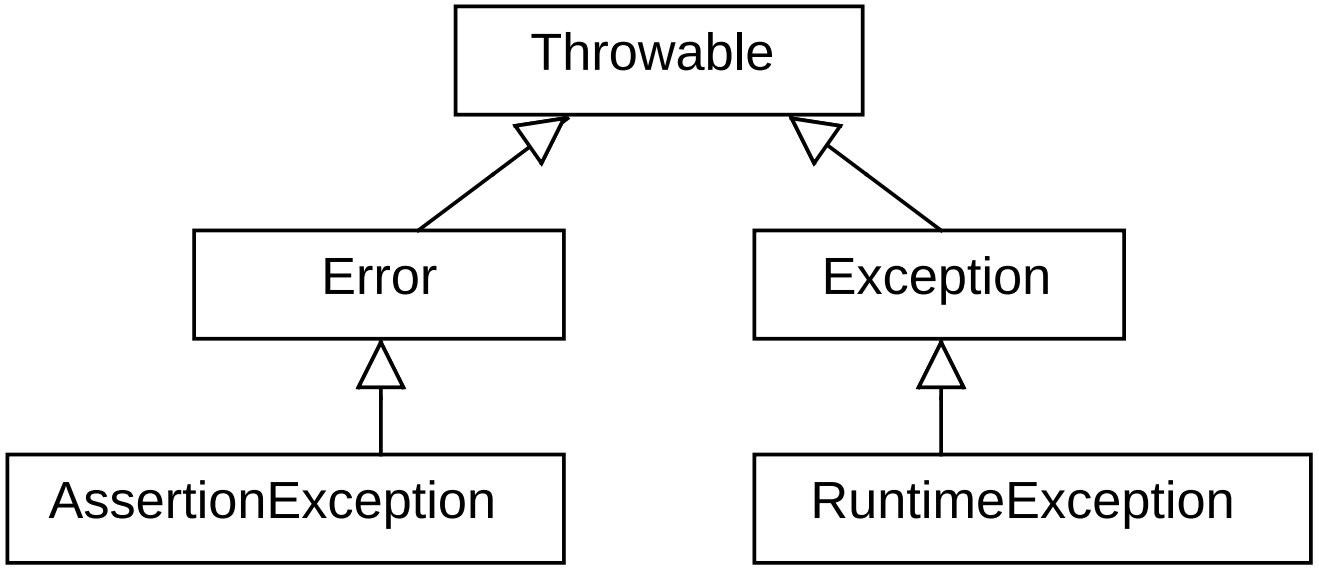
Exceptions

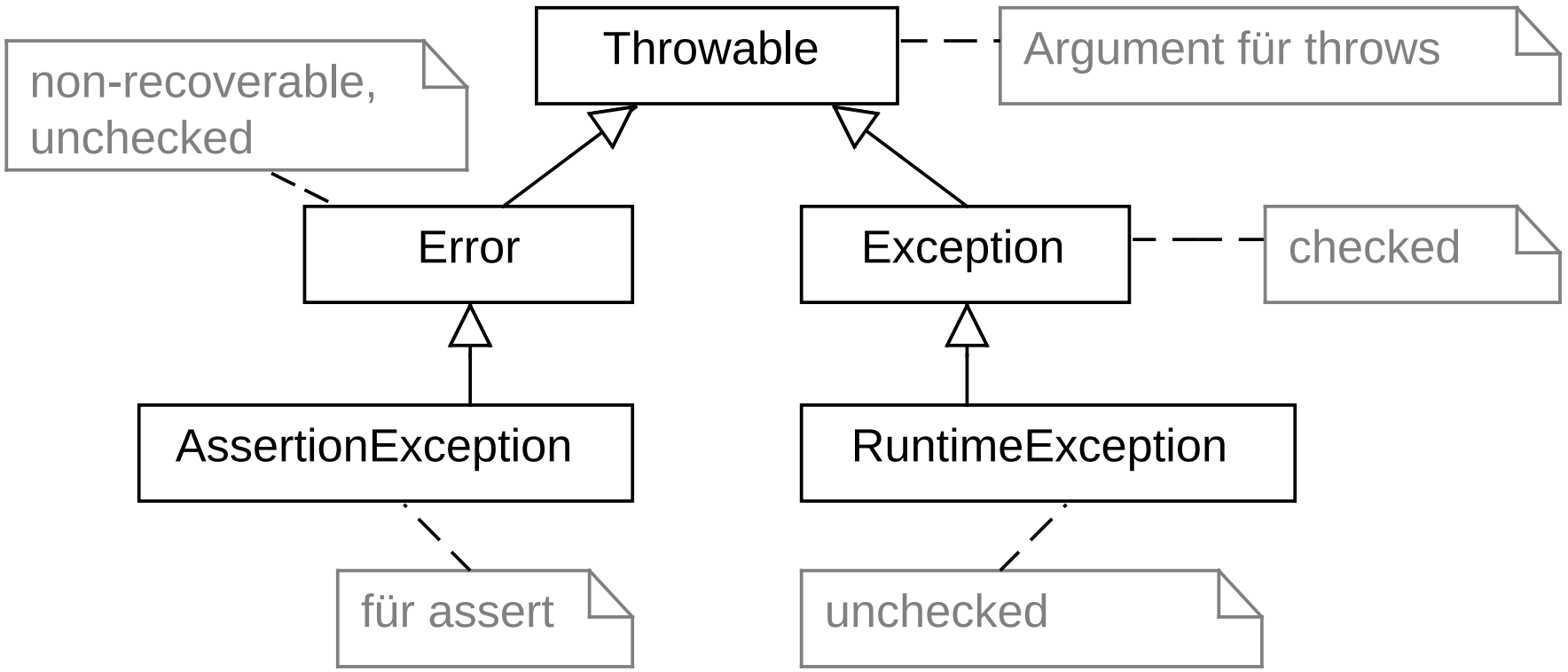
KW 27, Zentralübung Informatik II

2006-07-04

Throwables


Throwable
Error
Exception
AssertionException
RuntimeException

Throwables


Throwable
Argument für throws
non-recoverable,
unchecked
Error
Exception
checked
AssertionException
RuntimeException
für assert
unchecked

Assertions vs. Exceptions

Eine Assertion

```
assert i!=0 : "Argument is zero";
```

ist im Prinzip identisch zu

```
if (i==0) throw new AssertionException("Argument is zero");
```

Aber: Assertions sind üblicherweise nur zur Entwicklungszeit (und nicht im normalen Einsatz) angeschaltet und werden daher anders als Exceptions eingesetzt. . .

Assertions vs. Exceptions

Zitat aus http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html

> There are many situations where it is good to use assertions. [...]
>
> - Internal Invariants
> - Control-Flow Invariants
> - Preconditions, Postconditions, and Class Invariants
>
> There are also a few situations where you should not use them:
>
> - Do not use assertions for argument checking in public methods.
>   [Diese Checks sollten immer gemacht werden und die richtige Exception werfen.]
> - Do not use assertions to do any work that your application requires for correct operation.
>   [Dann funktioniert der Code nur, wenn Assertions angeschaltet sind.]

Beispiel: Checked Exception

```
public class CheckedException extends Exception {
}

public class CheckedUser {
    public int compute(int i) throws CheckedException {
        if (i < 0) throw new CheckedException();
        return 24;
    }
    public static void main(String[] args) {
        try {
            new CheckedUser().compute(3);
        } catch (CheckedException e) {
        } // Sonst unbedingt vermeiden: leerer catch-Block!
    }
}
```

Beispiel: Unchecked Exception

```
public class UncheckedException extends RuntimeException {
}

public class UncheckedUser {
    public int compute(int i) {
        if (i < 0)  throw new UncheckedException();
        return 24;
    }
    public static void main(String[] args) {
        new UncheckedUser().compute(3);
    }
}
```

Techniken: Von checked zu unchecked

„Verstecke“ eine checked Exception in einer unchecked Exception:

```
public int readOneCharachter(String fileName) {
    try {
        FileReader in = new FileReader(fileName);
        return in.read();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Die checked Exception wird als cause-Parameter dem Konstruktor von `RuntimeException` übergeben (Ausnahmeverkettung, engl. „exception chaining“).

Techniken: Kapseln von Exceptions

Anwender dieser Methode sollen nicht durch eine Vielzahl von Exceptions verwirrt werden (die letztlich Implementierungsdetails sind).

```
public int initDatabase() throws DatabaseException {
    try {
        ...
    } catch (IllegalArgumentException e) {
        throw new DatabaseException("Could not open database", e);
    } catch (ParserException e) {
        throw new DatabaseException("Could not open database", e);
    } catch (IOException e) {
        throw new DatabaseException("Could not open database", e);
    }
}
```

Wie behandelt man hier Ausnahmen am besten?

```
/** <ul>
 * <li> Read a mapping from String to String,
 * <li> replace keys in templateFile with values and
 * <li> write the result to outFile
 * </ul> */
public static void insert(File mapFile, File inFile, File outFile)
    throws Exception {
    // throws IOException, ParseException:
    Map<String, String> map = parseMapping(new FileReader(mapFile));
    InputStream in = new FileInputStream(inFile); // FileNotFoundException
    OutputStream out = new FileOutputStream(outFile); // FileNotFoundExcep

    performSubstitution(map, in, out); // throws IOException
}
```

Methode: In Transaktionen denken

- Transaktion: Unteilbare Folge von Anweisungen

  $\Rightarrow$ Sinnlos, zwischendurch abzubrechen (entweder die Transaktion wird ganz ausgeführt oder gar nicht).

- Überwache diese Folge als Ganzes (ein einziger `try`-Block):
  - Fehler behebbar $\Rightarrow$ selbst beheben.
  - Fehler nicht behebbar $\Rightarrow$ Exception werfen.
  - Fehler besser vom Aufrufer behandelt $\Rightarrow$ Exception werfen.

Lösung (1/2)

```
public static void insert2(File mapFile, File inFile, File outFile)
        throws IOException {
    Map<String, String> map; // ausserhalb von try, muss länger leben!
    try { // Transaktion 1
        map = parseMapping(new FileReader(mapFile));
    } catch(IOException e) {
        map = null;
    } catch (ParseException e) {
        map = null;
    }
    if (map == null) map = new HashMap<String, String>(); // fix
```

Lösung (2/2)

```
    InputStream in;
    try { // Transaktion 2
        in = new FileInputStream(inFile);
    } catch (FileNotFoundException e) {
        in = System.in;  // Behebe Problem
    }
    OutputStream out;
    try { // Transaktion 3
        out = new FileOutputStream(outFile);
    } catch (FileNotFoundException e) {
        out = System.out; // Behebe Problem
    }
    performSubstitution(map, in, out); // werfe Exception
}
```

Techniken: Zentrale Überprüfungs-Methoden

```
public class CheckArgument {
    public void prepend(String str) {
        ensureNonEmptyString(str);
        // ...
    }
    public void append(String str) {
        ensureNonEmptyString(str);
        // ...
    }
    private void ensureNonEmptyString(String str) {
        if (str == null || str.length() == 0) {
            throw new IllegalArgumentException(
                    "Expected non-empty string: "+str);
} } }
```

Techniken: Standard-Exceptions kennenlernen

...und selbst verwenden.

- UnsupportedOperationException
- IllegalArgumentException
- NullPointerException
- IndexOutOfBoundsException
- NoSuchElementException
- IllegalStateException

Warum Unit-Tests (1/2)?

- Sonst überprüft man auch, ob ein Programm funktioniert:
  - Manuell: Unit-Tests sind wiederverwendbar.
  - `print`-Anweisungen in der `main`-Methode: Unit-Tests zeigen eindeutig, ob ein Test klappt und sind gruppierbar.

- Sicherheit bei Änderungen: beim Programmieren sollte man folgende Aktivitäten nicht vermischen:
  - Aufräumen (Refactoring): Keine neue Funktionalität. Hier sind Unit-Tests essentiell.
  - Neue Funktionen implementieren.

Warum Unit-Tests (1/2)?

- Teamarbeit: Hier verschärft sich das Problem des vorhergehenden Punktes („Müller hat schon wieder meinen Code kaputt gemacht“).

- Dokumentation: Unit-Tests sind Anwendungsbeispiele.