6. Java Fehler und Ausnahmen

Fehler, Systemausnahmen, Benutzerausnahmen, Behandeln von Ausnahmen, Spezialfall Ressourcen

Fehler und Ausnahmen in Java

Fehler und Ausnahmen unterbrechen die normale Programmausführung abrupt und stellen ein *nicht geplantes Ereignis* dar.

Ausnahmen sind böse, oder doch nicht?

- Java ermöglicht es, solche Ereignisse abzufangen und zu behandeln (als Alternative zum Programmabsturz).
- Nicht behandelte Fehler und Ausnahmen werden durch den Aufrufstapel hochgereicht.

Fehler (Errors)

Hier ist nichts mehr zu machen

Fehler treten in der virtuellen Machine von Java auf und sind *nicht reparierbar*.

Beispiele

- Kein Speicher mehr verfügbar
- Zu hoher Aufrufstapel ($\rightarrow$ Rekursion)
- Fehlende Programmbibliotheken
- Bug in der virtuellen Machine
- Hardwarefehler

Ausnahmen (Exceptions)

Ausnahmen werden von der virtuellen Machine oder vom Programm selbst ausgelösst und können meist behandelt werden um die *Normalsituation wiederherzustellen*

Aufwischen und neu einschenken

Beispiele

- Dereferenzierung von `null`
- Division durch 0
- Schreib/Lesefehler (Dateien)
- Businesslogik Fehler

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

Beispiel einer Systemausnahme

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] args){
4          int i = readInt("Number");
5      }
6      private static int readInt(String prompt){
7          System.out.print(prompt + ": ");
8          Scanner input = new Scanner(System.in);
9          return input.nextInt();
10     }
11 }
```

Eingabe: `Number: asdf`

Nicht behandelte Fehler und Ausnahmen

Das Programm stürzt ab und hinterlässt auf der Konsole eine “Aufrufstapelzurückverfolgung” ☺ (ab jetzt: *Stacktrace*). Darin sehen wir, wo genau das Programm abgebrochen wurde.

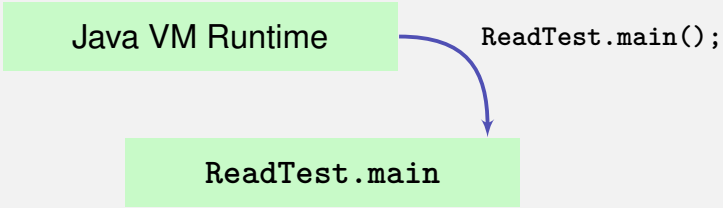
```
Exception in thread "main" java.util.InputMismatchException
        [...]
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at ReadTest.readInt(ReadTest.java:9)
    at ReadTest.main(ReadTest.java:4)
```

$\Rightarrow$ Forensische Nachforschungen mit Hilfe dieser Information.

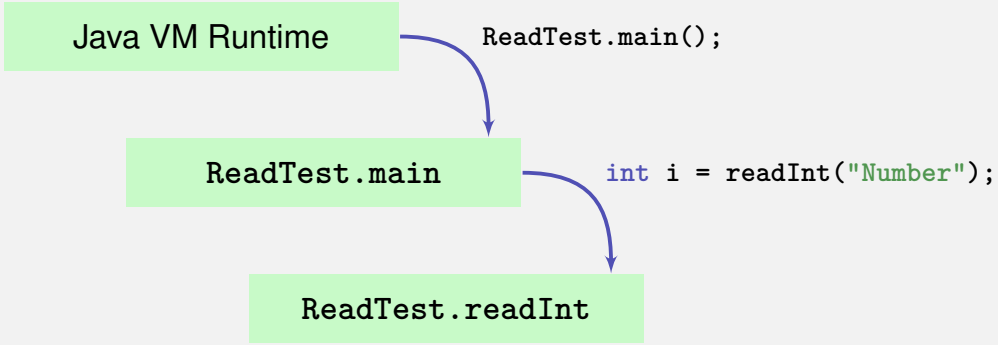
Ausnahme propagiert durch Aufrufstapel

Java VM Runtime

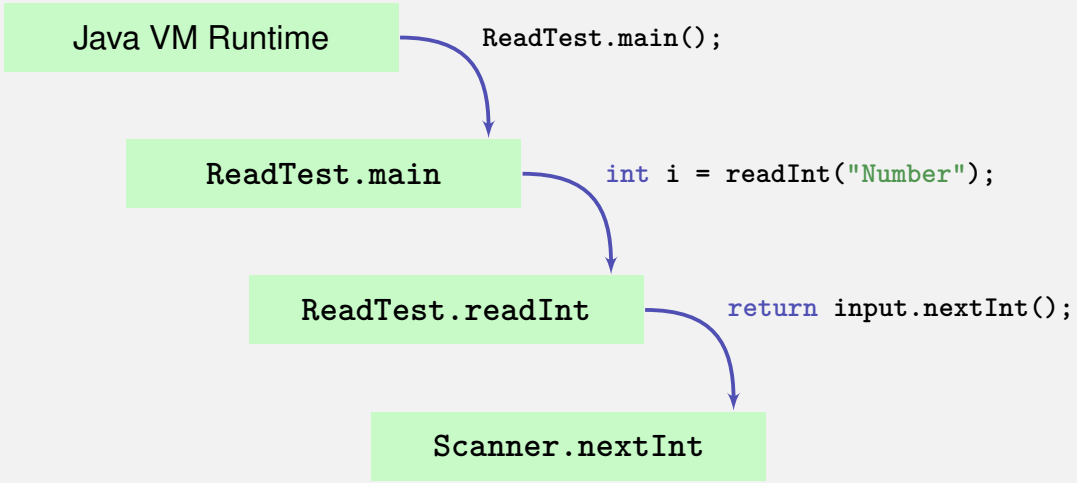
Ausnahme propagiert durch Aufrufstapel



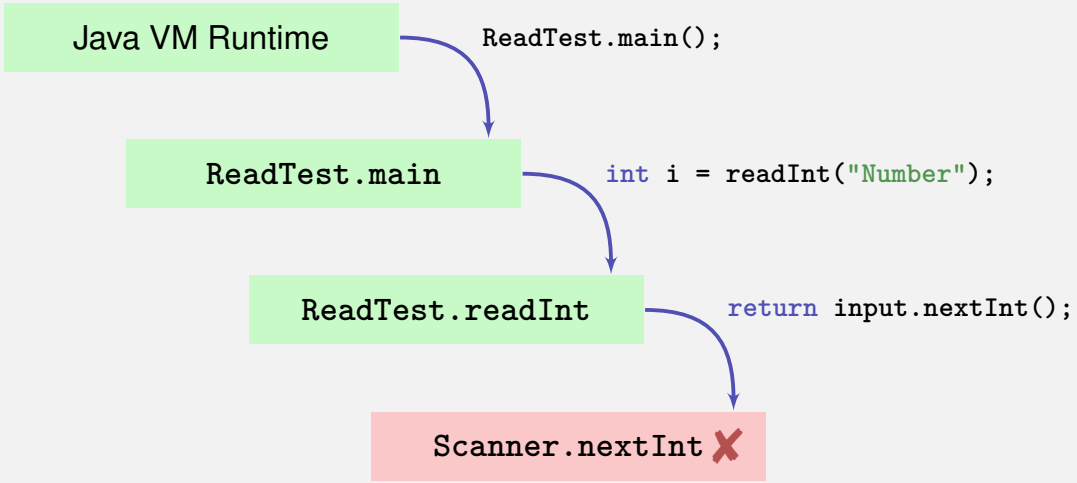
Ausnahme propagiert durch Aufrufstapel



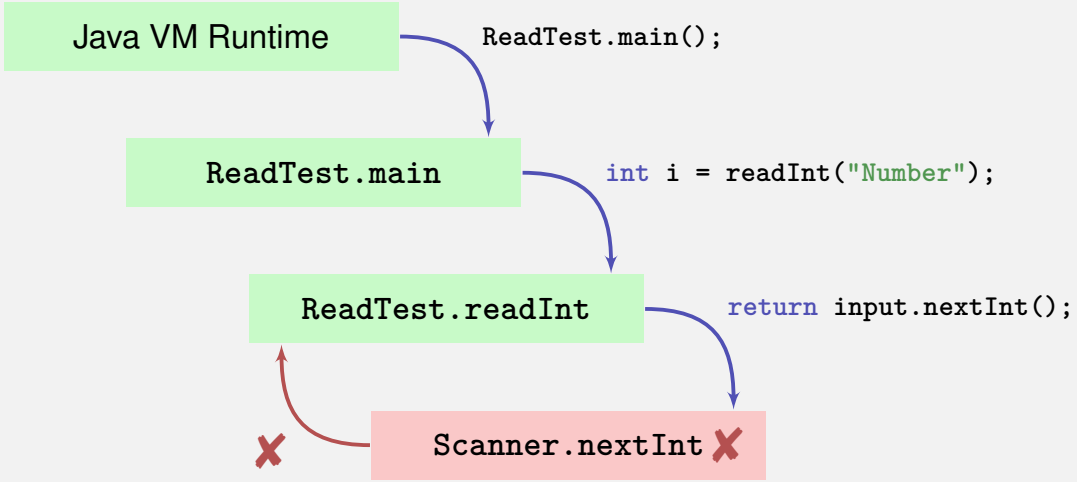
Ausnahme propagiert durch Aufrufstapel



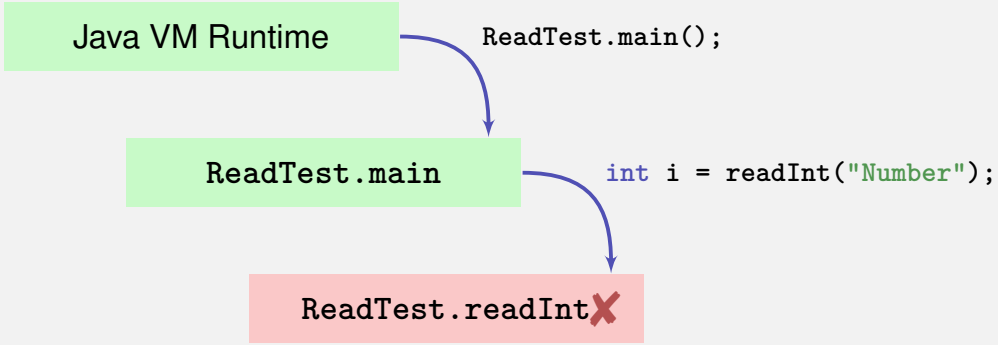
Ausnahme propagiert durch Aufrufstapel



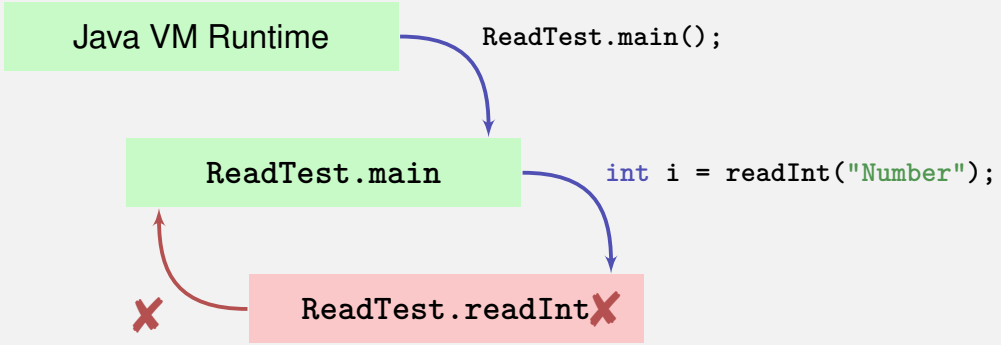
Ausnahme propagiert durch Aufrufstapel



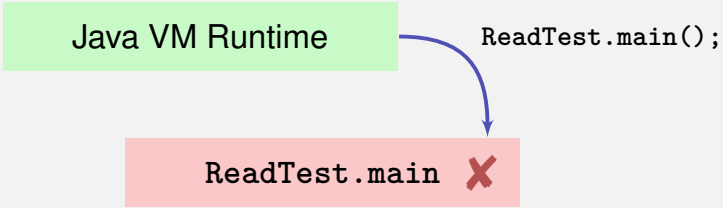
Ausnahme propagiert durch Aufrufstapel



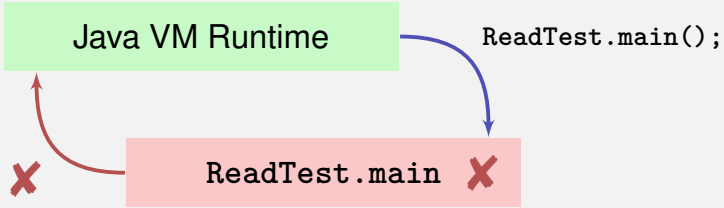
Ausnahme propagiert durch Aufrufstapel



Ausnahme propagiert durch Aufrufstapel



Ausnahme propagiert durch Aufrufstapel



Ausnahme propagiert durch Aufrufstapel

Java VM Runtime †

Stacktraces verstehen

Ausgabe:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Stacktraces verstehen

Eine unpassende Eingabe ...

Ausgabe:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Stacktraces verstehen

Ausgabe:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Stacktraces verstehen

Ausgabe:

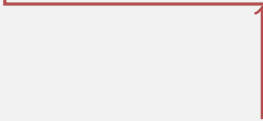
```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

... in Methode `readInt` auf Zeile 9 ...

Stacktraces verstehen

Ausgabe:

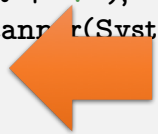
```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```



... aufgerufen durch Methode `main` auf Zeile 4.

Stacktraces verstehen

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] args){
4          int i = readInt("Number");
5      }
6      private static int readInt(String prompt){
7          System.out.print(prompt + ": ");
8          Scanner input = new Scanner(System.in);
9          return input.nextInt();
10     }
11 }
```



```
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

Stacktraces verstehen

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] ar){
4          int i = readInt("Number");
5      }
6      private static int readInt(String prompt){
7          System.out.print(prompt + ": ");
8          Scanner input = new Scanner(System.in);
9          return input.nextInt();
10     }
11 }
```

```
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

Systemausnahme: Bug im Programm?!

Wo ist der Fehler?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Systemausnahme: Bug im Programm?!

Wo ist der Fehler?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Nicht garantiert, dass als nächstes ein `int` *anliegt.*

Systemausnahme: Bug im Programm?!

Wo ist der Fehler?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```


Nicht garantiert, dass als nächstes ein `int` *anliegt.*

$\Rightarrow$ Die Scanner Klasse bietet ein Test dafür an

Systemausnahme: Bugfix!

Erst prüfen!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

Erste Erkenntnis: Oft keine Ausnahmesituation

Oft sind die “Sonderfälle” gar kein besonderes Ereignis, sondern absehbar. Hier sollten *keine* Ausnahmen verwendet werden!

Kinder kippen Becher um.
Man gewöhnt sich daran.

Beispiele

- Falsche Credentials beim Einloggen
- Leere Pflichtfelder in Eingabemasken
- Nicht verfügbare Internet-Ressourcen
- Timeouts

Zweite Erkenntnis: Ausnahmen verhindern

Problem gelösst.

Statt eine Systemausnahme abzuwarten *aktiv verhindern*, dass diese überhaupt auftreten kann.

Beispiele

- Usereingaben frühzeitig prüfen
- Optionale Typen verwenden
- Timeout Situationen voraussehen
- Plan B für nicht verfügbare Ressourcen

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

Arten von Ausnahmen

Systemausnahmen
(runtime exceptions)

- Können überall auftreten
- *Können* behandelt werden
- Ursache: Bug im Programm

Benutzerausnahmen
(checked exceptions)

- Müssen deklariert werden
- *Müssen* behandelt werden
- Ursache: Unwahrscheinliches, aber prinzipiell mögliches Ereignis

Beispiel einer Benutzerausnahme

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Beispiel einer Benutzerausnahme

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Compiler Fehler:

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be
    FileReader fr = new FileReader(filename);
                    ^
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
                               ^
2 errors
```

Kurzer Blick in die Javadoc

readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

Warum eine Benutzerausnahme?

Folgende Situationen rechtfertigen Benutzerausnahmen:

- Fehlerfall ist *unwahrscheinlich aber prinzipiell möglich* – und kann durch geeignete Massnahmen zur Laufzeit behoben werden können.

Warum eine Benutzerausnahme?

Folgende Situationen rechtfertigen Benutzerausnahmen:

- Fehlerfall ist *unwahrscheinlich aber prinzipiell möglich* – und kann durch geeignete Massnahmen zur Laufzeit behoben werden können.

Der Aufrufer einer Methode mit einer deklarierten Benutzerausnahme wird gezwungen, sich damit zu beschäftigen – behandeln oder weiterreichen.

Behandeln von Ausnahmen

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

Geschützter Bereich

Behandeln von Ausnahmen

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

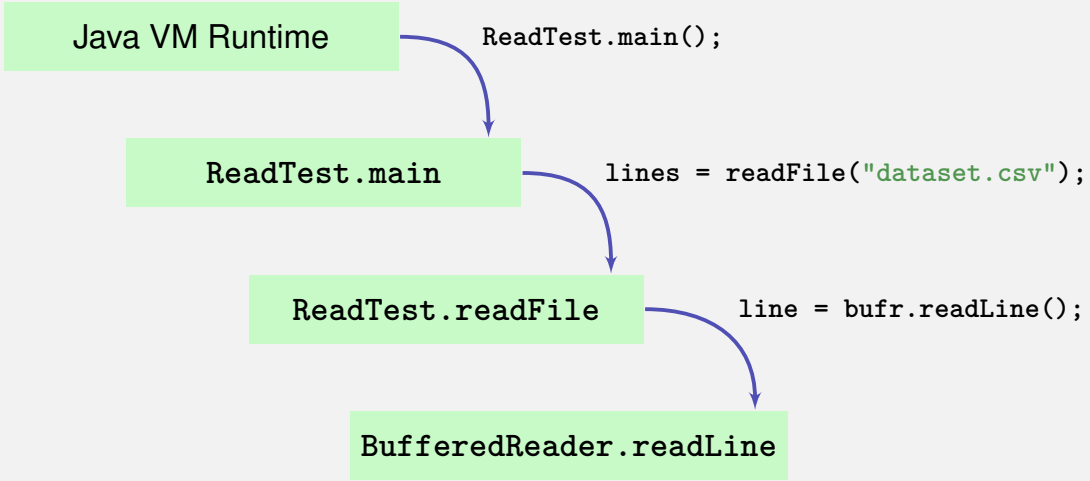
Massnahmen zur Wiederherstellung der Normalsituation

Behandeln von Ausnahmen

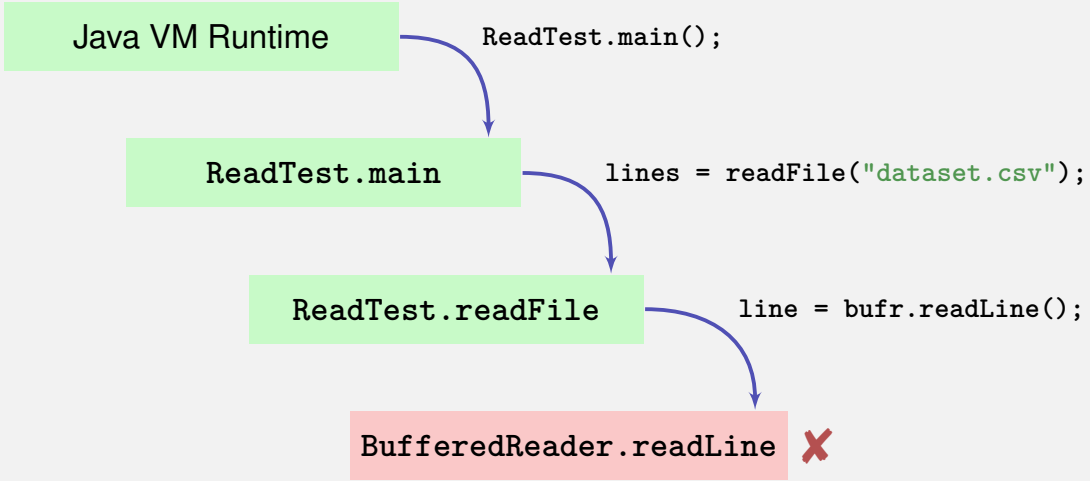
```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

Wird in jedem Fall am Schluss ausgeführt, immer!

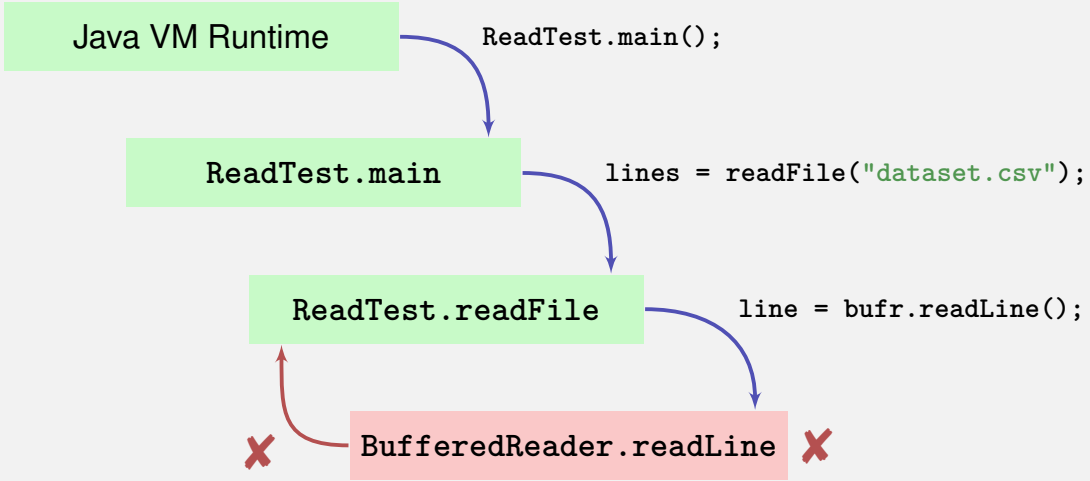
Behandlung von Ausnahmen: Propagieren stoppen!



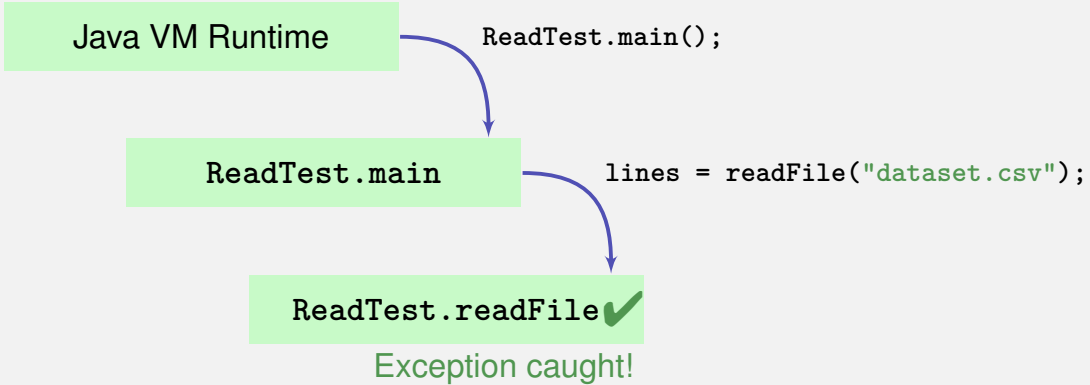
Behandlung von Ausnahmen: Propagieren stoppen!



Behandlung von Ausnahmen: Propagieren stoppen!



Behandlung von Ausnahmen: Propagieren stoppen!



Finally: Resourcen schliessen!

In Java müssen *Resourcen* unbedingt geschlossen werden nach Gebrauch. Ansonsten wird Speicher nicht freigegeben.

Resourcen:

- Dateien
- Datenströme
- GUI Elemente
- ...

Try-With-Resources Anweisung

Spezifische Syntax, um Ressourcen *automatisch* zu schliessen:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
          BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Try-With-Resources Anweisung

Spezifische Syntax, um Ressourcen *automatisch* zu schliessen:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
          BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Resourcen werden hier geöffnet

Try-With-Resources Anweisung

Spezifische Syntax, um Ressourcen *automatisch* zu schliessen:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
          BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Resourcen werden hier automatisch geschlossen

7. Funktionale Konzepte in Java

Funktionale Programmierung, Lambda Ausdrücke, Datenströme, Pipelines

Funktionale vs. Imperative Programmierung

Imperative Konzepte

Funktionale Konzepte

Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen

Funktionale Konzepte

- Evaluierung von Ausdrücken

Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen
- Zustand (z.B. Felder)

Funktionale Konzepte

- Evaluierung von Ausdrücken
- Kein Zustand

Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen
- Zustand (z.B. Felder)
- Änderbare Datentypen

Funktionale Konzepte

- Evaluierung von Ausdrücken
- Kein Zustand
- Immutable Datentypen

Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen
- Zustand (z.B. Felder)
- Änderbare Datentypen
- Fokus auf Datenstrukturen

Funktionale Konzepte

- Evaluierung von Ausdrücken
- Kein Zustand
- Immutable Datentypen
- Fokus auf Datenströme

Funktionale vs. Imperative Programmierung

Imperative Konzepte

- Ausführen von Anweisungen
- Zustand (z.B. Felder)
- Änderbare Datentypen
- Fokus auf Datenstrukturen
- Fokus auf das “wie”

Funktionale Konzepte

- Evaluierung von Ausdrücken
- Kein Zustand
- Immutable Datentypen
- Fokus auf Datenströme
- Fokus auf das “was”

Beispiel: Einlesen von Dateien - Imperativ

```
try (BufferedReader br=new BufferedReader(new FileReader("data.csv"))){
    LinkedList<Measurement> result = new LinkedList<>();
    br.readLine();
    String line;
    while ((line = br.readLine()) != null){
        Measurement m = new Measurement(line);
        result.add(m);
    }
    return result;
}
```

Beispiel: Einlesen von Dateien - Funktional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {




}
```

Beispiel: Einlesen von Dateien - Funktional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {

    return stream.skip(1).map(Measurement::new).collect(toList());

}
```

Streams - Datenströme

In Java sind *Streams* die Basis für funktionale Programmierung.
Quellen von Datenströmen:

- Dateien
- Arrays
- Datenstrukturen
- . . .

Streams - Datenströme

In Java sind *Streams* die Basis für funktionale Programmierung.
Quellen von Datenströmen:

- Dateien
- Arrays
- Datenstrukturen
- . . .

Beispiel

```
Stream<String> stream = Files.lines (...))
```

Operationen auf Datenströmen: Map

Map: Anwenden von Funktionen auf den einzelnen Elementen des Streams

- Mathematische Berechnungen
- Erstellen neuer Objekte basierend auf den existierenden Elementen.
- . . .

Operationen auf Datenströmen: Map

Map: Anwenden von Funktionen auf den einzelnen Elementen des Streams

- Mathematische Berechnungen
- Erstellen neuer Objekte basierend auf den existierenden Elementen.
- . . .

Beispiel

```
map(Measurement::new)
```

Operationen auf Datenströmen: Reduce

Reduce: Aggregierung der einzelnen Elemente eines Datenstroms zu einem einzelnen Wert.

- Statistische Aggregationen
- Ablegen der Elemente in Datenstrukturen
- . . .

Operationen auf Datenströmen: Reduce

Reduce: Aggregierung der einzelnen Elemente eines Datenstroms zu einem einzelnen Wert.

- Statistische Aggregationen
- Ablegen der Elemente in Datenstrukturen
- . . .

Beispiel

```
collect(toList())
```

Beispiel: Suchen von Daten - Imperativ

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();

for (Measurement m : data){
    if (m.position.near(ref)){
        System.out.println(m.originalLine);
    }
}
```

Beispiel: Suchen von Daten - Funktional

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();

data.stream()
        .filter(m -> ref.near(m.position))
        .forEach(System.out::println);
```

Operationen auf Datenströmen: Filter

Filter: Herausfiltern einzelner Elemente des Streams.

- Illegale Werte entfernen
- Auswahl von Werten basierend auf Anfragen
- . . .

Operationen auf Datenströmen: Filter

Filter: Herausfiltern einzelner Elemente des Streams.

- Illegale Werte entfernen
- Auswahl von Werten basierend auf Anfragen
- . . .

Beispiel

```
filter (m -> ref.near(m.position))
```

Operationen auf Datenströmen: Seiteneffekte

Seiteneffekte: Der nicht-funktionale Aspekt: Ausführen von beliebigen Operationen aufgrund einzelner Elemente.

- Input/Ouput
- Datenstrukturen updaten
- . . .

Operationen auf Datenströmen: Seiteneffekte

Seiteneffekte: Der nicht-funktionale Aspekt: Ausführen von beliebigen Operationen aufgrund einzelner Elemente.

- Input/Ouput
- Datenstrukturen updaten
- . . .

Beispiel

```
forEach(System.out::println)
```

Funktionalität als Parameter

Operationen auf dem Stream haben *Funktionalität* (Code) als Parameter, anstelle von *Daten*

Funktionalität als Parameter

Operationen auf dem Stream haben *Funktionalität* (Code) als Parameter, anstelle von *Daten*

Möglichkeiten, Funktionalität zu übergeben (statt Daten):

- “lose” Code-Stücke
- Referenzen auf Methoden
- Referenzen auf Konstruktoren

Wie können wir dies bewerkstelligen?

Lambda Ausdrücke

Lambda Ausdrücke sind im Wesentlichen Methoden ohne Namen.

Normale Methode

```
double discriminant(double a, double b, double c){
    return b*b - 4*a*c;
}
```

Lambda Ausdrücke

Lambda Ausdrücke sind im Wesentlichen Methoden ohne Namen.

Normale Methode

```
double discriminant(double a, double b, double c){
    return b*b - 4*a*c;
}
```

Gleichwertiger Lambda Ausdruck

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Lambda Ausdrücke

Lambda Ausdruck

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Lambda Ausdrücke

Lambda Ausdruck

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Ohne explizite Typdeklaration der Parameter

```
(a, b, c) -> {
    return b*b - 4*a*c;
}
```

Lambda Ausdrücke

Lambda Ausdruck

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Ohne explizite Typdeklaration der Parameter

```
(a, b, c) -> {
    return b*b - 4*a*c;
}
```

Mit einem einzelnen Ausdruck statt einem Block

```
(a, b, c) -> b*b - 4*a*c
```

Lambda Ausdruck im Beispiel

Beispiel

```
filter (m -> ref.near(m.position))
```

Lambda Ausdruck im Beispiel

Beispiel

```
filter (m -> ref.near(m.position))
```

- Die Methode `filter` erwartet eine Methode als Parameter, welche einen `Measurement` entgegennimmt, und einen `boolean` zurueckgibt.

Lambda Ausdruck im Beispiel

Beispiel

```
filter (m -> ref.near(m.position))
```

- Die Methode `filter` erwartet eine Methode als Parameter, welche einen `Measurement` entgegennimmt, und einen `boolean` zurueckgibt.
- `m` ist ein Parameter vom Typ `Measurement` ✓
- `ref.near(m.position)` ist ein einzelner `boolean`Ausdruck ✓

Lambda Ausdruck im Beispiel

Beispiel

```
filter (m -> ref.near(m.position))
```

- Die Methode `filter` erwartet eine Methode als Parameter, welche einen `Measurement` entgegennimmt, und einen `boolean` zurueckgibt.
- `m` ist ein Parameter vom Typ `Measurement` ✓
- `ref.near(m.position)` ist ein einzelner `boolean`Ausdruck ✓
- Die Variable `ref` aus dem definierenden Kontext ist zugänglich, solange sie *effektiv* konstant (`final`) ist.

Referenzen auf Methoden

Um eine Methode auf einem Objekt *auszuführen*, schreiben wir:

```
object.method()
```

Referenzen auf Methoden

Um eine Methode auf einem Objekt *auszuführen*, schreiben wir:

```
object.method()
```

Um eine *Referenz* auf eine Methode auf einem Objekt anzugeben, schreiben wir:

```
object::methode
```

Referenzen auf statische Methoden

Um eine statische Methode *auszuführen*, schreiben wir:

```
Clazz.method()
```

Referenzen auf statische Methoden

Um eine statische Methode *auszuführen*, schreiben wir:

```
Clazz.method()
```

Um eine *Referenz* auf eine statische Methode anzugeben, schreiben wir:

```
Clazz::method
```

Referenz auf eine Methode im Beispiel

Beispiel

```
forEach(System.out::println)
```

Referenz auf eine Methode im Beispiel

Beispiel

```
forEach(System.out::println)
```

- Die Methode `forEach` erwartet eine Methode, welche nichts zurück gibt und ein Argument vom Typ `Measurement` akzeptiert.

Referenz auf eine Methode im Beispiel

Beispiel

```
forEach(System.out::println)
```

- Die Methode `forEach` erwartet eine Methode, welche nichts zurück gibt und ein Argument vom Typ `Measurement` akzeptiert.
- Die Methode `println` auf Objekt `out` erfüllt diese Eigenschaften. ✓

Referenzen auf Konstruktoren

Um einen Konstruktur einer Klasse *auszuführen*, schreiben wir:

```
new Clazz()
```

Referenzen auf Konstruktoren

Um einen Konstruktur einer Klasse *auszuführen*, schreiben wir:

```
new Clazz()
```

Um eine *Referenz* auf einen Konstruktor anzugeben, schreiben wir:

```
Clazz::new
```

Referenz auf einen Konstruktor im Beispiel

Beispiel

```
map(Measurement::new)
```

Referenz auf einen Konstruktor im Beispiel

Beispiel

```
map(Measurement::new)
```

- Die Methode `map` erwartet eine Methode, welche ein Objekt eines gewissen Datentyps zurueckgibt (egal welcher) und ein Argument vom Typ `String` akzeptiert.

Referenz auf einen Konstruktor im Beispiel

Beispiel

```
map(Measurement::new)
```

- Die Methode `map` erwartet eine Methode, welche ein Objekt eines gewissen Datentyps zurueckgibt (egal welcher) und ein Argument vom Typ `String` akzeptiert.
- Der Konstruktor der Klasse `Measurement` erfüllt diese Eigenschaft. ✓

Vor- und Nachteile von Funktionaler Programmierung

- Weniger fehleranfällig
- Einfacher zu unterhalten
- Ermöglicht elegante Programmierung
- Nicht abhängig von spezifischer Architektur

- Neue Sprachkonzepte zu lernen
- Ausführungsdetails unbekannt
- Aufgesetzt auf imperativer Sprache