



Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Java 8 – Lambdas und Streams

E-Voting Group, 24. Juni 2014

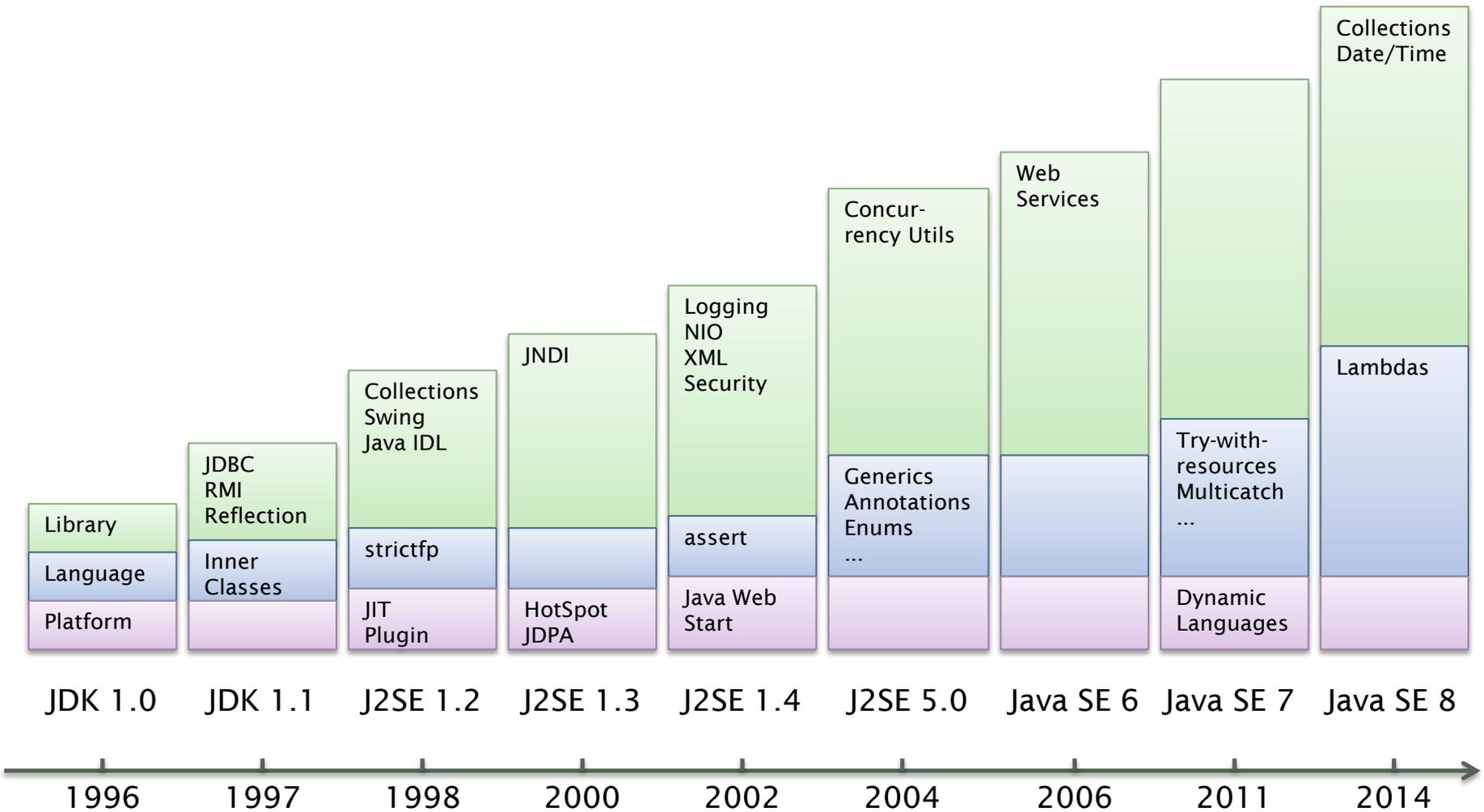
Stephan Fischli
Dozent BFH, Software-Architekt ISC-EJPD

Inhalt

- ▶ Einführung
- ▶ Lambda-Ausdrücke
- ▶ Collections, Streams und Bulk-Operationen
- ▶ Fazit

Einführung

Geschichte von Java



Wichtigste Neuerungen

Sprache

- ▶ Annotationen auf Java-Typen ([JSR 308](#))
- ▶ Lambda-Ausdrücke ([JSR 335](#))

Bibliotheken

- ▶ Date & Time API ([JSR 310](#))
- ▶ Bulk-Operationen für Collections ([JSR 335](#))

Plattform

- ▶ Modulsystem ([JSR 277](#)) auf Java 9 verschoben
- ▶ Profile als Ersatz

Vollständige Liste <http://openjdk.java.net/projects/jdk8/features>

Quellen

- ▶ Java 8 SE Spezifikation (JSR 337)
<https://jcp.org/en/jsr/detail?id=337>
- ▶ JDK 8 Download
<http://www.oracle.com/technetwork/java/javase/downloads/>
- ▶ Java 8 API Dokumentation
<http://docs.oracle.com/javase/8/docs/api/>
- ▶ Java Tutorial
<http://docs.oracle.com/javase/tutorial/>

Lambda-Ausdrücke

Warum ein neues Sprachkonstrukt?

- ▶ Java ist objektorientiert, also müssen Funktionen als Methoden von Klassen codiert werden:

```
File[] files = directory.listFiles(new FileFilter() {  
    public boolean accept(File file) {  
        return file.length() < 1024;  
    }  
});
```

- ▶ Mit Lambda-Ausdrücken können Funktionen auch ohne Erzeugung einer Klasse oder eines Objekts übergeben werden:

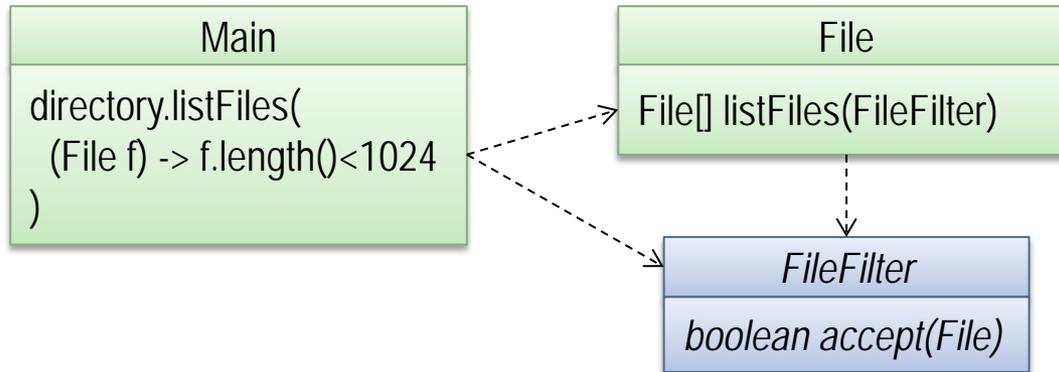
```
File[] files = directory.listFiles(  
    (File file) -> file.length() < 1024  
);
```

Lambda-Ausdrücke

- ▶ Lambda-Ausdrücke sind Ausdrücke der Form
argument list -> body
- ▶ Der Body kann sein:
 - ▶ ein einfacher Ausdruck, der ausgewertet wird
 - ▶ ein Block von Statements, die ausgeführt werden; kann einen Rückgabewert haben
- ▶ Beispiele
 - (int x, int y, int z) -> x + y + z
 - (Account a1, Account a2) -> a1.balance() - a2.balance()
 - (String s) -> { System.out.println(s); }
 - () -> 42

Funktionale Interfaces

- ▶ Lambda-Ausdrücke sind vom Typ eines funktionalen Interface, das ist ein Interface mit genau einer Methode



- ▶ Beispiele:

```
interface FileFilter { boolean accept(File file); }
interface Comparator<T> { int compare(T o1, T o2); }
interface Runnable { void run(); }
interface ActionListener { void actionPerformed(ActionEvent e); }
```

Verwendung funktionaler Interfaces

- ▶ Funktionale Interfaces dienen oft als Parametertypen bei der Implementierung von Verhaltensmustern wie dem Observer-, Strategy- oder Visitor-Pattern
- ▶ Beispiel

```
public File[] listFiles(FileFilter filter) {  
    ArrayList<File> files = new ArrayList<>();  
    for (File file : listFiles()) {  
        if (filter.accept(file)) files.add(f);  
    }  
    return files.toArray(new File[files.size()]);  
}
```

Verwendung von Lambda-Ausdrücken

- ▶ Lambda-Ausdrücke können überall verwendet werden, wo ein Objekt eines funktionalen Interfaces benötigt wird, insbesondere in
 - ▶ Variablendeklarationen
 - ▶ Zuweisungen
 - ▶ Methodenaufrufen
 - ▶ Return-Statements
- ▶ Beispiele

```
FileFilter filter = (File file) -> file.length() < 1024;  
Collections.sort(accounts,  
    (Account a1, Account a2) -> a1.balance() - a2.balance());  
return () -> { System.out.println("Hello world"); };
```

Zieltyp eines Lambda-Ausdrucks

- ▶ Der Compiler leitet den Typ eines Lambda-Ausdrucks aus dem Verwendungskontext ab (Zieltyp)
- ▶ Ein Lambda-Ausdruck ist kompatibel zu einem funktionalen Interface, wenn die Parameter, Rückgabewerte und Exceptions zu der Methode des Interface passen
- ▶ Da die Parametertypen aus dem Zieltyp bekannt sind, können sie im Lambda-Ausdruck meistens weggelassen werden

```
FileFilter filter = file -> file.length() < 1024;  
Collections.sort(accounts, (a1, a2) -> a1.balance() - a2.balance());
```

Scoping und Variablenbindung

- ▶ Lambda-Ausdrücke definieren keinen eigenen Scope, sondern gehören zum Scope des umgebenden Kontexts
- ▶ Die Referenzen *this* und *super* sowie Variablennamen werden somit im Kontext interpretiert
- ▶ Lambda-Ausdrücke dürfen lesend auf lokale Variablen des Kontexts zugreifen, sofern diese *effektiv final* sind

- ▶ **Beispiel**

```
int count = 100, sum = 0;  
Runnable r = () -> { for (int n = 1; n <= count; n++) sum += n; }; // Fehler  
new Thread(r).start();
```

Methodenreferenzen

- ▶ Methodenreferenzen sind wie Lambda-Ausdrücke, referenzieren aber existierende Klassen-, Objektmethoden oder Konstruktoren

- ▶ Beispiel:

```
class Account {  
    public static int compare(Account a1, Account a2) {  
        return a1.balance() - a2.balance();  
    }  
    public int compareTo(Account a) { return this.balance() - a.balance(); }  
}  
Collections.sort(accounts, Account::compare);  
Collections.sort(accounts, Account::compareTo);
```

Collections, Streams und Bulk-Operationen

Externe vs interne Iteration

- ▶ Das bisherige Collection-Framework basiert auf externer Iteration, d.h. Iterationen werden vom Client kontrolliert:

```
for (Account a: accounts) { if (a.balance() < 0) a.alert(); }
```

- ▶ Bei einer internen Iteration delegiert der Client die Iteration an die Bibliothek:

```
accounts.forEach(a -> { if (a.balance() < 0) a.alert(); });
```

- ▶ Vorteile:
 - ▶ Iteration und Logik sind getrennt
 - ▶ Optimierte Ausführung möglich

Streams

- ▶ Ein Stream
 - ▶ repräsentiert eine Folge von Elementen
 - ▶ hat Operationen zur Manipulation aller Elemente, diese beruhen auf interner Iteration
 - ▶ kann eine unbeschränkte Grösse haben
 - ▶ wird konsumiert, d.h. jede Verarbeitung benötigt einen neuen Stream

- ▶ Ein Stream ist ein Objekt vom generischen Typ *Stream<T>* oder der spezialisierten Typen *IntStream*, *LongStream*, *DoubleStream*

Erzeugen von Streams

Streams können erzeugt werden aus

- ▶ **Collections und Arrays**

 - `accounts.stream()`

 - `Arrays.stream(text.split("\\s+"))`

- ▶ **I/O-Kanälen**

 - `new BufferedReader(in).lines()`

 - `Files.lines(file), Files.list(dir), Files.walk(dir), Files.find(dir, depth, matcher)`

- ▶ **Generatorfunktionen**

 - `Stream.generate(Math::random)`

 - `Stream.iterate(1, x -> x+1)`

 - `Stream.empty()`

Stream-Operationen

- ▶ Stream-Operationen
 - ▶ haben funktionale Interfaces als formale Parameter, die das Verhalten definieren
 - ▶ sind funktional, d.h. sie verändern die Quelle des Streams nicht
 - ▶ können zustandslos oder zustandsbehaftet sein
- ▶ Stream-Operationen werden in Zwischen- und Terminaloperationen unterschieden

Zwischenoperationen

- ▶ **Zwischenoperationen transformieren Streams**

 - Stream<T> distinct()

 - Stream<T> limit(long maxSize)

 - Stream<T> skip(long n)

 - Stream<T> filter(Predicate<T> predicate)

 - Stream<R> map(Function<T, R> mapper)

 - Stream<T> sorted(Comparator<T> comparator)

- ▶ **Zugehörige Interfaces**

 - interface Predicate<T> { boolean test(T o); }

 - interface Function<T,R> { R apply(T o); }

 - interface Comparator<T> { int compare(T o1, T o2); }

Terminaloperationen

- ▶ Terminaloperationen erzeugen ein Resultat oder einen Nebeneffekt

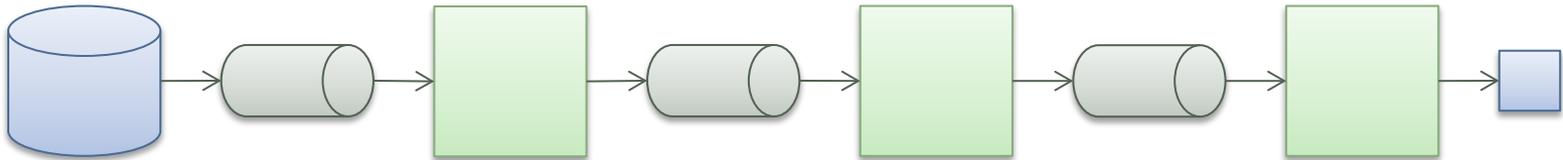
```
long          count()
Optional<T>   findAny/findFirst()
boolean       allMatch/anyMatch/noneMatch(Predicate<T> predicate)
Optional<T>   min/max(Comparator<T> comparator)
Optional<T>   reduce(BinaryOperator<T> accumulator)
void          forEach(Consumer<T> action)
```

- ▶ Zugehörige Interfaces

```
interface Predicate<T>      { boolean test(T o); }
interface Comparator<T>    { int compare(T o1, T o2); }
interface BinaryOperator<T> { T operate(T o1, T o2); }
interface Consumer<T>      { void accept(T o); }
```

Pipelines

- ▶ Stream-Operationen werden in Pipelines ausgeführt
- ▶ Eine Pipeline besteht aus
 - ▶ einer Quelle, welche die Elemente liefert
 - ▶ Zwischenoperationen, welche den Stream transformieren
 - ▶ einer Terminaloperation, die ein Resultat produziert

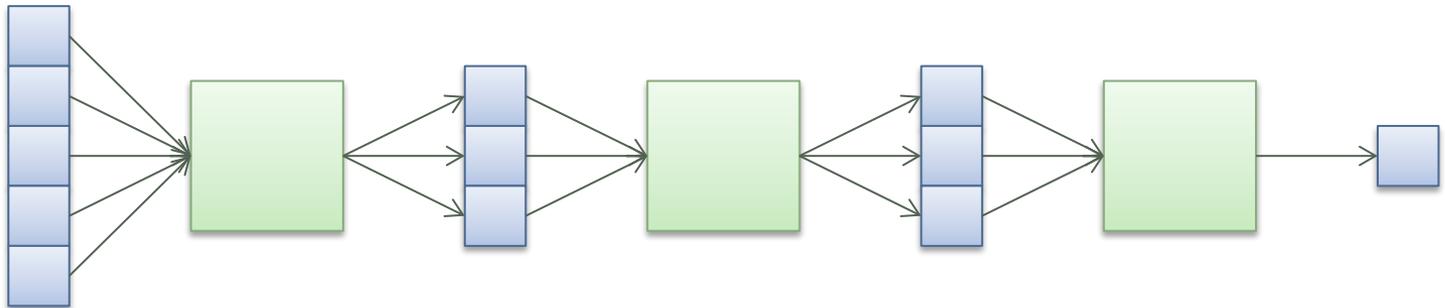


- ▶ **Beispiel**

```
accounts.stream().filter(a -> a.customer().age() > 65)  
    .mapToInt(a -> a.balance())  
    .average();
```

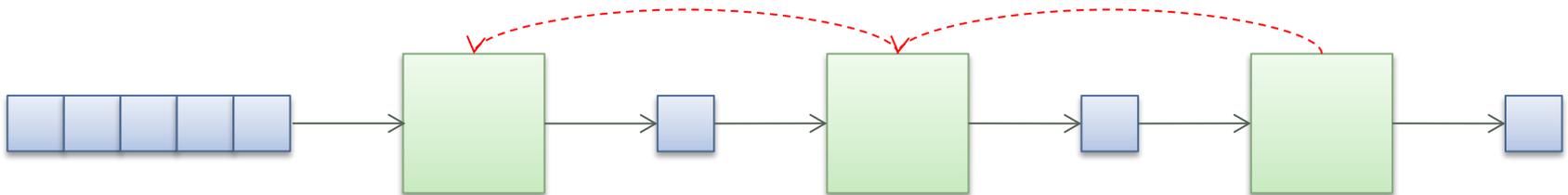
Laziness

- ▶ Stream-Operationen werden wenn möglich lazy ausgeführt
- ▶ Vorteile:
 - ▶ Elemente können in einem Durchgang verarbeitet werden, es braucht keine mehrfachen Iterationen und keine Zwischenspeicher
 - ▶ Beim Suchen von Elementen kann die Verarbeitung oft vorzeitig abgebrochen werden



Laziness

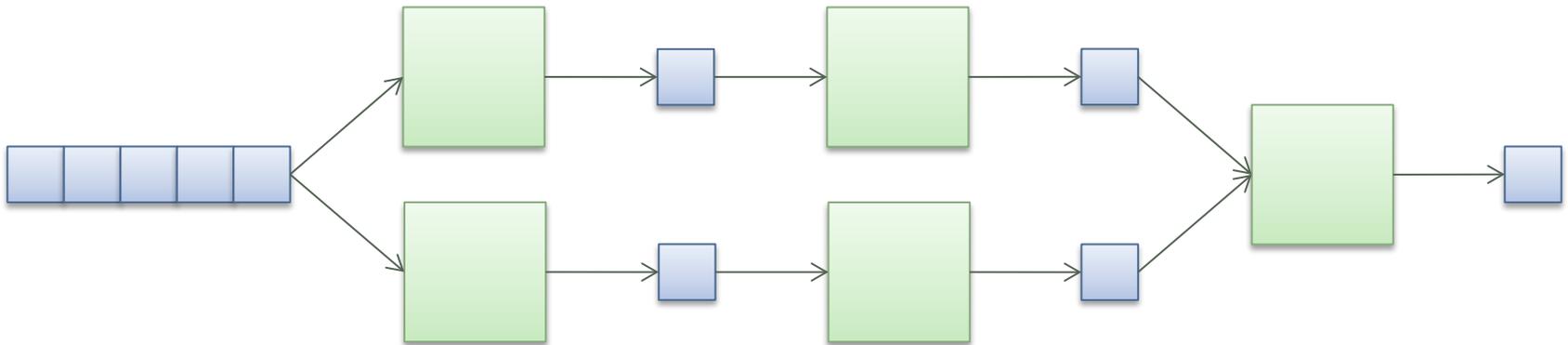
- ▶ Stream-Operationen werden wenn möglich lazy ausgeführt
- ▶ Vorteile:
 - ▶ Elemente können in einem Durchgang verarbeitet werden, es braucht keine mehrfachen Iterationen und keine Zwischenspeicher
 - ▶ Beim Suchen von Elementen kann die Verarbeitung oft vorzeitig abgebrochen werden



Parallelisierung

- ▶ Stream-Pipelines können sequentiell oder parallel ausgeführt werden
- ▶ Für eine parallele Ausführung dürfen sich die Operationen nicht gegenseitig beeinflussen (non-interference)
- ▶ Beispiel

```
accounts.parallelStream().filter(a -> a.customer().age() > 65)  
    .mapToInt(a -> a.balance()).average();
```



Funktionale Programmierung

Programmierparadigmen

- ▶ Imperative Programmierung
 - ▶ Strukturiert: Verwendung bestimmter Kontrollstrukturen
 - ▶ Prozedural: Aufteilung eines Programms in Prozeduren
 - ▶ Modular: Logische Einheiten mit Daten und Prozeduren
 - ▶ Objektorientiert: Klassen als instanziiierbare Module
- ▶ Deklarative Programmierung
 - ▶ Funktional: Aufgabe als funktionaler Ausdruck, Interpreter macht Funktionsauswertung
 - ▶ Logisch: Aufgabe als logischer Ausdruck, Interpreter versucht Aussage herzuleiten

Imperative vs funktionale Programmierung

- ▶ Imperative Programme
 - ▶ bestehen aus Anweisungen und arbeiten mit Speicherplätzen
 - ▶ beschreiben, *wie* etwas gemacht wird
- ▶ Funktionale Programme
 - ▶ bestehen aus Funktionen und arbeiten mit Variablen
 - ▶ beschreiben, *was* gemacht wird
- ▶ Vorteile
 - ▶ Funktionen haben keine Nebeneffekte
 - ▶ Funktionen werden immer lazy ausgewertet
 - ▶ Funktionen höherer Ordnung sind möglich

Fazit

Fazit

Lambda-Ausdrücke

- ▶ bieten eine einfache Syntax für anonyme Methoden und fördern dadurch einen neuen Programmierstil (funktionale Programmierung)
- ▶ ermöglichen die Implementierung effizienter Bibliotheken und damit die bessere Nutzung moderner Rechnerarchitekturen

Referenzen

- ▶ **Brian Goetz, State of the Lambda**
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html>
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>
- ▶ **Angelika Langer, Lambda Expressions and Streams in Java**
<http://www.angelikalanger.com/Lambdas/Lambdas.html>
- ▶ **The Java Tutorial, Lambda Expressions**
<http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
- ▶ **Mark Reinholds, Closures for Java**
<https://blogs.oracle.com/mr/entry/closures>

Carl Friedrich Gauss



$$1+2+3+4+5+\dots+99+100 = ?$$

$$(1+100)+(2+99)+(3+98)+\dots+(50+51) = 5050$$

```
IntStream.iterate(1, x -> x+1).limit(100).sum()
```

Danke für Ihre Aufmerksamkeit.