

# 9. Exceptions

---

Harald Gall, Prof. Dr.

Institut für Informatik

Universität Zürich

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



University of Zurich  
Department of Informatics



---

# Ziele

- Den Grundgedanken der Ausnahmebehandlung beschreiben können
- Ausnahmen (Exceptions) korrekt behandeln können

---

# Erwartetes & Unerwartetes

Wir erwarten einen Erfolg für

```
int numberInStock = kb.nextInt();
```

Aber wir erwarten nicht, dass

- eine Datei auf einem Laufwerk gespeichert ist, das irrtümlicherweise entfernt worden ist
- eine Netzwerk-Verbindung plötzlich ausgefallen ist
- eine Datei auf einer Festplatte gespeichert ist, die wegen Defekts ausgefallen ist

---

# 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 ausgelegt, dass sie diese unerwarteten Ereignisse und Eingaben berücksichtigt und **korrekt** darauf **reagiert**.
- Reagieren auf *Unerwartetes* ist mindestens so essentiell wie die Problemstellung per se.

---

# 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.**

---

# Beispiel:

```
public int methodY() throws SomeException {  
    // ...  
    throw new SomeException("Something went wrong!");  
    // ...  
}
```

```
public void methodX() {  
    try {  
        int result = methodY();  
        // more statements...  
    } catch (SomeException e) {  
        // error handling code...  
    }  
}
```

---

---

# Exceptions in Java

## **try** Block

- Enthält “gefährliche” Codezeilen, sprich solche, wo potentiell etwas schief gehen kann
- Wenn etwas schief geht, dann wird eine Exception geworfen

## **catch** Block

- Wenn eine Exception geworfen wird, dann wird der **catch** Block ausgeführt
  - Vergleichbar mit einer Methode mit einem Parameter (das geworfene Objekt)
-

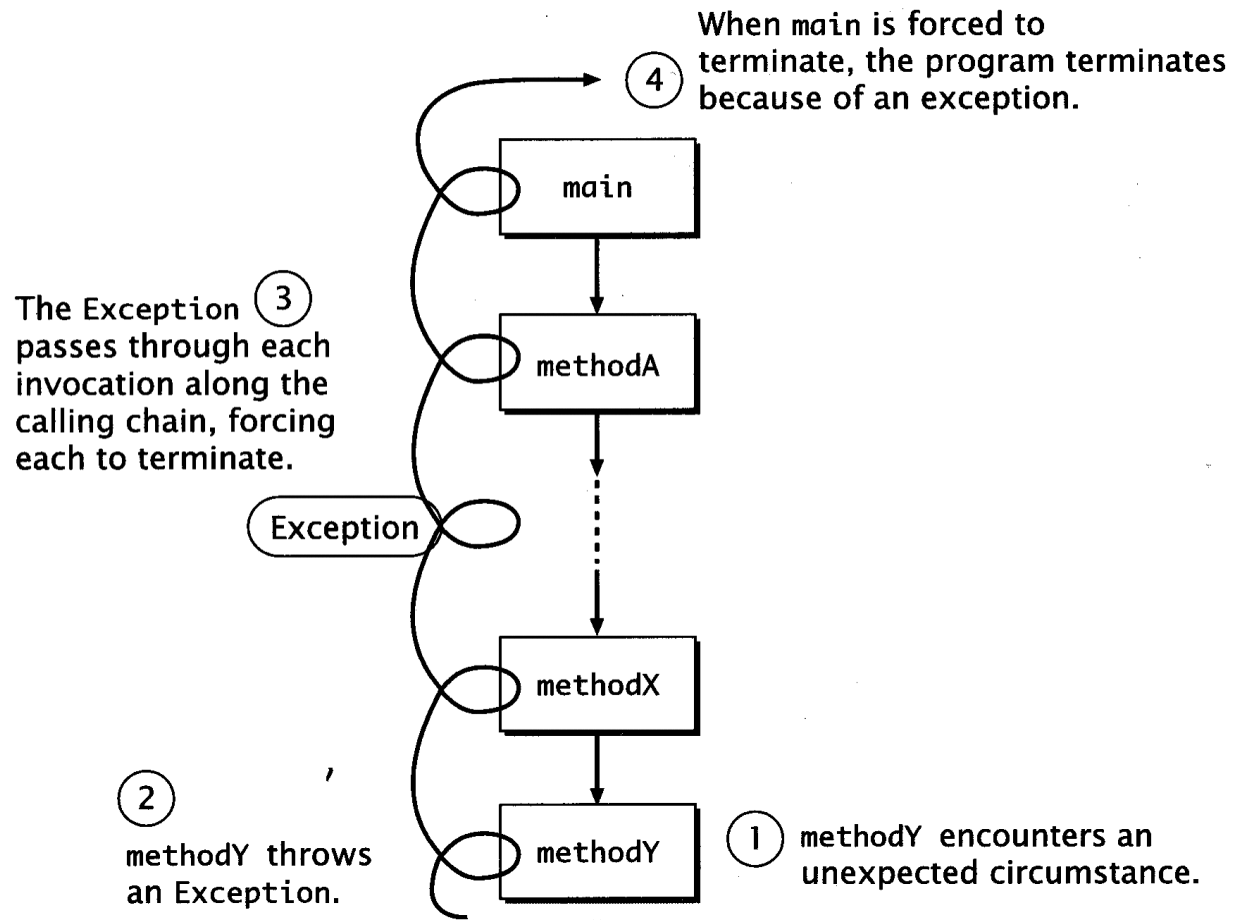


---

# 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.

# Chain of method calls



---

# 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.

---

# Throw and Catch

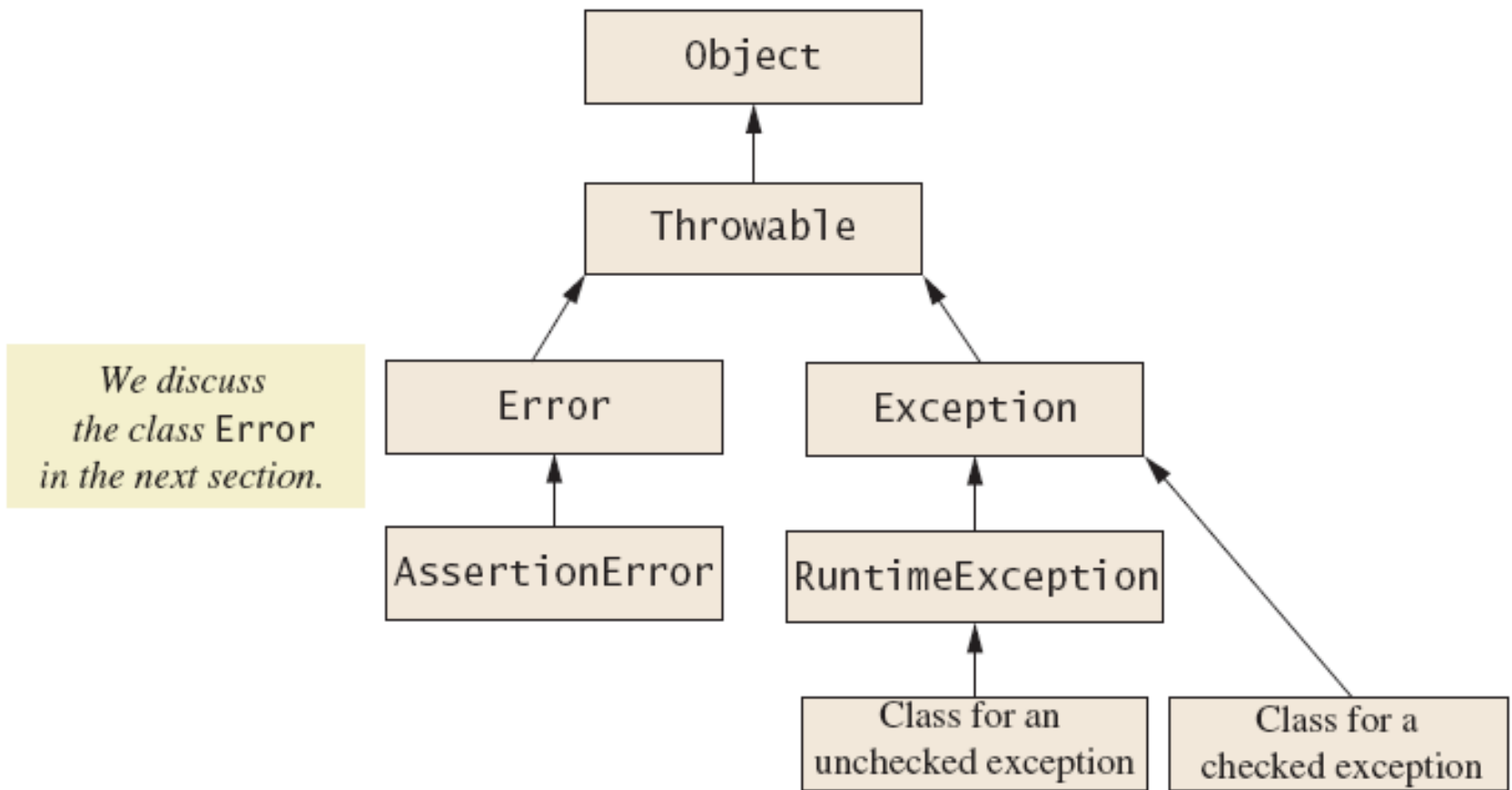
- Ein try Block kann eine beliebige Anzahl Exceptions verschiedener Typen werfen
- Jeder catch Block kann nur Exceptions eines bestimmten Typs behandeln (ausser Multi-Catch)
- Die Reihenfolge von Catch Blocks ist relevant

---

# Der `finally` Block

- Optional kann man einen `finally` Block nach einer Sequenz von `catch` Blocks definieren
- Code im `finally` Block wird immer ausgeführt
  - Egal ob Exception geworfen wird oder nicht
  - Egal ob ein `catch` Block existiert oder nicht

# Typen von Exceptions



---

# Typen von Exceptions

- *Checked* exception
  - Muss in **catch** Block behandelt werden
  - Oder im **throws** Clause deklariert werden
- *Unchecked* exception
  - Auch *run-time* exception genannt
  - Muss nicht im **catch** Block behandelt oder im **throws** Clause deklariert werden

---

# Typen von Exceptions

- Beispiele, wieso unchecked Exceptions geworfen werden:
  - Array index out of bounds
  - Teilen durch Null
- Unbehandelte runtime Exception terminiert Programmausführung



---

# Errors

- Ein *error* ist ein Objekt der Klasse **Error**
  - Vergleichbar mit einer unchecked exception
  - Muss nicht gefangen oder deklariert werden
- Errors sind ausserhalb des Einflussbereichs der Programmierer

---

# Zusammenfassung

- Wie ein Objekt unerwartete Ereignisse behandelt gehört zu seinem Verhalten.
- Java liefert eine Klasse `Exception`, die unerwartete Situationen modelliert.
  - Es gibt ein `throws` Statement, um den Kontrollfluss abrupt zu ändern.
  - Dieses `throw` Statement wird im ersten umschliessenden `try - catch` Block für diese Exception behandelt
  - Gibt es keine Behandlung, dann wird die Exception entlang der Aufrufer-Kette weitergeleitet (bis das Programm terminiert).