Inhaltsverzeichnis

1. Exception Handling

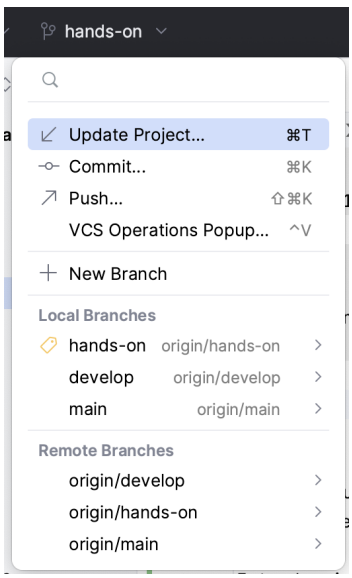
1.1. Preparation

Das **Projekt** bzw. der "*lokale Workspace*", d.h. euer lokales Arbeitsverzeichnis, in dem alle Sourcen liegen, muss als allererstes zum Start in den Tag aktualisiert werden, d.h. ...

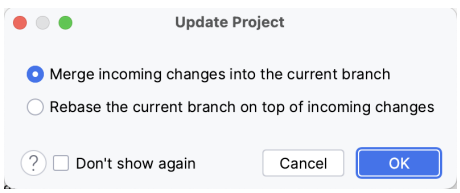
↙ `Update Project…`

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:



Danach muss - im sich öffnenden Dialog - noch folgendes bestätigt werden: *merge incoming changes into the current branch*



Der Vorgang sollte mit einer Erfolgsmeldung abschließen.

1.2. Theorie & Einführung

1.2.1. Exceptions Hierarchie

Allgemein:

- ☐ Checked vs. Runtime (unchecked) exceptions
- ☐ Technische vsw. Fachliche Exceptions
- ☐ Exception Handling - was ist zu tun?

Fachlicher Kontext

```
+ Fach-Exceptions (z.B. InvalidTrainNumberException, DelayedDepartureWarning)
+ Dokumentation des fachlichen Klassenmodells aus den vorangegangenen Beispielen
```

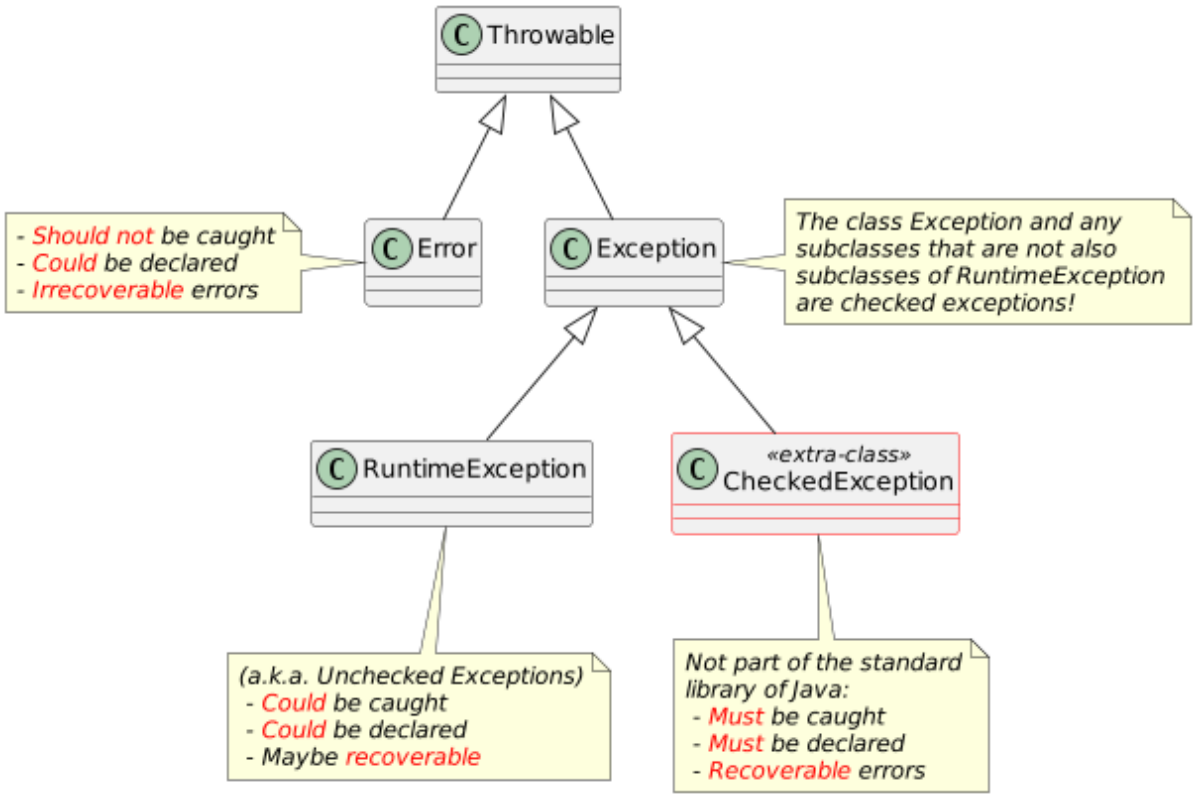


Figure 1. Exceptions Hierarchie

Throwable

Die `Throwable` Klasse ist die Superklasse von allen `Errors` oder `Exceptions` innerhalb von Java. Nur Objekte, die Instanzen dieser Klasse oder einer seiner Subklassen sind, werden von der JVM selbst geworfen, durch `throw new` manuell geworfen oder das entsprechende Schlüsselwort `throws` deklariert werden. Gleichermaßen können nur diese oder ihre Subklassen als Argumenttyp im `catch` Abschnitt genutzt werden.

Die wichtigsten Codefragmente:

```java
public void process() throws ValidationException {
   // code that may throw an exception
}

public void process() {
  // code that may throw an exception in
  // a specific situation
  if (!condition) {
    throw new ValidationException();
  }
}

try {
    // ...
} catch (ValidationException ve) {
    // ...
}
```

Error

Die `Error` Subklasse zeigt ein "ernstes" Problem an, das eine Applikation nicht "fangen" oder "behandeln" sollte. Die meisten solcher Fehler bilden außergewöhnliche Fehlerbedingungen oder -zustände ab, die (in aller Regel) nicht *zur Laufzeit* gehoben werden können.

Exception

Die Klasse `Exception` und dessen Subklassen bilden Situationen im Code ab, die bekannt sind, eintreten könnten und daher "gefangen" und behandelt werden sollten. Tritt eine solche geplante Fehlersituation auf, so sollte der Fehler so behandelt werden, dass die Applikation nicht abgebrochen werden muss. Eine häufige Reaktion auf diese Art von Fehlern münden häufig in Meldungen an die Benutzer einer Anwendung.

Ein **Beispiel**-Klassenmodell:

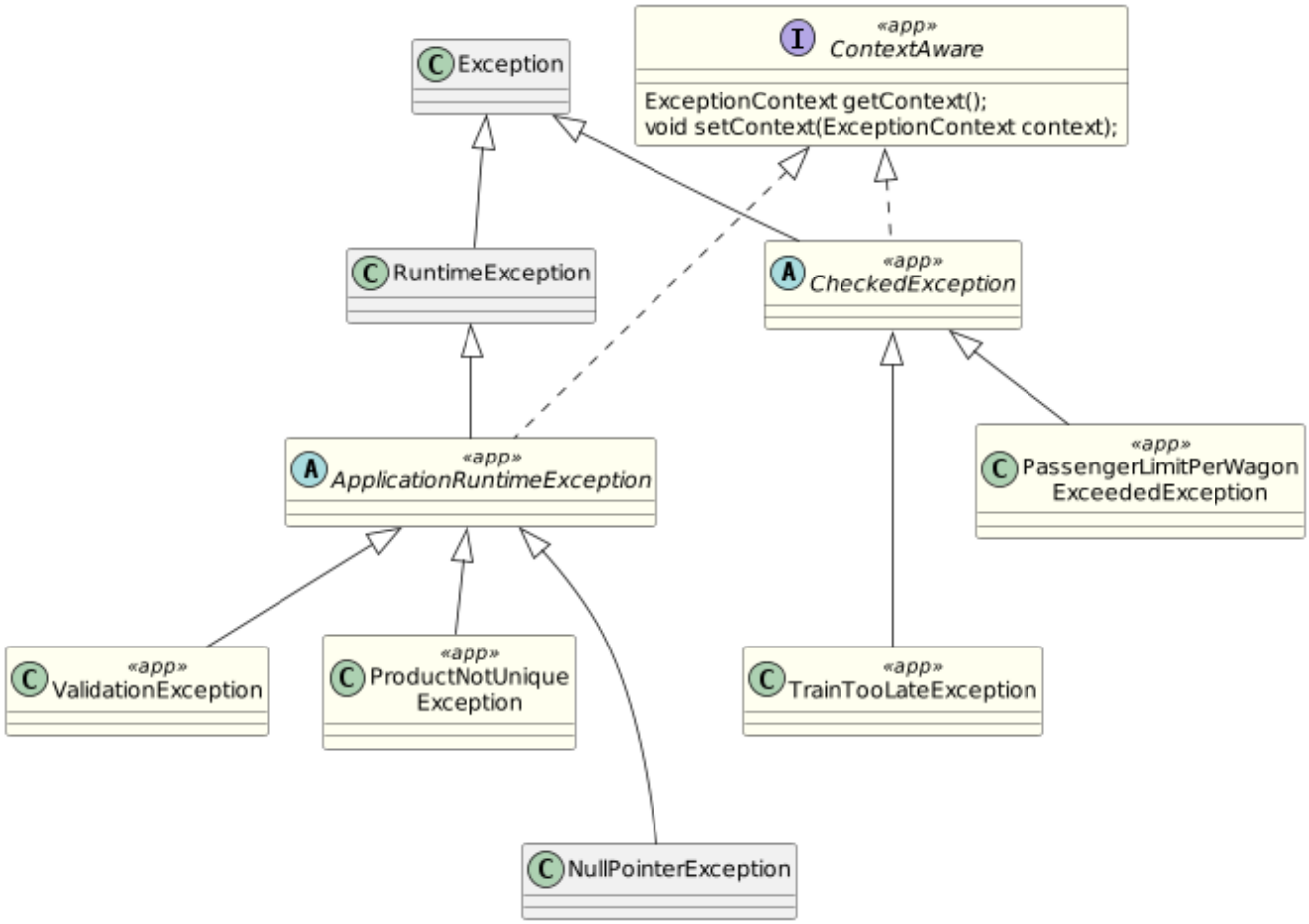


Figure 2. Exceptions Beispiel

1.2.2. Exception Handling

Grundsätzlich stellt Java einen sogenannten `exception handler` bereit, und zwar einfach mithilfe des `catch` Abschnittes. Hier sollte der abgefangene Fehler "behandelt" werden, aber **NICHT** so:

```java
} catch (FileNotFoundException e) {
    // DON'T DO THIS!
    e.printStackTrace();
}
```

Besser ist eine echte Verarbeitung des Fehlers. Hier sind sehr verschiedene Prozesse möglich, anhängig vom eingetretenen Fehler. In vielen Fällen sind bei Fehlern sowohl ...

- *technische* als auch
- *fachliche*

Dinge zu tun. Dazu kann z.B. ein eigener, applikationsspezifischer `ExceptionHandler` eingesetzt werden, der die Behandlung an eine andere Komponente *delegiert*:

```java
try {
    risky();
} catch (Exception ex) {
    // do something important for this exception situation
    this.getLogger().error(ex);
    this.rollback();
    this.clearCache();
    this.sendEmailToVIP();
}
```

oder

```java
catch (IOException ioe) {
    // we want to handle this exception
    // our own way, using an application-
    // specific exception handler!
    handler.handle(ioe);
}
```

1.2.3. Mehrere Exceptions

In manchen Fällen gibt es Methoden oder Codeabschnitte, die gleich mehrere Fehler verursachen können. Sind dies `checked` Exceptions, so müssen sie alle mittels `catch` erfasst und behandelt werden. Dazu gibt es 3 Optionen:

1. Fangen der **allgemeinsten Exception** als derjenigen, von denen alle anderen vorkommenden Exceptions abgeleitet sind

```java
try {
    risky();
} catch (Exception ex) {
    // ...
}
```

→ *Anti-Pattern: Generische Exception Handler*

2. **Mehrere** `catch` Abschnitte

```java
try {
    risky();
} catch (FileNotFoundException ex) {
    // ...
} catch (EOFException ex) {
    // ...
}
```

3. Ein **Multi-Catch** Block

```java
try {
    risky();
} catch (FileNotFoundException | EOFException ex) {
    // ...
}
```

→ *Anti Pattern "Throw-Rethrow"*

```java
try {
    risky();
} catch (FileNotFoundException ex) {
    throw new IAmSureThisIsAMuchBetterException(ex);
}
```

1.2.4. Der `finally` Block

Der `finally` Block, der grundsätzlich zum Konstrukt `try-catch-finally` gehört, wird *immer* ausgeführt, wenn der `try` Block beendet wird. Dies stellt sicher, dass der `finally` Block auch dann ausgeführt wird, wenn eine unerwartete Exception aufgetreten ist. Darüber hinaus ist der `finally` Block auch über das reine Exception Handling hinaus nützlich, er erlaubt dem Entwickler insbesondere ein *clean up* durchzuführen, d.h. allokierte Ressourcen wie z.B. geöffnete Dateien oder Speicherbereiche wieder freizugeben:

```java
String data = "data-to-save-to-file";
try {

    // can go wrong
    fileWriter.write(data);

} catch (Exception ex) {
    // ... handle 'expectable' exception
} finally {
    // clean up 'ressources'
    if (fileWriter != null) {
        f.close();
    }
}
```

Vor allem bei der Verarbeitung von Dateien bei sogenannten `IO` (kurz für Input-Output) Operationen - siehe Beispiel-Code - ist das *CleanUp* sehr wichtig und eine gute Praxis!

1.2.5. `throws` und `throw`

Das `throws` **Schlüsselwort** zeigt im Rahmen einer Methodensignatur an, dass diese Methode eine Ausnahme werfen könnte.

Das Schlüsselwort `throw` dagegen wirft eine tatsächliche, konkrete Exception.

```java
public void vote(int ageOfVoter) throws TooYoungToVoteException {
  if (ageOfVoter < 18) {
    throw new TooYoungToVoteException(
      "Voter's must be at least 18 years old!");
  }
  // continue normally ...
}
```

1.3. Demonstrationen

Die Unit-Tests zur **Demonstration** finden sich hier:

```
src/test/java/de/dhbw/exceptions/ExceptionTests.java
```

Der zugehörige, in den Tests genutzte **Quellcode** findet sich hier:

```
src/main/java/de/dhbw/exceptions/demo/*.java
```

1.4. Exercises

Nutze folgendes Package für deine **Unit-Tests**:

```
src/test/java/de/dhbw/exceptions/ExerciseTests.java
```

Die im Test benutzten **Implementierungen** gehören in das Package:

```
src/main/java/de/dhbw/exceptions/exercises/*.java
```

Übung 1:

1. **Implementiere** das Interface `Executable` sowie eine Methode `void execute()`.
2. **Implementiere** dazu auch eine neue *echte, eigene* Exception, die von `CheckedException` abgeleitet werden soll.
3. **Wirf** diese neue Exception einfach mittels `throws` in der Methode `execute()`.
4. Schreibe nun einen Unit-Test, der die Methode `execute()` aufruft und die geworfene Exception **abfängt** und **behandelt**.

1.5. Tipps, Patterns & Best Practices

Empfehlung: Flache Exception Hierarchien

sind eine gute Praxis. Es erleichtert vor allem Entwicklern den Zugang zur Nutzung zum Exception Handling, da es sehr "gerne" vernachlässigt wird.

Anti Pattern: Generische Exception Handler

Das ist ein "Anti-Pattern", weil die wahre Fehlerursache hierdurch sehr schnell verschleiert wird!

Anti Pattern: Throw-Rethrow Exceptions

Das *"throw-rethrow"* Muster ist auch ein Anti-Pattern. Auch dieses erschwert stark das Erkennen der Fehlerursache und erzeugt einfach viel Code (*"Boilerplate-Code"*). Es verstösst auch gegen das `KISS` Prinzip (*Keep-It-Simple-And-Stupid*).