptions, regardless of

```
catch (IOException e) {
    e.printStackTrace();     // output of error stack
    e.printStackTrace(System.out);  // send trace to stdout
}
```

Vererbungsstruktur der Exceptions

- Throwable
  - Error (ungeprüft)
    - LinkageError
    - VirtualMachineError
      - OutOfMemoryError
  - Exception (alle geprüft, ausser RuntimeException)
    - RuntimeException (ungeprüft)
      - ArithmeticException
      - IndexOutOfBoundsException
        - ArrayOutOfBoundsException
      - IllegalArgumentException
        - NumberFormatException
    - IOException (geprüft, dh zu behandeln)
      - FileNotFoundException
      - MalformedURLException
    - InterruptedException (geprüft)

geprüft ... vom Compiler geprüft, dh die Exception muss im Programm behandelt werden!!!

Anforderung

- Jede Methode, die eine Exception werfen könnte, muss dies in ihrer Deklaration mittels `throws Exception` anführen.
- Ein `throws` Ausdruck besteht aus
  - Schlüsselwort `throws` und
  - der Liste von Subklassen von Exception, die von der Methode geworfen werden können.
- Dies trifft auch zu auf Methoden, die Methoden mit `throws` Ausdruck aufrufen.

Spezifische Ereignisse

- Spezifische Information über die Art der möglichen Exception kann angegeben werden:

```
throws IOException

// more specific subclasses of IOException
throws FileNotFoundException, RemoteException
```

- Labels für Exceptions mit mehr Information:

```
throw new FileNotFoundException("log file
                                is always necessary");
```

Runtime Exceptions

- Runtime Exceptions können/müssen nicht spezifiziert werden, da sie zur Laufzeit beliebig auftreten können.
- Diese sind meist Programmierfehler...
- Erweiterungen von `RuntimeException` sind z.B.
  - `NullPointerException`
  - `IOException`
  - `NumberFormatException`

Exception Handling

- Möglichkeit, auf eine Exception zu reagieren: „*catch an exception*“
- und damit die Kaskade zu brechen und geeignet fortzusetzen (anstatt der abrupten Termination entlang der Aufrufkette)

```
try {
   someObject.someMethod();
} catch (Exception e) {
   // statements that are executed if and only if an
   // exception is thrown by the code within the try
}
```

Beispiel: Movie /1

```
// ignoring the unexpected
public static Movie readMovie(BufferedReader br) throws IOException {
   String name;
   int playingTime;
   Movie newMovie;

   name = br.readLine();
   if (name==null)
      return null;
   playingTime = Integer.parseInt(br.readLine());
   newMovie = new Movie(name, playingTime);
   return newMovie;
}
```

Beispiel: Movie /2

```
// partially handling the unexpected
public static Movie readMovie(BufferedReader br) throws IOException {
    String name;
    int playingTime;
    Movie newMovie;

    name = br.readLine();
    if (name==null)
        return null;

    try {
        playingTime = Integer.parseInt(br.readLine());
    } catch (NumberFormatException e) {
        System.err.println("Bad playing time format for "+ name);
        throw e;
    }

    newMovie = new Movie(name, playingTime);
    return newMovie;
}
```

Beispiel: Movie /3

- Fehlerbehandlung <source code Movie.java>
  - Anzeigen einer Fehlermeldung (Warnung)
  - Verwerfen der fehlerhaften Daten
  - Fortfahren und nächsten Film-Eintrag lesen und verarbeiten

```
gotGoodData = false;
while (!(name==null || gotGoodData)) {
   gotGoodData = true;// Optimistic! but if an Exception
                      // is thrown, it will be set to false.
   try {
      playingTime = Integer.parseInt(br.readLine());
   } catch (NumberFormatException e) {// Skip this, do next
       System.err.print("Bad playing time data for "+name);
       System.err.println(" -- movie skipped");
       gotGoodData = false;
       name = br.readLine();
   }
}
```

`finally` Block

- The final step in setting up an exception handler is to clean up before allowing control to be passed to a different part of the program.
- The `finally` block is optional and provides a mechanism to clean up regardless of what happens within the try block. Use the finally block to close files or to release other system resources.

Clean up with `finally`

```
try {
        ...
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Caught " + 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");
            }
        }
```

Verantwortung für Unerwartetes

- Jedes Objekt hat eine Verantwortung für sein Verhalten.
- Das Behandeln von unerwarteten Ereignissen ist meist Bestandteil des gesamten Objekt-Verhaltens.
- Vorgangsweise:
  - Wo wird eine *Exception* geworfen und welche?
  - Analysieren der Objekte entlang der Aufrufkette
  - Wer kann aufgrund seiner Verantwortung die Exception behandeln?

Specific Exception Handlers

- However, in most situations, you want exception handlers to be as specific as possible.
- The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy.
- In effect, by not catching specific errors, the handler must accommodate any possibility.
- Exception handlers that are too general can make code more error prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.
- Exceptions are not always Errors! (-> finding bad URLs example)

Case Study

- A Line-Oriented Calculator
  - Should do addition, subtraction, division, multiplication
  - Will use line input/output
- User will enter
  - Operation, space, number
  - Calculator displays result

Case Study

- Proposed initial methods
  - Method to **`reset`** value of **`result`** to zero
  - Method to **`evaluate`** result of one operation
  - Method **`doCalculation`** to perform series of operations
  - Accessor method **`getResult`**: returns value of instance variable **`result`**
  - Mutator method **`setResults`**: sets value of instance variable **`result`**

Case Study

- View exception class, listing 9.10
  `class UnknownOpException`
- View first version of calculator, listing 9.11
  `class PreLimCalculator`

```
Calculator is on.
Format of each line: operator space number
For example: + 3
To end, enter the letter e.
result = 0.0
+ 4
result + 4.0 = 4.0
updated result = 4.0
* 2
result * 2.0 = 8.0
updated result = 8.0
e
The final result is 8.0
Calculator program ending.
```

Sample screen output

Case Study

- Final version adds exception handling
- Ways to handle unknown operator
  - Catch exception in method **`evaluate`**
  - Let **`evaluate`** throw exception, catch exception in **`doCalculation`**
  - Let **`evaluate`**, **`doCalculation`** both throw exception, catch in **`main`**
- Latter option chosen

Case Study

- View final version, listing 9.12

`class Calculator`

```
Calculator is on.
  % 4
    -2
    result - 2.0 = 78.0
    updated result = 78.0
    * 0.04
    result * 0.04 = 3.12
    updated result = 3.12
    e
    The final result is 3.12
    Calculator program ending.
```

Sample screen output

Zusammenfassung

- Wie ein Objekt unerwartete Ereignisse behandelt gehört zu seinem Verhalten.
- Java liefert eine Klasse `Exception`, die unerwartete Situationen modelliert.
  - Darin gibt es ein `throws` statement, um den Kontrollfluss abrupt zu ändern.
  - Dieses `throws` wird im ersten einschliessenden `try` - `catch` Block für diese Exception behandelt
  - Gibt es keine Behandlung erfolgt eine Terminierung der Methoden entlang der Aufrufkette.