Java 8

Die wichtigsten Neuerungen

W3L AG
info@W3L.de

2013

Agenda

- **Java Versionshistorie und Zeitplan**
- **Neue Features in Java 8 – Überblick**
- **Neue Features im Detail**
  - Lambda Expressions
    - Functional interfaces
    - Default/Defender-Methoden
    - Methoden-Referenzen
    - Bulk Data Operations
  - Annotations API Update
- **Fazit**

Java Versionshistorie und Zeitplan

- **Java Spracherweiterungen**
  - JDK 1.0 (Januar 1996)
  - JDK 1.1 (Februar 1997)
    - Inner classes, JavaBeans, Reflection
  - **J2SE 5.0 (September 2004)**
    - **Generics, Annotations, Autoboxing/unboxing, Enumerations, Varargs, for-each-Schleife, Static imports**
  - Java SE 6 (Dezember 2006)
    - Nichts!
  - Java SE 7 (Juli 2011)
    - Strings in switch, **try-with-resources statement**, <>-Operator (Generics), ...
    - Kleinere Spracherweiterungen
  - Java SE 8 (März 2014)
    - Lambda Expressions (JSR 335), Annotations on Java Types (JSR 308)

Java Versionshistorie und Zeitplan

- **.Net/C# Versionshistorie**
  - C# 1.0 (Januar 2002)
  - C# 2.0 (November 2005)
    - Generics, Partial Classes, Anonymous Methods, Nullable types, …
  - **C# 3.0 (November 2007)**
    - **Extensions Methods, Lambda Expressions, LINQ, …**
  - C# 4.0 (April 2010)
    - Named und Optionale Parameter, …
  - C# 5.0 (August 2012)
    - Asynchronous methods, …
- **Fazit**
  - Viele gute und erprobte Features wurden erst sehr spät nach Java übernommen
  - Java wurde ein wenig „abgehängt“

Neue Features in Java 8 - Überblick

- **Neuerungen in Java 8 (JSR 337)**
  - Lambda Expressions (JSR 335)
  - Neues Date und Time API (JSR 310)
    - java.time
    - OffsetDate, OffsetTime, OffsetDateTime
  - Annotations API Update (JSR 308)

- **JavaScript-Engine Rhino wird durch Nashorn ersetzt**
  - Wesentlich schnellere Ausführung von JavaScript-Code
- **Java FX jetzt Bestandteil des JDK**
  - Versionssprung von 2.2 auf JavaFX 8
- **Security Erweiterungen**
  - Implementierung besserer Verschlüsselungsalgorithmen
- **Unterstützung von Unicode 6.2**

Neue Features im Detail – Lambda Expressions

- **Hauptgründe für die Einführung von Lambda Expressions**
  - Besser lesbarerer, kompakterer Quellcode
  - Einfacher zu parallelisierende Algorithmen
- **Lambda Expressions**
  - Kompakte Schreibweise zum Formulieren einer anonymen Klasse mit einer Methode
- **Single Abstract Method Type (SAM type)**
  - Interface mit einer Methode
    - In Java bis Version 8 einzige Möglichkeit zur Realisierung eines Funktionstyps
  - Beispiele
    - Runnable, Comparator, ActionListener, FilenameFilter, ...

Neue Features im Detail – Lambda Expressions

- **Java alt (Java 7 und früher)**

```
File myDir = new File("c:\\");

FilenameFilter filter = new FilenameFilter(){
    public boolean accept(File f, String name){
        return name.toLowerCase().startsWith("w");
    }
};

File[] files = myDir.listFiles(filter);

for (File file : files){
    System.out.println(file.getName());
}
```

Neue Features im Detail – Lambda Expressions

- **Java 8 mit Lambda Expression**

```
File myDir = new File("c:\\");

File[] files = myDir.listFiles((f,s)-> s.toLowerCase().startsWith("w"));

for (File file : files){
    System.out.println(file.getName());
}
```

Neue Features im Detail – Lambda Expressions

- **Was fällt bei Betrachtung von myDir.listFiles((f,s)-> s.toLowerCase().startsWith("w")) auf?**
- **listFiles ist überladen**
  - Nur eine Operation verwendet SAM type mit Methode, die zwei Parameter akzeptiert
- **Lambda Expression implementiert SAM type interface bzw. die accept-Methode**
- **Bei Deklaration von Lambda Expression (f,s)->… werden keine Typen verwendet**
  - Compiler bestimmt Parameter-Typen aus dem Kontext
  - Alternativ
    - myDir.listFiles((File f,String s)-> s.toLowerCase().startsWith("w"))

Neue Features im Detail – Lambda Expressions

- **Bei Lambda Expressions andere Sichtbarkeiten als bei anonymen Klassen**
  - Lambda Expression hat direkten Zugriff auf alle im umgebenden Block sichtbaren Variablen
  - Lamda Expression selbst definiert **keinen eigenen Gültigkeitsbereich**

```
int i = 10;

//kompiliert nicht wegen Namenskonflikt!
Action a = ()-> { int i = 5; System.out.println(i); };
…

//kompiliert
Action a = new Action(){
   int i = 5;
   public void action(){ System.out.println(i); }
};
```

Neue Features im Detail – Lambda Expressions

- **Variable Capture**
  - Einschränkung für Zugriff auf lokale Variablen
    - Müssen final deklariert sein oder sich so verhalten (effectively final)
  - Effectively Final
    - Variable wird nach ihrer ersten Zuweisung nicht mehr verändert

```
int i=10;

//kompiliert nicht!
Action a = ()-> { System.out.println(i); };
…
i = 9;
```

Neue Features im Detail – Default/Defender-Methoden

- **Viele Schnittstellen können um eine sinnvolle Anwendung von Lambda Expressions erweitert werden**

```
public interface Iterable<T> {
    ...
    //Ueber jedes Element iterieren und action anwenden
    void forEach(Consumer<? super T> action);
}

public interface Collection<E> extends Iterable<E> {
    ...
}
```

- **Problem**
  - Änderungen an den Schnittstellen führen zu Inkompatibilität mit allen zuvor realisierten Implementierungen
  - Alle existierenden Programme müssten bei Umstieg auf Java 8 angepasst werden

Neue Features im Detail – Default/Defender-Methoden

- **Lösung**
  - In Java 8 können Methoden in Schnittstellen mit dem Schlüsselwort **default** deklariert und implementiert werden

```
public interface Iterable<T> {

    ...

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

}
```

Neue Features im Detail – Default/Defender-Methoden

- **Problem**
  - Für Schnittstellen kann Mehrfachvererbung eingesetzt werden!

```
public interface A{
    default void op(){
        System.out.println("Hello A!");
    }
}

public interface B{
    default void op(){
        System.out.println("Hello B!");
    }
}

public class C implements A,B{ //Fehler bzw. kann nicht kompiliert werden!

}
```

Neue Features im Detail – Default/Defender-Methoden

- **Lösung**
  - Methode in Unterklasse implementieren und Mehrdeutigkeit auflösen

```
public class C implements A,B{
      public void op(){
         A.super.op();
      }
  }
```

- **Fazit**
  - Default-Methoden sollten äußerst sparsam eingesetzt werden!

Neue Features im Detail – Functional interfaces

- **Wie zuvor erwähnt existieren bereits eine Reihe von SAM-type-Schnittstellen**
  - Die Schnittstellen sind jedoch nicht originär für die Nutzung in Java 8 realisiert worden

- **In Java 8 werden neue Schnittstellen für Funktionstypen im Paket java.util.function eingeführt**
  - **Consumer<T>**: Ausführen einer Funktion auf Objekt vom Typ T
  - **Supplier<T>**: Keine Eingabe; Rückgabe eines Objekts vom Typ T
  - **Predicate<T>**: Prüfen einer Bedingung auf Objekt vom Typ T; Rückgabe eine boolean-Wertes
  - **Function<T,R>**: Aufruf einer Funktion auf einem Objekt vom Typ T; Rückgabe eines Objekts vom Typ R
  - ... (im Paket java.util.function nachschauen)

Neue Features im Detail – Functional interfaces

- **Java 8**
  - Einsatz der neuen Schnittstellen für Funktionstypen aus dem Paket java.util.functions
    - Collection.forEach(**Consumer**)
    - Collection.removeIf(**Predicate**)
    - List.replaceAll(**UnaryOperator**)
    - ...

Neue Features im Detail – Methoden Referenzen

- **Über Klasse::statischeMethode kann eine Referenz auf eine statische Methode übergeben werden**

```
Integer i[] = {4,2,7,1,3};
Arrays.sort(i, Integer::compare);
```

- **Für Methoden-Referenz wird intern eine SAM-type-Implementierung auf Basis der Schnittstelle Comparator erzeugt**

```
Comparator<Integer> c = Integer::compare;
```

- **Alternative Verwendung von Lambda Expression**

```
Arrays.sort(i, (p1, p2)-> Integer.compare(p1, p2));
```

Neue Features im Detail – Methoden Referenzen

- **Auch nicht statische Methoden lassen sich als Parameterwert übergeben**

```
public class MyComparator implements Comparator<Integer>{

        int i = 1;

        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println("Vergleich " + i++ + ": o1=" + o1 + "; o2=" + o2 );
            return o1.compareTo(o2);
        }
}

MyComparator comp = new MyComparator();
Arrays.sort(i, comp::compare);
```

Neue Features im Detail – Methoden Referenzen

- **Referenzen auf Objekt-Methoden eines beliebigen Objektes eines bestimmten Typs**

```
public class Person{
        private String name;

        public Person(String name){
            this.name = name;
        }

        public void printName(){
            System.out.println(name);
        }
    }
```

```
List<Person> pList = new ArrayList<>();
pList.add(new Person("Max"));
pList.add(new Person("Fritz"));
pList.add(new Person("Maria"));

pList.forEach(Person::printName);
```

Neue Features im Detail – Bulk Data Operations

- **JEP 107: Bulk Data Operations for Collections**

"Add functionality to the Java Collections Framework for bulk operations upon data. This is commonly referenced as "filter/map/reduce for Java." The bulk data operations include both serial (on the calling thread) and parallel (using many threads) versions of the operations. Operations upon data are generally expressed as lambda functions."

- **Einführung von Streams (java.util.stream)**
  - erlauben filter/map/reduce auf Collections in Java
  - sequentielle oder parallele Verarbeitung möglich
    - bessere Auslastung der zugrundeliegenden Hardware
  - Schnittstelle java.util.stream.Stream von zentraler Bedeutung für Bulk Data Operations in Java

Neue Features im Detail – Bulk Data Operations

- **Streams – Intermediate operations**
  - **filter**: alle Elemente im Stream bezüglich eines **Predicate**-Objektes aussortieren
  - **map**: Transformation der Elemente im Stream mit Hilfe einer **Function**
  - **flatMap**: transformiert alle Elemente im Stream in 0 oder mehr Elemente mit Hilfe eines weiteren Streams
    - Ausflachen einer Datenstruktur
  - **peek:** Eine Aktion auf jedem Element im Stream ausführen
  - **distinct:** Aussortieren von Duplikaten

Neue Features im Detail – Bulk Data Operations

Beispiel

```
List<Person> pList = new ArrayList<>();
pList.add(new Person("Max", 5));
pList.add(new Person("Fritz", 27));
pList.add(new Person("Maria", 41));

pList.stream().filter(p-> p.getAge() > 18) //alle Personen aelter als 18
      .map(p-> new Employee(p)) //Personen in Mitarbeiter umwandeln
      .forEach(Person::printName); //Mitarbeiter ueber 18 ausgeben
```

Neue Features im Detail – Bulk Data Operations

- **Streams – Terminating operations**
  - Beenden die Verarbeitung eines Streams
  - Reducers
    - Reduzieren die Elemente im Stream
    - Je nach Operation werden unterschiedliche Typen zurückgeliefert
    - reduce(...), count(), findAny(), findFirst()
  - Collectors
    - Stellen für das Ergebnis der Verarbeitung eine Collection zur Verfügung
  - forEach
    - Eine Aktion auf jedes Element im Stream anwenden

Neue Features im Detail – Bulk Data Operations

Beispiel Reducers

```
//Die Anzahl aller Personen ueber 18 ausgeben
System.out.println(pList.stream().filter(p-> p.getAge() > 18).count());
```

Beispiel Collectors

```
List<Employee> eList = pList.stream().filter(p-> p.getAge() > 18)
            .map(p-> new Employee(p))
            .collect(Collectors.toList());
```

Neue Features im Detail – Bulk Data Operations

Über Streams einfache Parallelisierung von Operationen möglich

```
//Liste mit 10 Mio. Personen erzeugen
List<Person> s = Stream.generate(() -> new Person("Max Mustermann", (int)(Math.random()*60)))
                                .limit(10_000_000).collect(Collectors.toList());

//Liste sortieren und filtern sowie Personen in Mitarbeiter umwandeln
long start = System.currentTimeMillis();
List<Employee> eList = s.stream()
        .parallel() //ab hier parallele Verarbeitung
        .sorted()
        .filter(p->p.getAge()>18 && p.getAge()<=60)
        .sequential() //ab hier sequenzielle Verarbeitung
        .map(person-> new Employee(person))
        .collect(Collectors.toList());

long stop = System.currentTimeMillis();

System.out.println("Verbrauchte Zeit in ms: " + (stop - start));
```

Durch Parallelisierung bessere Auslastung möglich

- Mit Parallelisierung ca. 7 s
- Wird die Zeile .parallel() auskommentiert ca. 9s

Neue Features im Detail - Annotations API Update

- **Annotations bisher nur für Klassen, Methoden, Felder und Variablen**

- **Ab Java 8**
  - Generische Typen, Arrays oder Throws-Ausdrücke

```
Map<@NonNull String, @NonEmpty List<String>> map;
```

  - Eine Annotation lässt sich mehrfach verwenden

```
@Author("Meier")
@Author("Schulze")
public class Person{
   ...
}
```

Neue Features im Detail - Annotations API Update

- **Neue Annotation @FunctionalInterface für SAM-Typen**
  - Compiler prüft, ob Schnittstelle nur eine Methoden-Deklaration enthält, wie für SAM-Typen gefordert

```
@FunctionalInterface
public interface Supplier<T> {

   T get();
}
```

Fazit

- **Die neuen Features in Java 8, vor allem die Lambda Expressions, werden die Programmierung nachhaltig beeinflussen**
  - Ähnlich den Änderungen die mit Java 5 eingeführt wurden
    - Generics, Annotations, ...
- **Leider kommen die Neuerungen sehr spät**

Diskussion!

Inhouse-Schulungen

Wir bieten Inhouse-Schulungen und Beratung durch unsere IT-Experten und –Berater.

Schulungsthemen

- Softwarearchitektur (OOD)
- Requirements Engineering (OOA)
- Nebenläufige & verteilte Programmierung

Gerne konzipieren wir auch eine individuelle Schulung zu Ihren Fragestellungen.

Sprechen Sie uns an!
Tel. 0231/61 804-0, info@W3L.de

W3L-Akademie

In Zusammenarbeit mit der Fachhochschule Dortmund bieten wir

zwei Online-Studiengänge

- B.Sc. Web- und Medieninformatik
- B.Sc. Wirtschaftsinformatik

und 7 Weiterbildungen im IT-Bereich an.

Besuchen Sie unsere Akademie!
http://Akademie.W3L.de