252-0027
Einführung in die Programmierung

12. Exceptions

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

Exceptions («Ausnahmen»)

Package java.lang

Class RuntimeException

Class RuntimeException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, CatalogException, ClassCastException, ClassNotPreparedException, CMMException, CompletionException, ConcurrentModificationException, DateTimeException, DOMException, DuplicateRequestException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, FindException, IllegalArgumentException, IllegalCallerException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, InaccessibleObjectException, IncompleteAnnotationException, InconsistentDebugInfoException, IndexOutOfBoundsException, InternalException, InvalidCodeIndexException, InvalidLineNumberException, InvalidModuleDescriptorException, InvalidModuleException, InvalidRequestStateException, InvalidStackFrameException, JarSignerException, JMRuntimeException, JSException, LayerInstantiationException, LSException, MalformedParameterizedTypeException, MalformedParametersException, MirroredTypesException, MissingResourceException, NashornException, NativeMethodException, NegativeArraySizeException, NoSuchDynamicMethodException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ObjectCollectedException, ProfileDataException, ProviderException, ProviderNotFoundException, RangeException, RasterFormatException, RejectedExecutionException, ResolutionException, SecurityException, SPIResolutionException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnknownTreeException, UnmodifiableModuleException, UnmodifiableSetException, UnsupportedOperationException, VMDisconnectedException, VMMismatchException, VMOutOfMemoryException, WrongMethodTypeException, XPathException

Exceptions: Quellen/Arten

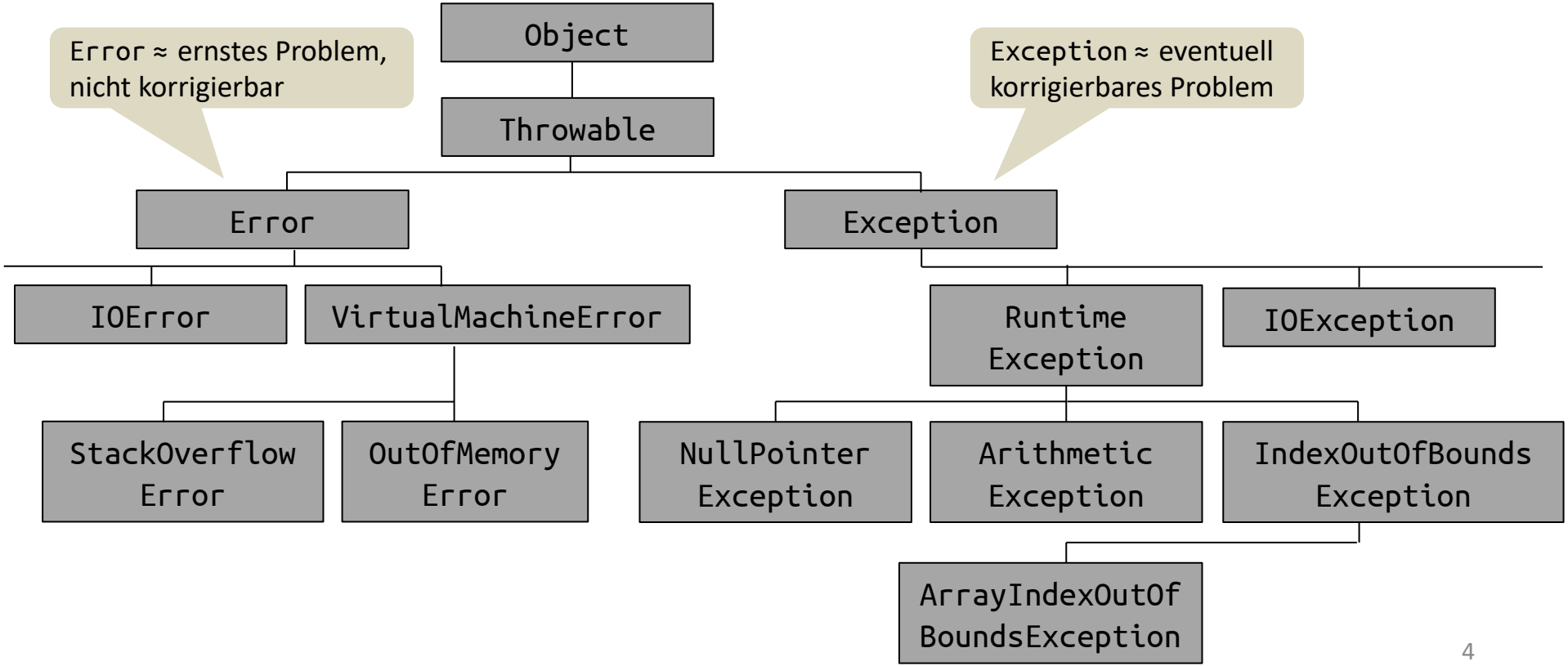
- Zur Laufzeit tritt eine Situation ein, die eine *sinnvolle* weitere Programmausführung verhindert → Laufzeitfehler. Beispiele:
  - Division eines Integers durch 0: `x / 0`
  - Zugriff ausserhalb der Array-Grenzen: `a[-1]`
  - Dereferenzieren der Null-Referenz: `null.f()`
  - Lesen eines falschen Typs im `Scanner`
  - Lesen einer nicht existierenden Datei
  - Fehlende Netzwerkverbindung
  - Kein freier Arbeitsspeicher mehr
  - Hardwarefehler (von Hardware detektiert)
  - Von Entwicklern/innen festgelegte Bedingungen

Programmieren

Umgebungsinteraktion

OS/Hardware

Vererbungshierarchie für Exceptions (Auszug)



Exceptions («Ausnahmen»): Ablaufsüberblick

1. Bemerken der Fehlersituation (durch ausgeführtes Programm oder Java-Laufzeitsystem)
2. Entsprechende Exception (ein Objekt) wird generiert
   - Wir sagen:
     Programm(-teil) **wirft** eine Exception («**throws** an exception»)
3. Exception wird weitergereicht (rückwärts im Aufrufstapel), bis ein Programmteil sinnvoll auf die Exception reagieren kann
   - Wir sagen:
     Programm(-teil) **fängt** eine Exception («**catches** an exception»).
   - Von der `main`-Methode geworfene Exceptions werden vom System gefangen

12. Exceptions

12.1 Exceptions fangen

12.2 Checked vs. Unchecked Exceptions

12.3 Exceptions werfen

12.4 Best Practices

Exceptions fangen

Für jeden Code-Teil und jede Art von Exception können wir einen **Handler (try-catch-Block)** definieren, der die Exception **fängt**

- Folge von Anweisungen, die (nur) beim Werfen der Exception in diesem Code-Block ausgeführt wird

```
try {
  // try-Block: Wirft evtl. Exception
} catch (SomeExceptionType name) {
  // catch-Block: Reagiert auf Exception
}
// Code nach try (und evtl. catch)
```

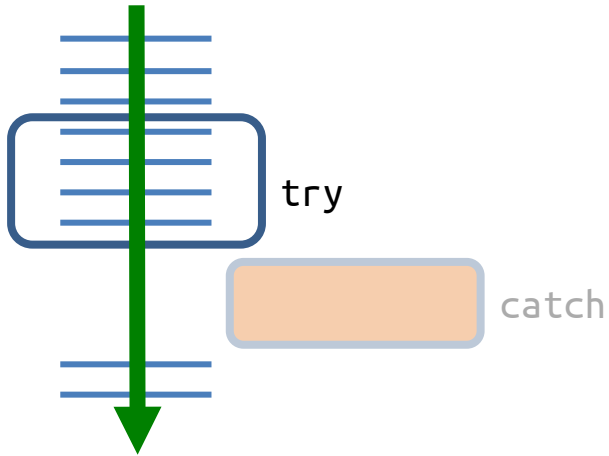
- Variable mit Exception-Objekt
- Nutzbar im catch-Block

Ablauf: Ohne Exception

Falls keine Exception im `try`-Block auftritt

1. Führe Code im `catch`-Block nicht aus
2. Führe Code nach der `try-catch`-Anweisung aus

```
try {
  // try-Block
} catch (SomeExceptionType name) {
  // …
}
// Code nach try
```

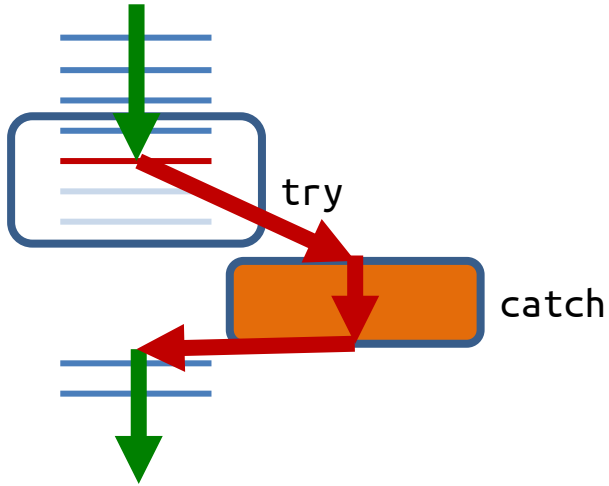


Ablauf: Mit Exception

Falls Exception (vom Typ *SomeExceptionType*) im `try`-Block auftritt

1. Breche Ausführung des `try`-Blocks ab
2. Führe Code im `catch`-Block aus
3. Führe Code nach der `try-catch`-Anweisung aus

```
try {
   // try-Block: might throw exception
} catch (SomeExceptionType name) {
   // catch-Block: handler
}
// code after the try-catch statement
```



try-catch – Beispiel 1 – Kein Fehler

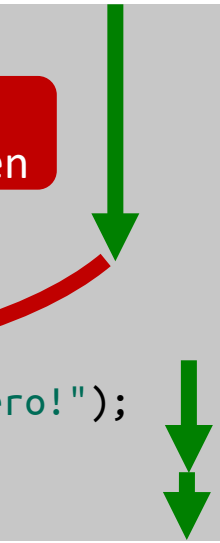
```
System.out.println("Hello");
try {
  int x = console.nextInt();
  int y = console.nextInt();
  System.out.print("Your result: " + (x / y));
  System.out.println(" - excellent choice");
} catch (ArithmeticException e) {
  System.out.println("Your second number was zero!");
}
System.out.println("Goodbye");
```

Möglicher Laufzeitfehler:
Teilen durch 0

```
Hello
4 2
Your result: 2 - excellent choice
Goodbye
```

try-catch – Beispiel 1 – Fehler

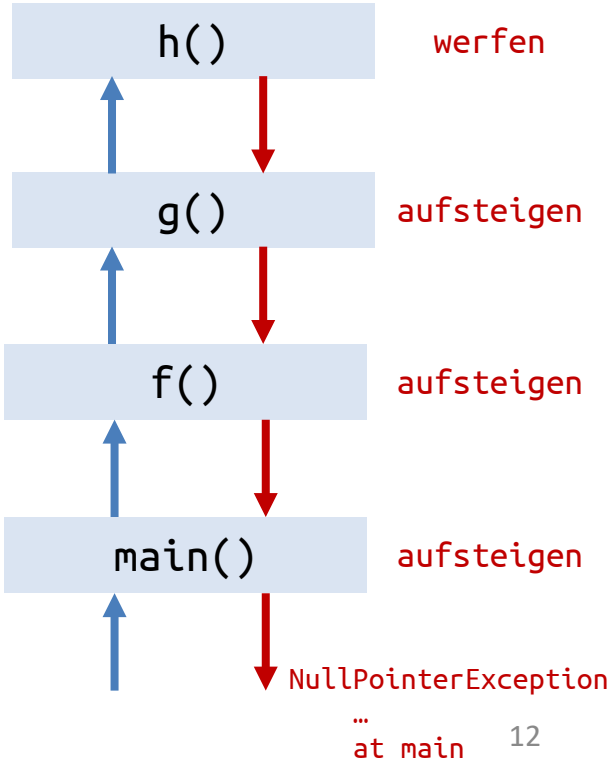
```
System.out.println("Hello");
try {
  int x = console.nextInt();
  int y = console.nextInt();
  System.out.print("Your result: " + (x / y));
  System.out.println(" - excellent choice");
} catch (ArithmeticException e) {
  System.out.println("Your second number was zero!");
}
System.out.println("Goodbye");
```



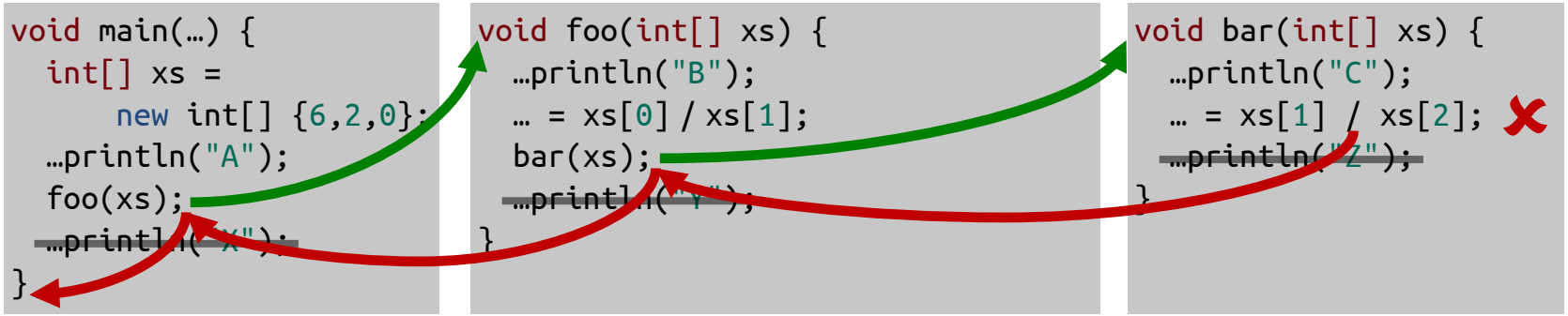

```
Hello
6 0
Your second number was zero!
Goodbye
```

Exceptions und Methodenaufrufstapel

- Wird eine Exception geworfen und nicht gefangen, dann
  - Bricht der aktuelle Methodenaufruf ab
  - Steigt die Exception zum Aufrufer hoch («bubbles up»)
- Wird sie am Aufrufort nicht gefangen
  - Bricht auch dieser Aufruf ab
  - Steigt die Exception zum nächsten Aufrufer hoch
- Steigt die Exception von `main()` auf, fängt sie die Java-Laufzeitumgebung und gibt sie aus

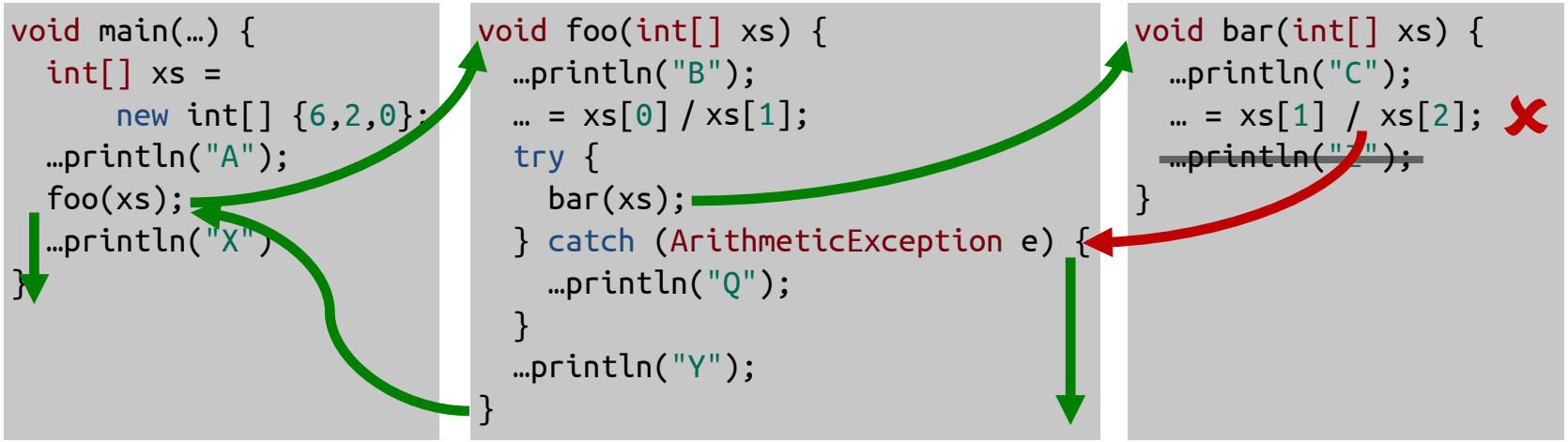


Exceptions und Aufrufstapel: Beispiel 1




```
A B C
Exception java.lang.ArithmeticException: / by zero
  at bar
  at foo
  at main
```

Exceptions und Aufrufstapel: Beispiel 2



A B C Q Y X

Unterschiedliche Exceptions fangen

- Jeder Exception-Typ kann mit einem separaten Handler gefangen werden
  - Ein `catch`-Block pro gewünschtem Exception-Typ
  - Reihenfolge ist wichtig: `catch`-Blöcke werden von oben nach unten berücksichtigt

```
try {
  // try-Block: Exceptions möglich
} catch (ExceptionType1 name) {
  // catch-Block: Handler für ExceptionType1
} catch (ExceptionType2 name) {
  // catch-Block: Handler für …
} catch (ExceptionType3 name) {
  // catch-Block: Handler für …
}
```

- Der erste passende (Subtyp!) `catch`-Block wird ausgeführt
- Nur dieser wird ausgeführt – auch wenn darunter noch ein typspezifischerer käme (z.B. falls `ExType2 extends ExType1`)

- Falls kein `catch`-Block passt, steigt die Exception im Aufrufstapel auf

Unterschiedliche Exceptions fangen: Beispiel

```
File file = new File("might_not_exists.txt");
try {
  Scanner input = new Scanner(file);
  double d = input.nextDouble();
} catch (FileNotFoundException e) {
  System.out.println("Please create missing file!");
} catch (InputMismatchException e) {
  System.out.println("First token must be a double!");
}
```

Arbeiten mit gefangenen Exceptions

```
try {
  Scanner in = new Scanner(file);
} catch (FileNotFoundException e) {
  System.out.println(e);
  System.out.println(e.getMessage());
  e.printStackTrace(System.out);
}
```

```
java.io.FileNotFoundException: missing.txt (The system
cannot find the file specified)
```

```
missing.txt (The system cannot find the file specified)
```

```
java.io.FileNotFoundException: missing.txt (The system
cannot find the file specified)
  at FileInputStream.open0(Native Method)
  at FileInputStream.open(FileInputStream.java:213)
  at FileInputStream.<init>(FileInputStream.java:152)
  at java.util.Scanner.<init>(Scanner.java:645)
  at Main.main(Main.java:12)
```

- `class Throwable` deklariert Methoden, um mit Exceptions zu arbeiten
- Für uns nicht weiter relevant

Zu guter Letzt: `finally`

- Situation: Aufräumcode, der immer ausgeführt werden muss
- Typische Beispiele: Dateien schliessen, Ressourcen freigeben

```
try {
  int data = fileScanner.nextDouble();
  …
} catch (InputMismatchException e) {
  System.out.println("…");
} finally {
  fileScanner.close(); // Schliesst geöffnete Datei
}
```

- `finally`-Block wird immer* ausgeführt, egal ob eine Exception auftrat

* Bei Interesse: https://www.baeldung.com/java-finally-keyword illustriert viele verschiedene Situationen

12. Exceptions

Checked vs. Unchecked Exceptions

- `new Scanner(file)` wirft evtl. `FileNotFoundException`
  - Code muss diese abfangen (try/catch) oder explizit weiterreichen (throws-Deklaration), sonst **Compilerfehler** – denn sie ist eine **checked Exception**

```
void foo(File file) {
  try {
    Scanner input = new Scanner(file);
    …
  } catch (FileNotFoundException e) {
    …
  }
}
```

```
void foo(File file)
    throws FileNotFoundException {

  Scanner input = new Scanner(file);
  …
}
```

- `myArray[idx]` wirft evtl. `ArrayIndexOutOfBoundsException`
  - try/catch bzw. throws-Deklaration möglich, aber nicht zwingend – **unchecked Exception**

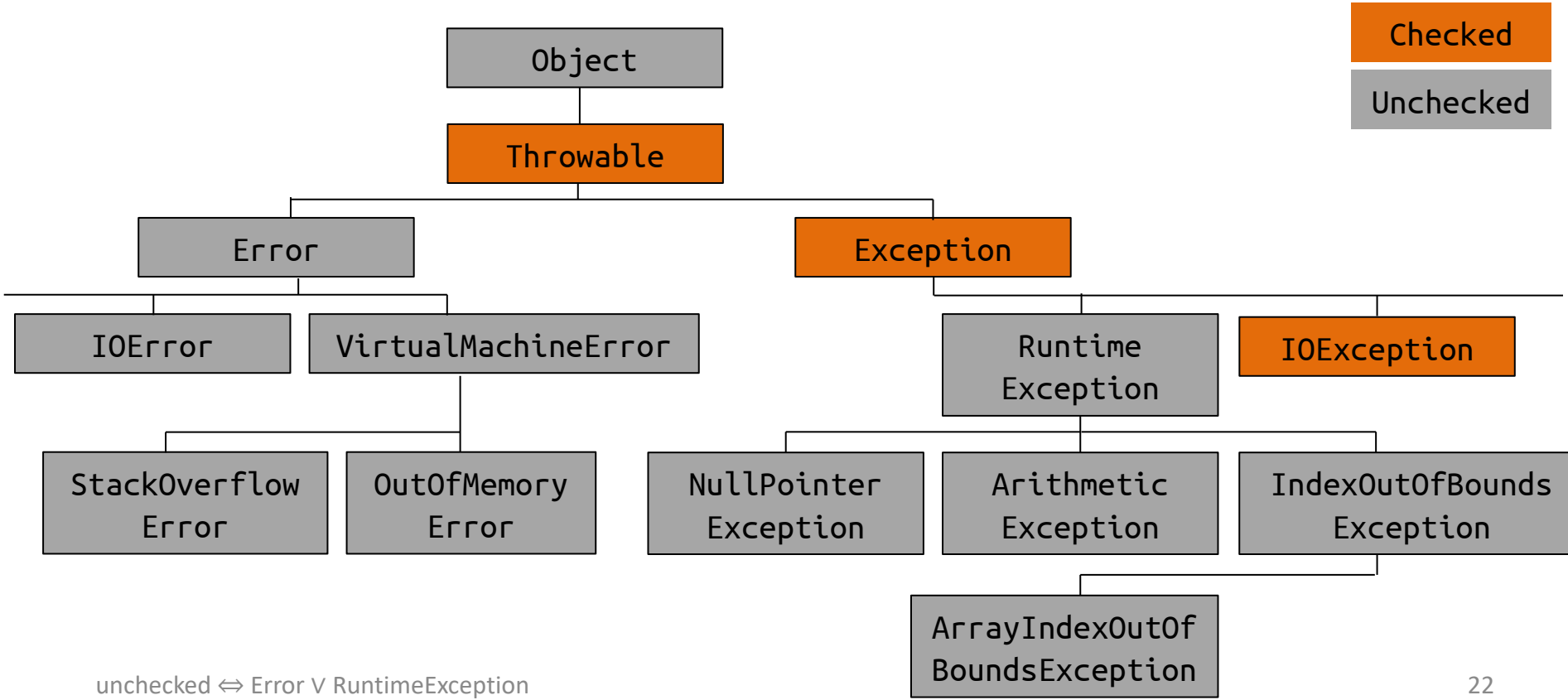
Sprachdesign

- Wirft eine Methode evtl. eine Exception, sollten Klienten dies wissen
  → spricht für checked Exceptions
- Auf manche Exceptions können Klienten aber oft nicht sinnvoll reagieren
  - Programmierfehler, z.B. `ArrayIndexOutOfBoundsException`
  - OS-/Hardwarefehler, z.B. `OutOfMemoryError`
- Java unterscheidet daher zwischen
  - Checked Exceptions für Fehler, auf die Aufrufer wahrscheinlich sinnvoll reagieren können
  - Unchecked Exceptions, für die anderen Fälle

Bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

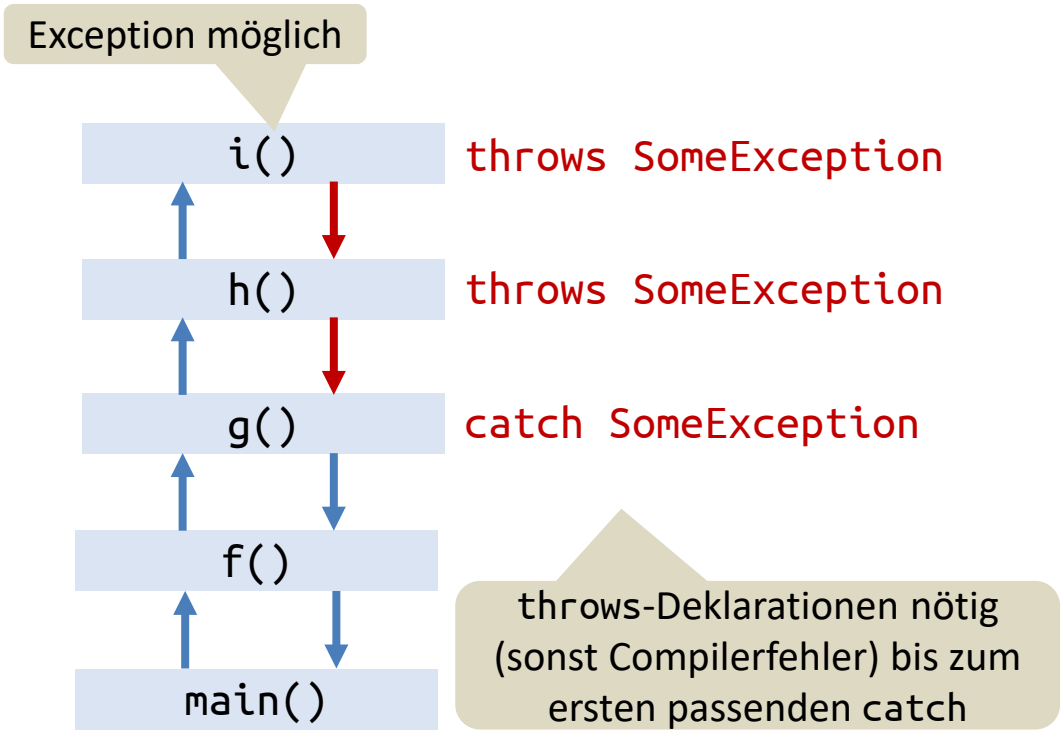
Guideline: https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

Vererbungshierarchie für Exceptions (Auszug)



unchecked ⇔ Error V RuntimeException

Checked Exceptions und Methodenaufrufe




```
void i() throws SomeException {
  …
}
```

```
void h() throws SomeException {
  … i() …
}
```

```
void g(){
  try { … h() … }
  catch (SomeException e) { … }
}
```

```
void f() {
  … g() …
}
```

```
void main(…) {
  … f() …
}
```

`throws` mit mehreren Exceptions

`throws`-Deklaration kann mehrere Exceptions ankündigen

```
void playAudioFile(File file)
    throws FileNotFoundException,
           DataFormatException,
           UnsupportedAudioFileException {
  …
}
```

12. Exceptions

Werfen von Exceptions

- Java wirft Exceptions bei
  - Bestimmten Operationen, z.B. `null.foo()` ➜ `NullPointerException`
  - Methoden der Standardbibliothek,
    z.B. `arraylist.get(-1)` ➜ `IndexOutOfBoundsException`
- Programm kann selbst Exceptions werfen, z.B.

```
throw new IllegalArgumentException (
    "Favourite course must be one of {EProg, EProg}");
```

- Es können auch eigene Exception-Typen definiert werden:
  neue Subklassen von `Throwable`, `Exception`, `RuntimeException`, …

Beispiele: Ungültiges Argument

```
import java.lang.IllegalArgumentException
```

```
void printAge(int age) {
  if (age < 0) {
    throw new IllegalArgumentException("negative age");
  } else {
    System.out.println("You are " + age +  "years old.");
  }
}
```

```
class Person {
  String name; // INV: neither null nor ""

  Person(String name) {
    if (name == null) throw new IllegalArgumentException("name must not be null");
    if (name == "") throw new IllegalArgumentException("name must not be empty");
    this.name = name;
  }
}
```

Beispiel: Eigene Exception-Klasse

Eigene Exception deklarieren …
(extends Exception → checked,
extends RuntimeException → unchecked)

```
public class PasswordPolicyViolationException extends Exception {
  public PasswordPolicyViolationException(String message) {
    super(message);
  }
}
```

```
void setPassword(String password) throws PasswordPolicyViolationException {
  if (password.length() < 12)
    throw new PasswordPolicyViolationException("Password too short");
  if (/* password does not contain special characters */)
    throw new PasswordPolicyViolationException("Password must contain …");
  …
}
```

… und werfen

Exceptions sind auch «nur» Klassen

```
public class PasswordPolicyViolationException extends Exception {
  private String password;
  private String offence;

  public PasswordPolicyViolationException(String password, String offence) {
    super("Password '" + password + "' not acceptable: " + offence); // Setzt this.message
    this.password = password;
    this.offence = offence;
  }

  // getPassword() etc.
}
```

Exception mit zusätzlichen Details

Wirft evtl. PasswordPolicyViolationException

```
try { setPassword(newPassword) }
catch (PasswordPolicyViolationException e) {
  System.out.println("Rejected password: " + e.getPassword());
  System.out.println("Reason: " + e.getOffence());
}
```

12. Exceptions

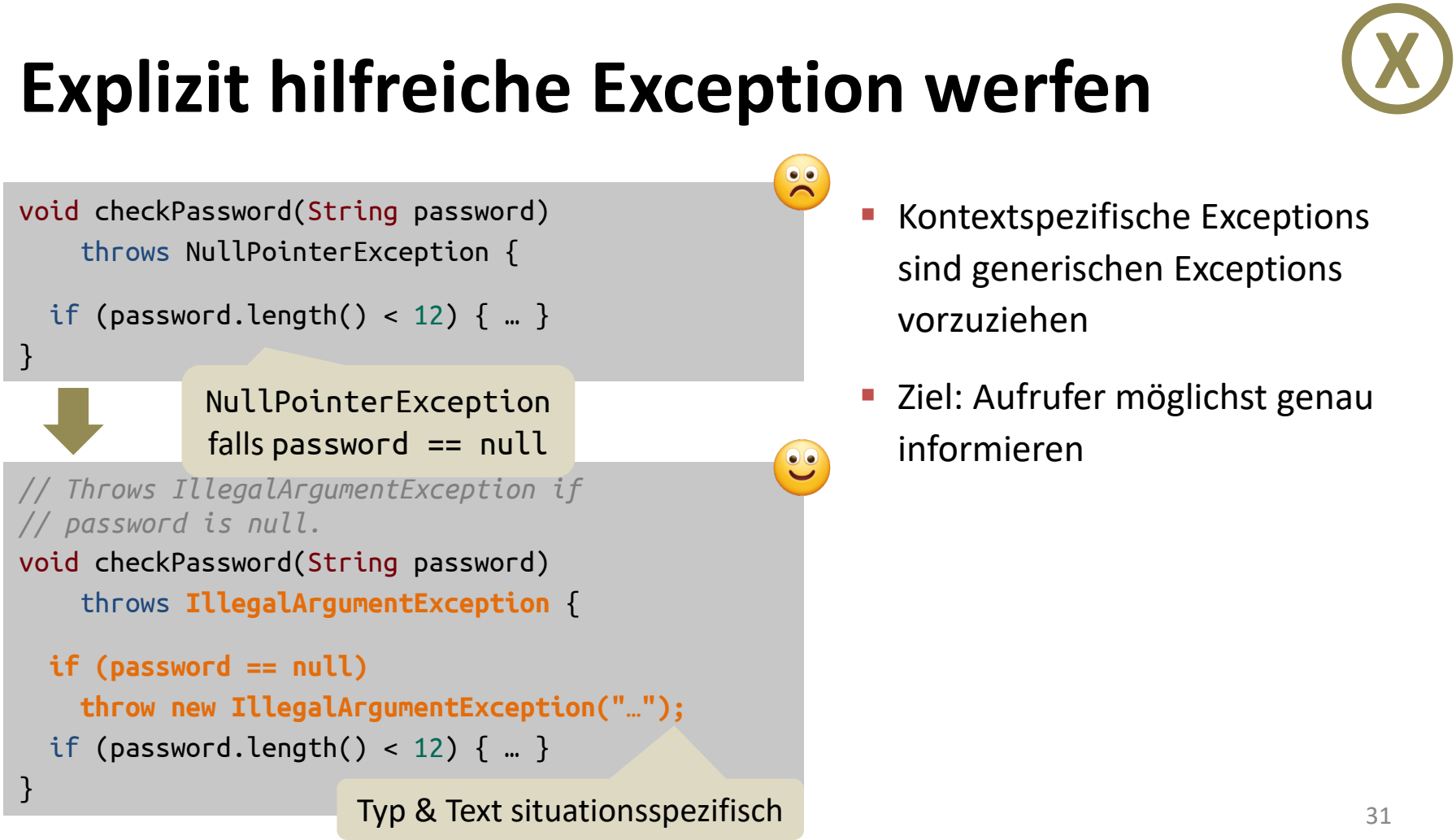
12.1 Exceptions fangen

12.2 Checked vs. Unchecked Exceptions

12.3 Exceptions werfen

12.4 Best Practices ⓧ

Explizit hilfreiche Exception werfen



```
void checkPassword(String password)
    throws NullPointerException {

  if (password.length() < 12) { … }
}
```

NullPointerException falls password == null

```
// Throws IllegalArgumentException if
// password is null.
void checkPassword(String password)
    throws IllegalArgumentException {

  if (password == null)
    throw new IllegalArgumentException("…");
  if (password.length() < 12) { … }
}
```

Typ & Text situationsspezifisch

- Kontextspezifische Exceptions sind generischen Exceptions vorzuziehen
- Ziel: Aufrufer möglichst genau informieren

Exceptions nie «verschlucken»

```
try {
  … new Scanner(file);
} catch (FileNotFoundException e) {}
```

```
try {
  … new Scanner(file);
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
```

- (Checked) Exception nicht bloss fangen, damit der Compiler Ruhe gibt
- Fehlermeldung verschwindet stillschweigend → erschwert das Melden & Debuggen von Problemen

Exceptions separat fangen

```
try {
  myCode();
} catch (Exception t) {
  …
}
```

```
try {
  myCode();
} catch (ConcreteException1 e1) {
  …
} catch (ConcreteException2 e2) {
  …
}
```

- Auf konkrete Exceptions kann i.d.R. besser reagiert werden (spezifischere Fehlermeldung, zusätzliche Informationen)
- Wird `myCode()` weiterentwickelt, wirft es evtl. eine weitere Exception. Diese Änderung bleibt bei «catch-all» (oben) evtl. unbemerkt und erschwert so das Debuggen.

Exceptions vermeiden ist besser als fangen

```
while (true) {
  try {
    …print("Divisor d = ");
    int d = console.nextInt();
    return x / d;
  } catch (ArithmeticException e) {
    …println("d mustn't be 0");
  }
}
```

```
while (true) {
  …print("Divisor d = ");
  int d = console.nextInt();
  if (d == 0) { …println("d mustn't be 0");
  else { return x / d; }
 }
```

- Intention und Kontrollfluss i.d.R. klarer, wenn problematische Situation mit `if-else` verhindert wird (statt darauf mit `catch` zu reagieren).
- Exceptions verursachen vergleichsweise viel Arbeit zur Laufzeit, relativ zu `if-else` (nur relevant, falls der Ausnahmefall oft eintritt)