adesso business. people. technology.

Neuerungen in Java 8

Eric Giese

adesso AG

Neuerungen in Java 8

- Sprachänderungen
- Die Streams API
- Java Platform, 8th Edition
- Ein Blick unter die Haube
- Auswirkungen und Ausblick

Java 8 - Die Sprache

Klassische Java - Interfaces

Iterator<E>

- hasNext() : boolean
- next() : E
- remove() : void

List<E>

- size() : int
- isEmpty() : boolean
- contains(Object) : boolean
- iterator() : Iterator<E>
- toArray() : Object[]
- toArray(T[]) <T> : T[]
- add(E) : boolean
- remove(Object) : boolean
- containsAll(Collection<?>) : boolean
- addAll(Collection<? extends E>) : boolean
- addAll(int, Collection<? extends E>) : boolean
- removeAll(Collection<?>) : boolean
- retainAll(Collection<?>) : boolean
- clear() : void
- equals(Object) : boolean
- hashCode() : int
- get(int) : E
- set(int, E) : E
- add(int, E) : void
- remove(int) : E
- indexOf(Object) : int
- lastIndexOf(Object) : int
- listIterator() : ListIterator<E>
- listIterator(int) : ListIterator<E>
- subList(int, int) : List<E>

Path

- getFileSystem() : FileSystem
- isAbsolute() : boolean
- getRoot() : Path
- getFileName() : Path
- getParent() : Path
- getNameCount() : int
- getName(int) : Path
- subpath(int, int) : Path
- startsWith(Path) : boolean
- startsWith(String) : boolean
- endsWith(Path) : boolean
- endsWith(String) : boolean
- normalize() : Path
- resolve(Path) : Path
- resolve(String) : Path
- resolveSibling(Path) : Path
- resolveSibling(String) : Path
- relativize(Path) : Path
- toUri() : URI
- toAbsolutePath() : Path
- toRealPath(LinkOption...) : Path
- toFile() : File
- register(WatchService, Kind<?>[], Modif

SAM (Single Abstract Method) Interfaces

Runnable
- run() : void

Comparator<T>
- compare(T, T) : int

EventHandler<T extends Event>
- handle(T) : void

Function<F, T>
- apply(F) : T

Predicate<T>
- apply(T) : boolean

SAM‘s als anonyme Klassen

Bei <u>späterem</u> Klick auf Button Dialogbox anzeigen

```
EventHandler<E>{ void handle(E); }

Button         { void setOnAction(EventHandler<ActionEvent>); }
```

```
final Window win = ...
button.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent e) {
    showMessage(win, "Clicked: " + e.getTarget());
  }
});
```

SAM‘s durch Lambda Expressions

Bei <u>späterem</u> Klick auf Button Dialogbox anzeigen

```
EventHandler<E>{ void handle(E); }

Button         { void setOnAction(EventHandler<ActionEvent>); }
```

```
final Window win = ...
button.setOnAction((ActionEvent e) -> {

    showMessage(win, "Clicked: " + e.getTarget());

});
```

SAM‘s durch Lambda Expressions (Inline)

Bei <u>späterem</u> Klick auf Button Dialogbox anzeigen

```
EventHandler<E>{ void handle(E); }

Button         { void setOnAction(EventHandler<ActionEvent>); }
```

```
Window win = ...
button.setOnAction(
        e -> showMessage(win, "Clicked: " + e.getTarget());
```

DONE!

Wofür kann man Lambdas einsetzen?

- Asynchrone Verarbeitung

```
Runnable answer = () -> out.println("The Answer is 42");
new Thread(answer).start();
```

- Lazyness

```
logger.fine("Expensive even if not logged:" + veryBigMap);
logger.fine(() -> "Cheap if not logged:" + veryBigMap);
```

- Funktionale Transformation

```
File dir = ...
File[] files = dir.listFiles(f -> f.isFile());
```

Schick: MethodReferences!

- Lambda-Expression

```
dir.listFiles(f -> f.isFile());
```

- **Method Reference**

```
dir.listFiles(File::isFile);
```

- Kandidaten:

`String::isEmpty, Integer::toString, Objects::isNull, ...`

Spaß mit Lambdas und Collections

```
List<Customer> customers = ...
```

- Kunden ohne Einkäufe entfernen…

```
customers.removeIf(c -> c.getItems().isEmpty());
```

- Rest per Name sortieren …

```
customers.sort(Comparator.comparing(Customer::getName));
```

- und eine Rechnung senden!

```
customers.forEach(c -> c.sendInvoice());
```

An die Java-Gurus:

Woher kommen die neuen Interface-Methoden?

```
customers.sort(Comparator.comparing(Customer::getName));
```

Neue Methode in List?

Statische Methode in Comparator??

Interfaces: Default Methoden

```
public interface List<E> {
  E get(int index);

  default void set(int index, E element){
    throw new UnsupportedOperationException();
  }

  default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
  }
```

Interfaces: Statische Methoden

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  ...
  public static <T> Comparator<T> naturalOrder() {
    return NaturalOrderComparator.INSTANCE;
  }

  public static <T, U> Comparator<T> comparing(Function<T, U> fn){
    return (c1, c2) -> fn.apply(c1).compareTo(fn.apply(c2));
  }
```

Anmerkung: Die Generics wurden vereinfacht, der Code ist KEIN gültiges Java!

Putting it together

Kunden nach Namen sortieren. Bei Namensgleichheit werden sie absteigend nach Alter sortiert.

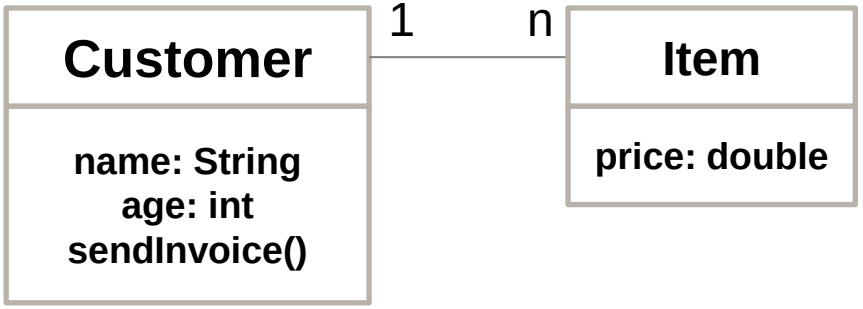
```
import static Comparator.comparing;

customerList.sort(
  comparing(Customer::getName).thenComparing(
    comparing(Customer::getAge).reversed()));
```

static methods

default methods

DIE STREAM API

Beispiel: Eine Kundendatenverarbeitung




```
Collection<Customer> customers = ...
```

Aufgabe: Für alle Kunden Rechnungen versenden!

Externe Iteration

```
for (Customer c : customers) { c.sendInvoice(); }
```

Interne Iteration

```
customers.forEach(Customer::sendInvoice); //c -> c.sendInvoice()
customers.iterator().forEachRemaining(Customer::sendInvoice);
customers.parallelStream().forEach(Customer::sendInvoice);

// roll your own: <E> void forEach(Consumer<E> c){ ... }
```

Die Stream-API: Queries auf iterierbaren Datenstrukturen

- Allen volljährigen Kunden eine Rechnung schicken

```
customers.stream()                          // Stream<Customer>
   .filter(c -> c.getAge() >= 18)           // in: Predicate<Customer>
   .forEach(Customer::sendInvoice);         // in: Consumer<Customer>
```

- Alle minderjährigen Kunden als Liste

```
List<Customer> underage = customers.stream()
   .filter(c -> c.getAge() < 18)
   .collect(Collectors.toList());
```

Die Stream-API: Queries auf iterierbaren Datenstrukturen

- Die Namen aller Kunden als Set

```
Set<String> names = customers.stream()
   .map(Customer::getName)                    // in: Function<Customer,String>
                                              // out: Stream<String>
   .collect(Collectors.toSet());
```

- Dasselbe als sortierte Liste mit paralleler Verarbeitung

```
List<String> names = customers.parallelStream()
   .map(Customer::getName)                     // Stream<String>
   .sorted().distinct()
   .collect(Collectors.toList());
```

Primitives Streaming

Umsatz berechnen

=> Den Wert aller Kundenpositionen aufaddieren!

```
double turnaround = customers.stream()  //Stream<Customer>
   .flatMap(c -> c.getItems().stream())  //Stream<Item>
   .mapToDouble(Item::getPrice)          //in:  ToDoubleFunction<Item>
                                         //out: DoubleStream
   .sum();
```

Geht mit `int, long, double`

Primitive Operationen: `average(), count(), ...`

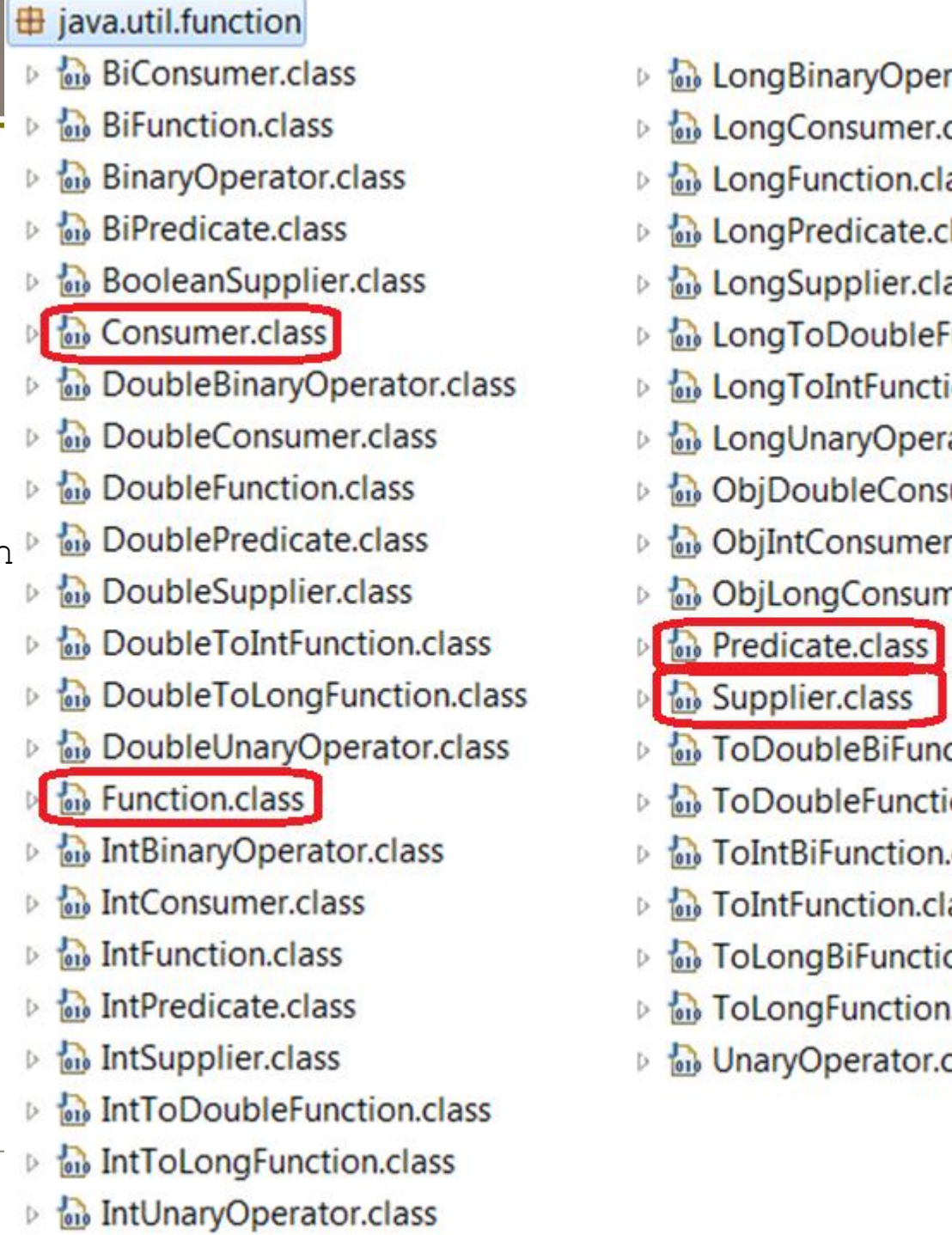
Woher kommen die Lambdas?

- **Generische Basisformen**

```
Consumer:  I -> void
Function:  I -> O
Predicate: I -> boolean
Supplier:  () -> O
```

- **Primitive Spezialisierungen**

```
int
long
double
```



Neuerungen auf der Java Platform

- *Nicht neu, aber ab 8 im JDK enthalten (mit eigenem launcher)*

- Neues Client-GUI-Framework
  - JavaFX vs. Swing + AWT
  - CSS-Styling, XML-Layouting
  - Java, Javascript

  - *… more cool features …*

Die neue Javascript Engine „Nashorn“

- Ersatz für das lahme „Rhino“
- *Rocked-Powered by invokeDynamic*

- Calling from java

```
ScriptEngineManager factory = ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("nashorn");
engine.eval("print('Hello, World!');");
```

- (JavaFx) GUI-Coding in JS

```
var button = new Button();
button.text = "Say 'Hello World'";
button.onAction = function() print("Hello World!");
```

Was gibt‘s sonst so Neues im JDK?

- **JEP 150: Date & Time API**
  - basiert auf joda-time *(selber Author)*
  - java.util.Calendar ist Vergangenheit

```
LocalDate date = LocalDate.of(2013, DECEMBER, 13);

assertEquals(DECEMBER, date.getMonth());


ZonedDateTime time = ZonedDateTime.from(date.withYear(2012));

long millis = Instant.from(time).toEpochMilli();


String out = DateTimeFormatter.ISO_DATE.format(time);
```

Was gibt‘s sonst so Neues im JDK?

- **JEP 155: Concurrency Updates**
  - `ForkJoinPool` improvements
  - `ConcurrentHashMap` mit map-reduce
- **JEP 135: Base64 Encoding & Decoding**
  - *wow!*
- *Modellpflege:* JDBC 4.2, Unicode 6.2, …

Memory-Tuning:

Das RT und die JVM im Schlankheitswahn!

Runtime Library

- JEP 147: Reduce Class Metadata Footprint
- JEP 149: Reduce Core-Library Memory Usage

Virtual Machine

- JEP 148: Small VM *(9 -> 3MB!)*
- JEP 122: Remove the Permanent Generation
- JEP 178: Statically Linked Libraries

Ongoing Roadwork:

Modularisierung mit Jigsaw

- JEP 162: Prepare for Modularization
- JEP 161: Compact Profiles

Profile	Size (MB)	libs
1	< 14	lang, util, (n)io, math, text, time, security-core
2	~ 18	+ rmi, xml, sql
3	~ 21	+ instrumentation, naming, script, xml-dsig
full	45+++	+ * (awt, swing, javafx, …)

Ein Blick unter unter die Haube

Lambdas sind keine inneren Klassen!

```
class Hello {
  public String toString() { return "Hello, world!"; }

  Runnable r = () -> System.out.println(this);
}
new Hello().r.run();
```

Ausgabe: "Hello, World!"

Bei innerer Klasse: "Hello$Runnable$1@42"

Lambdas in kompilierter Form

```
class Hello {
  public String toString() { return "Hello, world!"; }

  Runnable r = [invokeDynamic: use lambda$1 as Block];
  synthetic void lambda$1(){
    System.out.println(this);
  }
}
```

- Deklaration: invokeDynamic-Linking +„Rezept“
- Body: synthetische Methode *(entfällt bei MethodReference)*

Backporting Lambdas?

- Retrolambda
  1. Klassen mit javac 8 kompilieren
  2. Durch Retrolambda parsen: Ruft die J8-Lambda-Factory auf um bytecode für Java 5, 6, 7 zu erzeugen
- **Einschränkungen**
  - Keine default oder static Methoden auf interfaces
  - Keine neuen Bibliotheken (Streams, functions)

- Als funktionale Library z.B. Guava:

```
ImmutableSet<String> names = FluentIterable.from(customers)
        .filter(c -> c.getAge() > 18)
        .transform(Customer::getName)
        .toSortedSet(Ordering.natural());
```

Abbildung von Lambdas auf bisherigen Hotspot-VMs:

```
void forEach(Consumer<E> c){
        for (E e : this) c.apply(e);
}
```

- `forEach` ruft virtuelle Methode `apply` in Schleife auf

```
customers.forEach ( c -> println(c));
```

- `apply` ist **monomorphic** -> inlined

```
customers.forEach ( c -> c.sendInvoice());
```

- `apply` wird **bimorphic** -> inlined, conditional

```
customers.forEach ( c -> println(c.getItems()));
```

- `apply` wird **megamorphic** -> <u>not inlined</u>, vtable dispatch

Auswirkungen von Java 8

Einsatzmöglichkeiten von Lambdas

- … für Queries und Transaktionen (JDBC, JPA)
- … für asychrone Callbacks (AJAX, JMS?)
- … statt Strings für Bean Properties
- … in anderen JVM-Sprachen *(LambdaMetafactory , Stream-API)*
  - javascript!
  - groovy?
  - Scala???

Wie wird sich die Java-Platform entwickeln?

- Compile Once, Run Everywhere revisited
  - Geschwindigkeitsoptimierte Maxi-VMs für Server
  - Speicheroptimierte Mini-VM‘s für Embebbed / Mobile
- Standardumgebung für neue funktionale und dynamische Sprachen
- Zurück auf die *(embedded)* Clients!
  - Gui-Entwicklung mit javaFX und javascript?
  - Fat-Binaries mit eingebetteten VM‘s und statisch verlinkten nativen Libs

Checkliste: ‚Veraltete' Java-Sprachkonstrukte

Konstrukt	Ersatz
Arrays *(außer primitive)*	Generischer Container (seit 1.5)
Anonyme & Innere Klassen	Lambda Expressions
Abstrakte Klassen	Interfaces mit Default Methoden
For-Schleife (1.5)	`forEach()`
Checked Exceptions	Unchecked Exceptions
null	`Optional<T>`

So long…

Wenns später noch Fragen gibt:
giese@adesso.de

Further Reading

- Tech Empower: Everything about Java 8
- Maurice Naftalin's Lambda FAQ
- Brian Goetz: State Of The Lambda
- Brian Goetz: State Of The Lambda – Libaries Edition
- Brian Goetz: Lambda Translation (2012)
- Brian Goetz: Implementing Lambda Expressions in Java