Softwareentwicklung II (IB)

Exceptions

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 28.01.2020 17:34

Inhaltsverzeichnis

Exceptions

Idee

- **Laufzeitfehler** = Probleme im ablaufenden Programm (im Ggs. zu Fehlern beim Übersetzen)
- Ursachen vielfältig:
  - Logische Fehler
  - fehlerhafte Bedienung
  - Problem im Java-Laufzeitsystem
  - …
- **Exceptions** (dt. „Ausnahmen“) = Sprachmittel zur kontrollierten Reaktion (Assertions Sonderfall von Exceptions)
- Angemessene Reaktion abhängig von der Art des Fehlers:

logischer Fehler:	⇒ Programm stoppen
Bedienungsfehler:	⇒ Aufforderung zur Korrektur, neuer Versuch
Problem im Java-Laufzeitsystem:	⇒ ?

- Vorteile von Exceptions:
  - Laufzeitfehler müssen behandelt werden, können nicht ignoriert werden
  - Code für normalen Ablauf und Code für Fehlerbehandlung getrennt
- Exception-Handling verteilt auf zwei getrennte Bereiche im Code:
  - Ausnahmesituationen erkennen und melden
  - Auf gemeldete Ausnahmesituation reagieren

Ausnahmesituationen

- Beispiel: `clip` erhält String, schneidet erstes und letztes Zeichen ab, gibt den Rest zurück:

```
String clip(String s) {
  return s.substring(1, s.length() - 1);
}
```

- Mögliche Probleme:
  - kein String (`s == null`)
  - String zu kurz (`s.length() < 2`)

- **Erkennen** der Fehlersituationen mit Tests:

```
String clip(String s) {
  if (s == null)
    …                          // kein String
  else if (s.length() < 2)
    …                          // String zu kurz
  else
    return s.substring(1, s.length() - 1);
}
```

Exceptions auslösen

- **Melden** des Fehlers durch „Werfen“ einer Exception„ (wird an anderer Stelle „aufgefangen“)
- Einfache Anweisung zum Werfen einer Exception: `throw` exception`;`
- exception = `Throwable` oder kompatibles Objekt
- Im Moment ausreichend: Objekt der Klasse `Exception`
- Ergänzung des vorhergehenden Beispiels:

```
String clip(String s) throws Exception {
  if (s == null)
    throw new Exception(); // kein String - Exception werfen
  else if (s.length() < 2)
    throw new Exception(); // String zu kurz - Exception werfen
  else
    return s.substring(1, s.length() - 1);
}
```

- Zusatz `throws Exception` in der Methodensignatur wird später diskutiert

Unterbrechung des regulären Ablaufs

- `throw` unterbricht laufenden Code sofort, nachfolgende Anweisungen entfallen
- `else` im Beispiel unnötig:

```
String clip(String s) throws Exception {
    if (s == null)
      throw new Exception();
    if (s.length() < 2)
      throw new Exception();
    return s.substring(1, s.length() - 1);
}
```

- `return` wird nur dann erreicht, wenn kein Fehler auftritt

Exception-Nachricht

- Custom-Konstruktor von `Exception` akzeptiert String
- String soll Fehlerursache beschreiben (was ist das Problem?)
- An Anwender gerichtet, nicht zur Auswertung im Programm

```
String clip(String s) throws Exception {
  if (s == null)
    throw new Exception("no string");
  if (s.length() < 2)
    throw new Exception("short string");
  return s.substring(1, s.length() - 1);
}
```

`try` und `catch`

- **Reaktion** auf Fehler („Fehlerbehandlung“) unabhängig vom Erkennen
- Regulärer Code (Ablauf ohne Fehler) und Fehlerbehandlung (Ablauf im Fehlerfall) in getrennten Programmabschnitten
- Beide als Block geklammert
- Entsprechende Blöcke markiert mit Schlüsselwörtern

`try`	für regulären Code,
`catch`	für Fehlerbehandlung

- Schema:

```
try {
  … regulärer Code …
} catch (Exception ex) {
  … Fehlerbehandlung …
}
```

- `try`- und `catch`-Block in dieser Reihenfolge lückenlos nacheinander

Beispiel für `try` und `catch`

- Beispiel einer Anwendung der Methode `clip`

- Regulärer Code bei korrektem (fehlerfreien) Ablauf:

```
public static void main(String[] args) throws Exception {
  String s = args[0];
  String c = clip(s);
  System.out.println(c);
}
```

- Mit Fehlerbehandlung:
  - Regulärer Code unverändert in `try`-Block
  - Fehlerbehandlung im `catch`-Block (hier nur Ausgabe einer Meldung)

```
public static void main(String[] args) {
  try {
    String s = args[0];
    String c = clip(s);
    System.out.println(c);
  } catch (Exception ex) {
    System.out.println("clipping failed - giving up");
  }
}
```

Ablauf

- Kontrollfluss bei regulärem Ablauf:
  - `try`-Block wird vollständig ausgeführt
  - Kein Fehler, kein `throw`
  - Ende des `try`-Blocks erreicht, `catch`-Block ignoriert
- Kontrollfluss im Fehlerfall:
  - Im `try`-Block Fehler, `throw`-Anweisung
  - Programmablauf wird sofort unterbrochen, der Rest des `try`-Blocks übergangen
  - `catch`-Block wird ausgeführt
- In beiden Fällen: Fortsetzung nach dem `catch`-Block

Einfache Exceptionsignatur

- Anwender einer Methode kennt i.d.R. nur Methodenkopf, aber nicht den Rumpf, nicht die `throw`-Anweisungen darin
- Bei welchen Methoden Exceptions erwarten und `try`/`catch` vorsehen, bei welchen nicht?

- Zur Abstimmung Methode/Aufrufer: mögliche Exceptions im Methodenkopf auflisten = **Exceptionsignatur**
- Exceptionsignatur Teil des Methodenkopfs, zusätzlich zu Ergebnistyp, Name, Parameterliste
- Schema (hier `throws`)

  ```
  returntype methodname(parameterlist) throws Exception
  ```

- Exceptionsignatur = Warnung: Diese Methode könnte scheitern und eine Exception werfen
- Beispiel: Methode `clip`

Exceptionklassen

Basisklasse `Throwable`

- Bisher: Mit `throw` Objekt vom Typ `Exception` werfen, mit `catch` fangen:

  ```
  throw new Exception();
  …
  catch (Exception ex) …
  ```

- Allgemein: Exceptionobjekte unterschiedlicher Typen zulässig, aber kompatibel zu `Throwable`
- Unterschiedliche Exceptiontypen ⇒ Klassifizierung der Art des Fehlers
- Reaktion auf Fehler in `catch` differenziert nach Exceptiontyp

Vordefinierte Exceptionklassen

- `Throwable` und abgeleitete Klassen
- `Throwable` selbst nicht zur direkten Nutzung
- `Exception` (und abgeleitete Typen) bei Fehlern im Benutzerprogramm, einschließlich Laufzeitbibliothek
- `Error` reserviert für Fehler in der JVM
- Vordefinierte Exceptionklassen, zum Beispiel:

`IndexOutOfBoundsException`	Unzulässiger Index wurde benutzt
`IllegalArgumentException`	Unzulässiger Argumentwert übergeben
`IllegalStateException`	Objekt in einem unzulässigen Zustand
`NullPointerException`	`null` wo ein Objekt sein sollte
`UnsupportedOperationException`	Aufruf einer verbotenen Methode

- Nutzbar für eigene Exceptions

```
String clip(String s) throws Exception {
  if(s == null)
    throw new NullPointerException("no string");
  if(s.length() < 2)
    throw new IllegalArgumentException("short string");
  return s.substring(1, s.length() - 1);
}
```

Benutzerdefinierte Exceptionklassen

- Vordefinierten Exceptionklassen bequem, aber unspezifisch, nicht genau passend
- Definition neuer Exceptionklassen
- Minimal: Leere Klasse von `Exception` ableiten

```
class StringClipException extends Exception
{}
```

- Enthält automatisch definierten Default-Konstruktor
- Sinnvoll: Custom-Konstruktor mit Stringparameter explizit definieren, zusätzlich zum Default-Konstruktor:

```
class StringClipException extends Exception {
    StringClipException() {}

  StringClipException(String message) {
    super(message);
  }
}
```

- Beispiel: Einsatz der maßgeschneiderten Exceptionklasse:

```
String clip(String s) throws Exception {
  if (s == null)
    throw new StringClipException("no string");
  if (s.length() < 2)
    throw new StringClipException("short string");
  return s.substring(1, s.length() - 1);
}
```

Exceptionsignaturen

- Exceptionsignatur kündigt Art der Exceptions einer Methode an

- Mehrere Exceptiontypen als Liste angeben:

  ```
  returntype methodname(parameterlist) throws exception1, exception2
  ```

- Alle tatsächlich im Rumpf mit `throw` geworfenen Exceptions müssen kompatibel zu einer Exception der Signatur sein
- Vorhergehendes Beispiel wird übersetzt, weil `StringClipException` abgeleitet von `Exception`
- „`throws Exception`“ wenig hilfreich, weil
  - der Aufrufer vor einer beliebigen Exception gewarnt wird,
  - tatsächlich aber nur eine `StringClipException` geworfen wird und keine andere.
- Möglichst spezifische Exceptionsignatur angeben

```
String clip(String s) throws StringClipException {
  if (s == null)
    throw new StringClipException("no string");
  if (s.length() < 2)
    throw new StringClipException("short string");
  return s.substring(1, s.length() - 1);
}
```

- Nur checked exceptions in Signaturen: Signatur definiert Vertrag mit Anwender, aber unchecked exception signalisiert Verletzung des Vertrages

Fehlerhafte Exceptionsignaturen

- Compiler prüft Exceptionsignatur, erkennt
  - Fehlende Exceptionsignatur

    ```
    void foo() {
      throw new Exception();
    }
    ```

  - Falsche Exceptionsignatur

    ```
    void foo() throws NullPointerException {
      throw new Exception();
    }
    ```

- Fehlermeldung: `unreported exception type; must be caught or declared to be`

Durchreichen von Exceptions

- Methode ohne `throw` ruft andere Methode, die Exception wirft
- Beispiel: `tripleClip` schneidet die ersten und die letzten drei Zeichen eines Strings ab:

```
String tripleClip(String s) {
  return clip(clip(clip(s)));
}
```

- Erzeugt selbst keine Exception, aber jeder `clip`-Aufruf kann eine werfen
- Quelle von Exceptions für einen Aufrufer von `tripleClip` unerheblich
- Methodensignatur nennt …
  1. die selbst ausgelösten Exceptions
  2. die Exceptions aller aufgerufenen Methoden
- Beispiel `tripleClip`:

```
String tripleClip(String s) throws StringClipException {
  return clip(clip(clip(s)));
}
```

- Compiler prüft Vollständigkeit

Runtime-Exceptions

- Beispiel: JVM wirft eine `NullPointerException`, wenn das Objekt beim Zugriff fehlt:

```
Rational r = null;
r.reduce();      // NullPointerException
```

- Könnte bei jedem Methodenaufruf passieren ⇒ `NullPointerException` wäre in (fast) jeder Exceptionsignatur nötig – hoher Aufwand, allgegenwärtige (= nutzlose) Information
- Kompromiss: Exceptiontyp `RuntimeException` darf in Signaturen fehlen
- Ebenso von `RuntimeException` abgeleitete Typen, z.B. `NullPointerException`

checked vs. unchecked Exceptions

- **Unchecked exceptions** = `RuntimeExceptions` und `Errors`
  - Behandlung vom Compiler ignoriert

  - Entwickler alleine verantwortlich, keine Unterstützung
- **Checked exceptions** = alle anderen Exceptions
  - Behandlung (Fangen oder Weitergeben) vom Compiler erzwungen
  - Irrtümliches Übersehen nicht möglich

Wann Runtime-Exceptions?

- Eigene oder vordefinierte `RuntimeException`s werfen in ausweglosen Situationen
- Reparaturversuch sinnlos, Programm muss sofort abbrechen
- Ursache meist Fehler im Code — ein korrektes Programm wirft keine `RuntimeException`
- Nicht wegen Fehlbedienung eines Programms
- Weglassen in Exceptionsignatur akzeptabel, weil `catch` ohnedies zwecklos
- Vergleichbar mit `Assertion`s: Absichern der Korrektheit von Code

Errors

- `Error` und abgeleitete Typen reserviert für Fehler der JVM
- Benutzercode sollte (von sich aus) keine `Error`s erzeugen (Ausnahme `AssertionError`)
- Beispiele:

`OutOfMemoryError`	JVM hat allen verfügbaren Speicherplatz verbraucht
`ClassFormatError`	Versuch eine defekte Bytecodedatei zu laden
`VirtualMachineError`	die JVM ist auf einen internen Fehler gelaufen

- Auf `Error`s keine sinnvolle Reaktion möglich ⇒ nicht mit `catch` fangen
- `Error`s fehlen in Exceptionsignaturen, ebenso wie `RuntimeException`s
- Sonderfall `AssertionError` = Gescheiterte Assertion: Programm ist fehlerhaft und muss sofort gestoppt werden