Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Java 8

Guild42, 18. November 2013

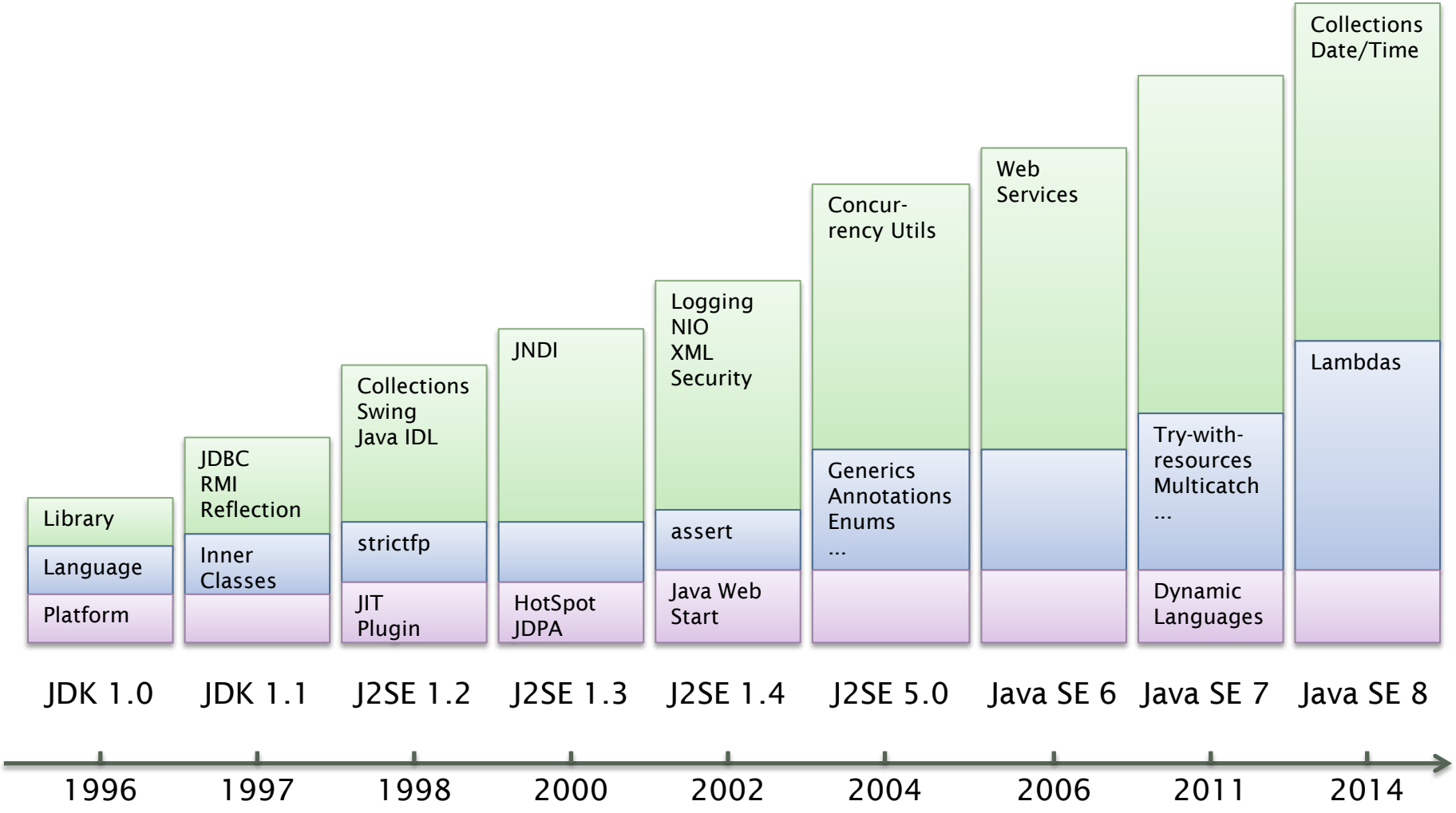
Stephan Fischli
Dozent BFH, Software-Architekt ISC-EJPD

Inhalt

- Einführung
- Lambda-Ausdrücke
- Collections und Bulk-Operationen

Einführung

Geschichte von Java



Aktueller Stand

Fahrplan

- Juni 2013 Feature Complete
- Sept 2013 Developer Preview
- Januar 2014 Final Candidate Release
- März 2014 General Availability

Quellen

- Java 8 SE Spezifikation (JSR 337)
- OpenJDK JDK 8 Projekt (http://openjdk.java.net/projects/jdk8/)
- Sourcecode und Binaries (https://jdk8.java.net/)

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

Date & Time API

- In Java war die Datums- und Zeitverarbeitung bisher umständlich
- Das neue Date & Time API
  - ist einfach und klar
  - unterstützt eine fluent Programmierung
  - beruht auf ISO-Standards
- Es gibt Klassen für
  - kontinuierliche Zeit, z.B. *Instant*, *Duration*
  - Alltagszeit, z.B. *ZonedDateTime*, *LocalDate*, *Period* usw.
- Beispiel

```
LocalDate date = LocalDate.of(2012, Month.NOVEMBER, 18).plusYears(1);
System.out.println(date.getDayOfWeek());
```

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 f) {
        return f.getName().endsWith(".java");
    }
});
```

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

```
File[] files = directory.listFiles(
    (File f) -> f.getName().endsWith(".java")
);
```

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
- Ein funktionales Interface ist ein Interface mit genau einer Methode (optionale Annotation @FunctionalInterface)
- Beispiele:

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

Funktionale Interfaces als Parameter

- Funktionale Interfaces dienen als Typen formaler Parameter, z.B.
  - in Callback-Situation
  - bei der Implementierung des Visitor-Patterns

- Beispiel

```
public File[] listFiles(FileFilter filter) {
    ArrayList<File> files = new ArrayList<>();
    for (File f : listFiles()) { if (filter.accept(f)) 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 f) -> f.getName().endsWith(".java");
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 zur Methode des Interface passen
- Da die Parametertypen aus dem Zieltyp bekannt sind, können sie im Lambda-Ausdruck meistens weggelassen werden

```
FileFilter filter = f -> f.getName().endsWith(".java");
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();
    }
}
Collections.sort(accounts, Account::compare);
accounts.forEach(System.out::print);
```

Funktionale Programmierung

- Prozedurale Programmierung:
  Funktionen operieren auf Argumenten und erzeugen Resultate
  y = f(x)

- Objektorientierte Programmierung :
  Methoden werden auf Objekten aufgerufen
  y = x.f()

- Funktionale Programmierung:
  Funktionen als Argumente und Resultate von Operationen
  y = F(f,x) bzw. y = x.F(f)

- Beispiel:

```
FileFilter filter = f -> f.getName().startsWith(".java");
File[] files = directory.listFiles(filter);
```

Collections und Bulk-Operationen

Externe vs interne Iteration

- Das bisherige Collection-Framework basiert auf externer Iteration, d.h. die Iteration wird vom Client kontrolliert:

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

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

```
accounts.forEach(a -> { if (a.balance() < a.threshold()) 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, die auf interner Iteration beruhen
  - 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 -> 2*x)
  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)
<R> Stream<R>      map(Function<T, R> mapper)
Stream<T>          sorted(Comparator<T> comparator)
```

- Zugehörige Interfaces

```
interface Predicate<E>     { boolean test(E e); }
interface Function<T,R>    { R apply(T t); }
interface Comparator<T>    { int compare(T o1, T o2); }
```

Terminaloperationen

- Terminaloperationen erzeugen ein Resultat oder einen Nebeneffekt

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

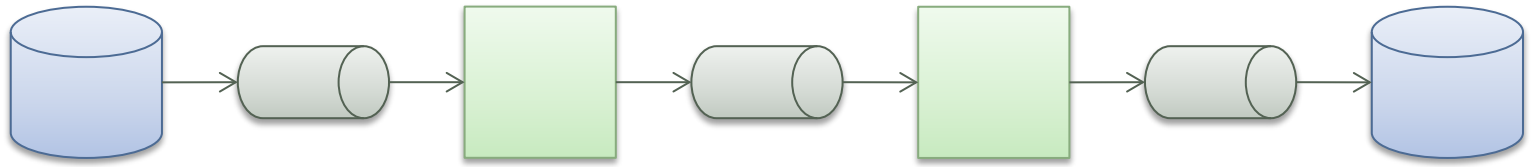
- Zugehörige Interfaces

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

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

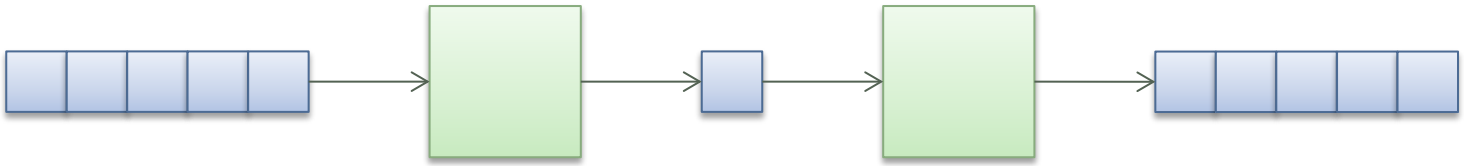


- Beispiel

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

Laziness

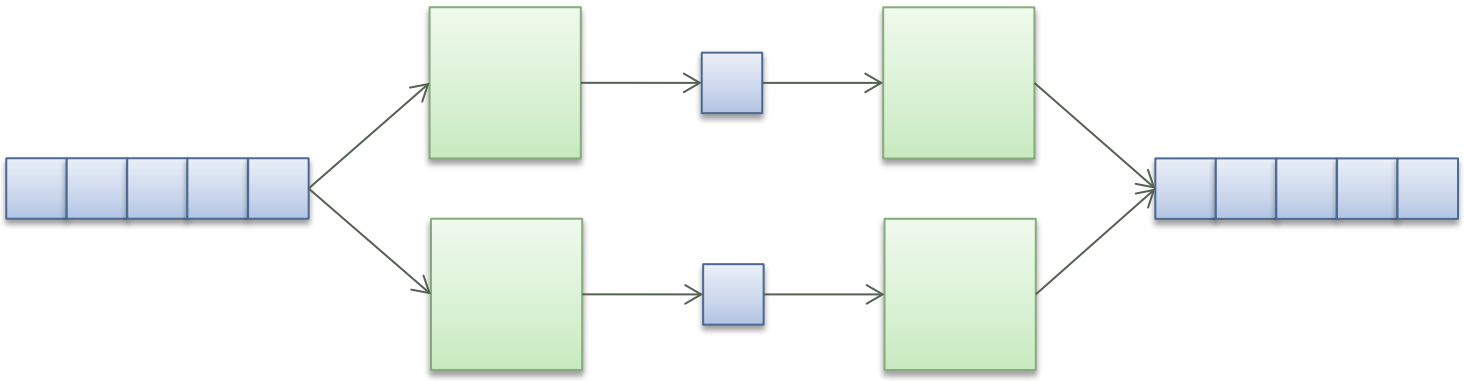
- Stream-Operationen werden wenn möglich lazy ausgeführt
- Vorteile:
  - Elemente können in einem Durchgang verarbeitet werden, es braucht keine mehrfachen Iterationen und keine Puffer
  - Beim Suchen von Elementen kann die Verarbeitung oft vorzeitig abgebrochen werden



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();
```



Default-Methoden

- Default-Methoden sind Methoden eines Interface mit einer Default-Implementierung
- Bestehende Klassen erben die Implementierung von Default-Methoden und bleiben somit kompatibel zum Interface
- Wenn eine Klasse mehrere Interfaces mit Default-Methoden implementiert, kann es zu Namenskonflikten kommen
- Beispiel

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<T> action) {
        for (T t : this) { action.accept(t); }
    }
```

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
- 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+...+99+100 = ?

(1+100)+(2+99)+(3+98)+...+(50+51) = 5050

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

Danke für Ihre Aufmerksamkeit.