Lambdas in Java-8

Floyd Kretschmar

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Inhaltsverzeichnis

Arbeitsgruppe zum Thema Lambda-Ausdrücken in Java-8

Project Lambda und JSR 335

Arbeitsgruppe zum Thema Lambda-Ausdrücken in Java-8

- Lambda-Ausdrücke

Project Lambda und JSR 335

Arbeitsgruppe zum Thema Lambda-Ausdrücken in Java-8

- Lambda-Ausdrücke
- Methoden- und Konstruktorreferenzen

Project Lambda und JSR 335

Arbeitsgruppe zum Thema Lambda-Ausdrücken in Java-8

- Lambda-Ausdrücke
- Methoden- und Konstruktorreferenzen
- erweiterte Zieltypisierung und Typreferenzierung

Project Lambda und JSR 335

Arbeitsgruppe zum Thema Lambda-Ausdrücken in Java-8

- Lambda-Ausdrücke
- Methoden- und Konstruktorreferenzen
- erweiterte Zieltypisierung und Typreferenzierung
- Standard- und statische Methoden in Interfaces

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale
Programmiersprachen
Funktionen

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale Programmiersprachen
Funktionen

Objektorientierte Programmiersprachen
???

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale Programmiersprachen
Funktionen

Objektorientierte Programmiersprachen
Objekte

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale Programmiersprachen
Funktionen

Objektorientierte Programmiersprachen
Objekte

- Gemeinsamkeiten nicht direkt offensichtlich, da Objekte oft "schwergewichtig" sind

Motivation

> Goetz (2013)
>
> "...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale Programmiersprachen	**Objektorientierte Programmiersprachen**
Funktionen	Objekte

- Gemeinsamkeiten nicht direkt offensichtlich, da Objekte oft "schwergewichtig" sind
- Häufig: Interfaces, die genau eine Methode definieren

Motivation

Goetz (2013)

"…**basic values** can dynamically encapsulate **program behavior**…"

Funktionale Programmiersprachen
Funktionen

Objektorientierte Programmiersprachen
Objekte

- Gemeinsamkeiten nicht direkt offensichtlich, da Objekte oft "schwergewichtig" sind
- Häufig: Interfaces, die genau eine Methode definieren
- Werden auch als "Callback" bezeichnet und oft anonym instanziiert

Motivation

Goetz (2013)

"...**basic values** can dynamically encapsulate **program behavior**..."

Funktionale Programmiersprachen	**Objektorientierte Programmiersprachen**
Funktionen	Callback

- Gemeinsamkeiten nicht direkt offensichtlich, da Objekte oft "schwergewichtig" sind
- Häufig: Interfaces, die genau eine Methode definieren
- Werden auch als "Callback" bezeichnet und oft anonym instanziiert

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax
- Handhabung von `this` und anderen Bezeichnern

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax
- Handhabung von `this` und anderen Bezeichnern
- Semantik des Klassenladens und der Objektinstanziierung

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax
- Handhabung von `this` und anderen Bezeichnern
- Semantik des Klassenladens und der Objektinstanziierung
- Verwendung von lokalen non-`final` Variablen nicht möglich

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax
- Handhabung von `this` und anderen Bezeichnern
- Semantik des Klassenladens und der Objektinstanziierung
- Verwendung von lokalen non-`final` Variablen nicht möglich
- Kontrollfluss ist nicht leicht zu abstrahieren

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Ein Interface . . .

- ist bereits ein elementarer Bestandteil des Typ-Systems

Anonyme innere Klassen Vorteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Ein Interface ...

- ist bereits ein elementarer Bestandteil des Typ-Systems
- hat eine definierte Repräsentation zur Laufzeit

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Ein Interface . . .

- ist bereits ein elementarer Bestandteil des Typ-Systems
- hat eine definierte Repräsentation zur Laufzeit
- kodifiziert eine informelle Vereinbarungen, ausgedrückt durch seine Javadoc-Kommentare

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

Ein Interface ...

- ist bereits ein elementarer Bestandteil des Typ-Systems
  → **Beispiel für nominale Typisierung**
- hat eine definierte Repräsentation zur Laufzeit
- kodifiziert eine informelle Vereinbarungen, ausgedrückt durch seine Javadoc-Kommentare

Strukturelle vs. nominale Typisierung

Vergleich von Typisierungsarten (Kyle, 2016)

```
class Foo {
    method(input: string): number { ... }
}
class Bar {
    method(input: string): number { ... }
}

let foo: Foo = new Bar();
```

- Ist diese Zuordnung gültig?

Strukturelle vs. nominale Typisierung

Vergleich von Typisierungsarten (Kyle, 2016)

```
class Foo {
    method(input: string): number { ... }
}
class Bar {
    method(input: string): number { ... }
}

let foo: Foo = new Bar();
```

- Ist diese Zuordnung gültig?
- **Nominale Typisierung:** Nein, denn `Foo` $\neq$ `Bar`

Strukturelle vs. nominale Typisierung

Vergleich von Typisierungsarten (Kyle, 2016)

```
class Foo {
    method(input: string): number { ... }
}
class Bar {
    method(input: string): number { ... }
}

let foo: Foo = new Bar();
```

- Ist diese Zuordnung gültig?
- **Nominale Typisierung:** Nein, denn `Foo` $\neq$ `Bar`
- **Strukturelle Typisierung:** Ja, denn `Foo` und `Bar` haben die selbe Struktur

Strukturelle vs. nominale Typisierung

Vergleich von Typisierungsarten (Kyle, 2016)

```
class Foo {
    method(input: string): number { ... }
}
class Bar {
    method(input: string): boolean { ... }
}

let foo: Foo = new Bar();
```

- Ist diese Zuordnung gültig?
- **Nominale Typisierung:** Nein, denn `Foo` $\neq$ `Bar`
- **Strukturelle Typisierung:** Nein, denn `Foo` und `Bar` haben nicht die selbe Struktur

Struktureller Funktionstyp als Alternative?

Funktion “String und Object nach Integer” (Goetz, 2013)

```
(String, object)->int
```

Struktureller Funktionstyp als Alternative?

Funktion “String und Object nach Integer” (Goetz, 2013)

```
(String, object)->int
```

- stärkere Vermischung von strukturellen und nominalen Datentypen in Java

Struktureller Funktionstyp als Alternative?

Funktion “String und Object nach Integer” (Goetz, 2013)

```
(String, object)->int
```

- stärkere Vermischung von strukturellen und nominalen Datentypen in Java
- Aufsplittung einheitlicher Bibliotheksstandards in zwei unvereinbare Formate

Struktureller Funktionstyp als Alternative?

Funktion “String und Object nach Integer” (Goetz, 2013)

```
(String, object)->int
```

- stärkere Vermischung von strukturellen und nominalen Datentypen in Java
- Aufsplittung einheitlicher Bibliotheksstandards in zwei unvereinbare Formate
- unhandliche Syntax (insbesondere im Bezug auf Exception-Behandlung)

Struktureller Funktionstyp als Alternative?

Funktion “String und Object nach Integer” (Goetz, 2013)

```
(String, object)->int
```

- stärkere Vermischung von strukturellen und nominalen Datentypen in Java
- Aufsplittung einheitlicher Bibliotheksstandards in zwei unvereinbare Formate
- unhandliche Syntax (insbesondere im Bezug auf Exception-Behandlung)
- keine verschiedenen Laufzeitrepräsentationen für jede einzelne Funktion $\rightarrow$ “type erasure”

Goetz (2013)

“So, we have instead followed the path of **"use what you know"** – since existing libraries use functional interfaces extensively, we codify and leverage this pattern.”

Goetz (2013)

“So, we have instead followed the path of "**use what you know**" – since existing libraries use functional interfaces extensively, we codify and leverage this pattern.”

- **functional interfaces**: Interfaces mit genau einer Methode

> **Goetz (2013)**
>
> “So, we have instead followed the path of **"use what you know"** – since existing libraries use functional interfaces extensively, we codify and leverage this pattern.”

- **functional interfaces**: Interfaces mit genau einer Methode
- `@FunctionalInterface`-Annotation kann verwendet werden um Designintention zu verdeutlichen

> **Goetz (2013)**
>
> “So, we have instead followed the path of "**use what you know**" – since existing libraries use functional interfaces extensively, we codify and leverage this pattern.”

- **functional interfaces**: Interfaces mit genau einer Methode
- `@FunctionalInterface`-Annotation kann verwendet werden um Designintention zu verdeutlichen
- Eine Vielzahl vordefinierter Interfaces dieser Art existieren in Java-8: `Consumer<T>`, `Function<T,R>`, `UnaryOperator<T>`, ...

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Unhandliche Syntax
- Semantik des Klassenladens und der Objektinstanziierung

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(
    e -> ui.dazzle(e.getModifiers())
);
```

- Unhandliche Syntax
- Semantik des Klassenladens und der Objektinstanziierung

Anonyme innere Klassen Nachteile

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(
    e -> ui.dazzle(e.getModifiers())
);
```

- Unhandliche Syntax
- Semantik des Klassenladens und der Objektinstanziierung

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(
    e -> ui.dazzle(e.getModifiers())
);
```

- Unhandliche Syntax
- Semantik des Klassenladens und der Objektinstanziierung

Allgemeine Form eines Lambda-Ausdrucks (Goetz, 2013)

```
(argument1, argument2, ..., argumentN) ->
{
    ...
    return ... ;
}
```

Allgemeine Form eines Lambda-Ausdrucks (Goetz, 2013)

```
(argument1, argument2, ..., argumentN) ->
{
    ...
    return ... ;
}
```

- **Parameterliste:** Klammern bei weniger als 2 Elementen optional

Allgemeine Form eines Lambda-Ausdrucks (Goetz, 2013)

```
(argument1, argument2, ..., argumentN) ->
{
    ...
    return ... ;
}
```

- **Parameterliste:** Klammern bei weniger als 2 Elementen optional
- **Körper:**
  - `break` und `continue` sind auf oberster Ebene verboten

Allgemeine Form eines Lambda-Ausdrucks (Goetz, 2013)

```
(argument1, argument2, ..., argumentN) ->
{
    ...
    return ... ;
}
```

- **Parameterliste:** Klammern bei weniger als 2 Elementen optional
- **Körper:**
  - `break` und `continue` sind auf oberster Ebene verboten
  - jeder Pfad musst etwas zurückgeben oder eine Exception werfen

Allgemeine Form eines Lambda-Ausdrucks (Goetz, 2013)

```
(argument1, argument2, ..., argumentN) ->
{
    ...
    return ... ;
}
```

- **Parameterliste:** Klammern bei weniger als 2 Elementen optional
- **Körper:**
  - `break` und `continue` sind auf oberster Ebene verboten
  - jeder Pfad musst etwas zurückgeben oder eine Exception werfen
  - bei einzeiligem Körper sind Klammern und `return` optional

Typ-Inferenz und Zieltypisierung

Typisierung eines Lambda-Ausdrucks (Goetz, 2013)

```
ActionListener l = e -> ui.dazzle(e.getModifiers())
```

Typ-Inferenz und Zieltypisierung

Typisierung eines Lambda-Ausdrucks (Goetz, 2013)

```
ActionListener l = e -> ui.dazzle(e.getModifiers())
```

- Name des Funktions-Interfaces wird nicht explizit angegeben

Typ-Inferenz und Zieltypisierung

Typisierung eines Lambda-Ausdrucks (Goetz, 2013)

```
ActionListener l = e -> ui.dazzle(e.getModifiers())
```

- Name des Funktions-Interfaces wird nicht explizit angegeben
- Compiler versucht den **Zieltyp** dynamisch anhand des **Programmkontextes** zu inferieren

Nicht jeder Labda-Ausdruck ist kompatibel mit jedem Funktions-Interface
$\rightarrow$ der Compiler führt bestimmte Kompatibilitätschecks durch

Nicht jeder Labda-Ausdruck ist kompatibel mit jedem Funktions-Interface
$\rightarrow$ der Compiler führt bestimmte Kompatibilitätschecks durch

- Zieltyp `T` ist ein Funktions-Interface

Nicht jeder Labda-Ausdruck ist kompatibel mit jedem Funktions-Interface
$\rightarrow$ der Compiler führt bestimmte Kompatibilitätschecks durch

- Zieltyp `T` ist ein Funktions-Interface
- Lambda-Ausdruck hat die selbe Anzahl und Art von Parametern wie `T`

Nicht jeder Labda-Ausdruck ist kompatibel mit jedem Funktions-Interface
$\rightarrow$ der Compiler führt bestimmte Kompatibilitätschecks durch

- Zieltyp `T` ist ein Funktions-Interface
- Lambda-Ausdruck hat die selbe Anzahl und Art von Parametern wie `T`
- jeder zurückgegeben Ausdruck ist kompatibel mit dem definierten Rückgabewert von `T`

Nicht jeder Labda-Ausdruck ist kompatibel mit jedem Funktions-Interface
$\rightarrow$ der Compiler führt bestimmte Kompatibilitätschecks durch

- Zieltyp `T` ist ein Funktions-Interface
- Lambda-Ausdruck hat die selbe Anzahl und Art von Parametern wie `T`
- jeder zurückgegeben Ausdruck ist kompatibel mit dem definierten Rückgabewert von `T`
- jede geworfene Exception ist kompatibel mit den definierten Exceptions von `T`

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Zuweisungen und return-Anweisungen (Goetz, 2013)

```
Comparator<String> c;
c = (String s1, String s2) ->
     s1.compareToIgnoreCase(s2);

public Runnable toDoLater() {
    return () -> {
        System.out.println("later");
    };
}
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Zuweisungen und return-Anweisungen (Goetz, 2013)

```
Comparator<String> c;
c = (String s1, String s2) ->
     s1.compareToIgnoreCase(s2);

public Runnable toDoLater() {
    return () -> {
        System.out.println("later");
    };
}
```

→ Typ `T` ist vom selben Typ wie die Zuweisung/der zurückgegebene Wert

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Array-Initialisierung (Goetz, 2013)

```
filterFiles(
    new FileFilter[] {
        f -> f.exists(), f -> f.canRead(), f ->
             f.getName().startsWith("q")
    }
);
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Array-Initialisierung (Goetz, 2013)

```
filterFiles(
    new FileFilter[] {
        f -> f.exists(), f -> f.canRead(), f ->
            f.getName().startsWith("q")
    }
);
```

→ Typ `T` wird abgeleitet vom Array-Typ

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Körper von Lambda-Ausdrücken (Goetz, 2013)

```
Supplier<Runnable> c = () -> () -> {
    System.out.println("hi"); };
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Körper von Lambda-Ausdrücken (Goetz, 2013)

```
Supplier<Runnable> c = () -> () -> {
    System.out.println("hi"); };
```

→ Innerer Typ `T` wird vom äußeren Zieltyp abgeleitet

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Bedingte Ausdrücke (Goetz, 2013)

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Bedingte Ausdrücke (Goetz, 2013)

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```

→ Typ `T` wird von bedingtem Ausdruck “weitergeleitet”

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Casting (Goetz, 2013)

```
// Illegal: Object o = () -> {
     System.out.println("hi"); };
Object o = (Runnable) () -> { System.out.println("hi");
     };
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Casting (Goetz, 2013)

```
// Illegal: Object o = () -> {
     System.out.println("hi"); };
Object o = (Runnable) () -> { System.out.println("hi");
     };
```

→ Typ `T` durch Cast-Operator festgelegt

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Parameter einer Methode (Goetz, 2013)

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p ->
    p.getName());
```

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Parameter einer Methode (Goetz, 2013)

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p ->
    p.getName());
```

- Kompliziertester Fall: Kompatibilität zu Methoden-Überladung und Typ-Inferenz für Parameter muss gewährleistet sein

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Parameter einer Methode (Goetz, 2013)

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p ->
    p.getName());
```

- Kompliziertester Fall: Kompatibilität zu Methoden-Überladung und Typ-Inferenz für Parameter muss gewährleistet sein
- Compiler nutzt Wissen über den Lambda-Ausdruck um Typ `T` zu inferieren:

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Parameter einer Methode (Goetz, 2013)

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p ->
    p.getName());
```

- Kompliziertester Fall: Kompatibilität zu Methoden-Überladung und Typ-Inferenz für Parameter muss gewährleistet sein
- Compiler nutzt Wissen über den Lambda-Ausdruck um Typ `T` zu inferieren:
  - **explizite Typisierung:** Compiler kennt Parameter-Typen und return-Typ

Verschiedene Programmkontexte erfordern unterschiedliche Inferenzregeln für den Compiler:

Parameter einer Methode (Goetz, 2013)

```
List<Person> ps = ...
Stream<String> names = ps.stream().map(p ->
    p.getName());
```

- Kompliziertester Fall: Kompatibilität zu Methoden-Überladung und Typ-Inferenz für Parameter muss gewährleistet sein
- Compiler nutzt Wissen über den Lambda-Ausdruck um Typ `T` zu inferieren:
  - **explizite Typisierung:** Compiler kennt Parameter-Typen und return-Typ
  - **implizite Typisierung:** Compiler beachtet ausschließlich Anzahl der Parameter des Lambda-Ausdrucks

Instanziierung eines ActionListeners (Goetz, 2013)

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ui.dazzle(e.getModifiers());
    }
});
```

- Handhabung von `this` und anderen Bezeichnern
- Verwendung von lokalen non-`final` Variablen nicht möglich

Handhabung von `this` und anderen Bezeichnern

Variablengültigkeit in anonymen Klassen

```
Runnable selfPrinter = new Runnable() {
    public void run() {
        // prints result of Object.toString()
        System.out.println(toString());
        System.out.println(this.toString());
        // prints "Hello World!"
        System.out.println(OuterClass.this.toString());
    }
}
@Override
public string toString() {
    return "Hello World!";
}
```

Handhabung von `this` und anderen Bezeichnern

Variablengültigkeit in anonymen Klassen

```
Runnable selfPrinter = new Runnable() {
    public void run() {
        // prints result of Object.toString()
        System.out.println(toString());
        System.out.println(this.toString());
        // prints "Hello World!"
        System.out.println(OuterClass.this.toString());
    }
}
@Override
public string toString() {
    return "Hello World!";
}
```

- Ungewollte Überlagerung von Methoden durch Vererbung

Handhabung von `this` und anderen Bezeichnern

Variablengültigkeit in anonymen Klassen

```
Runnable selfPrinter = new Runnable() {
    public void run() {
        // prints result of Object.toString()
        System.out.println(toString());
        System.out.println(this.toString());
        // prints "Hello World!"
        System.out.println(OuterClass.this.toString());
    }
}
@Override
public string toString() {
    return "Hello World!";
}
```

- Ungewollte Überlagerung von Methoden durch Vererbung
- Unqualifizierter Aufruf von `this` referenziert innere Klasse

Handhabung von `this` und anderen Bezeichnern

Variablengültigkeit in anonymen Klassen

```
Runnable selfPrinter = () -> {
    // prints "Hello World!"
    System.out.println(toString());
    System.out.println(this.toString());
    System.out.println(OuterClass.this.toString());
}
@Override
public string toString() {
    return "Hello World";
}
```

- Ungewollte Überlagerung von Methoden durch Vererbung
  $\rightarrow$ keine vererbten Bezeichner von Supertypen
- Unqualifizierter Aufruf von `this` referenziert innere Klasse

Handhabung von `this` und anderen Bezeichnern

Variablengültigkeit in anonymen Klassen

```
Runnable selfPrinter = () -> {
    // prints "Hello World!"
    System.out.println(toString());
    System.out.println(this.toString());
    System.out.println(OuterClass.this.toString());
}
@Override
public string toString() {
    return "Hello World";
}
```

- Ungewollte Überlagerung von Methoden durch Vererbung
  $\rightarrow$ keine vererbten Bezeichner von Supertypen

- Unqualifizierter Aufruf von `this` referenziert innere Klasse
  $\rightarrow$ **Lexikalisches Scoping:** Referenzen haben die selbe Bedeutung wie außerhalb des Lambda-Ausdrucks

Lokale non-`final` Variablen

Effektiv `final` Variable (Goetz, 2013)

```
public Callable<String> helloCallable(String name) {
    String hello = "Hello";
    return new Callable<String>() {
        public String call() {
            return hello + ", " + name; // ERROR!
        }
    }
}
```

Lokale non-`final` Variablen

Effektiv `final` Variable (Goetz, 2013)

```
public Callable<String> helloCallable(String name) {
    String hello = "Hello";
    return new Callable<String>() {
        public String call() {
            return hello + ", " + name; // ERROR!
        }
    }
}
```

- Verwendung von lokalen non-`final` Variablen nicht erlaubt
  $\rightarrow$ Relaxierung der Einschränkungen für “effektiv” `final` Variablen

Lokale non-`final` Variablen

Effektiv `final` Variable (Goetz, 2013)

```
public Callable<String> helloCallable(String name) {
    String hello = "Hello";
    return () -> (hello + ", " + name); // no error
}
```

- Verwendung von lokalen non-`final` Variablen nicht erlaubt
  → Relaxierung der Einschränkungen für “effektiv” `final` Variablen

Lokale non-`final` Variablen

Effektiv `final` Variable (Goetz, 2013)

```
public Callable<String> helloCallable(String name) {
    String hello = "Hello";
    return () -> (hello + ", " + name); // no error
}
```

- Verwendung von lokalen non-`final` Variablen nicht erlaubt
  $\rightarrow$ Relaxierung der Einschränkungen für “effektiv” `final` Variablen

- **Aber:** Veränderbare Variablen sind weiterhin nicht erlaubt

Veränderbare lokale Variablen und Lambda-Ausdrücke

Veränderung lokaler Variablen in Lambda-Ausdrücken (Goetz, 2013)

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // ERROR
```

Veränderbare lokale Variablen und Lambda-Ausdrücke

Veränderung lokaler Variablen in Lambda-Ausdrücken (Goetz, 2013)

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // ERROR
```

- **Problem:** Ausdruck grundlegend seriell → problematisch im Bezug auf Race-Konditionen

Veränderbare lokale Variablen und Lambda-Ausdrücke

Veränderung lokaler Variablen in Lambda-Ausdrücken (Goetz, 2013)

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // ERROR
```

- **Problem:** Ausdruck grundlegend seriell → problematisch im Bezug auf Race-Konditionen
- **Alternative:** Behandle Problem als Reduktion

Reduktion mit `java.util.stream` (Goetz, 2013)

```
int sum = list.stream()
              .mapToInt(e -> e.size())
              .reduce(0, (x,y) -> x+y);
```

Referenzierung von Methoden

- Lambda-Ausdrücke vereinfachen die anonyme Instanziierung von funktionalen Interfaces

Referenzierung von Methoden

- Lambda-Ausdrücke vereinfachen die anonyme Instanziierung von funktionalen Interfaces
- **Ziel:** definiere vereinfachte Form um bereits existierende Methoden zu referenzieren

Methoden-Referenzen (Goetz, 2013)

```
Comparator<Person> byName
    = Comparator.comparing(p -> p.getName());
Arrays.sort(people, byName);
```

Referenzierung von Methoden

- Lambda-Ausdrücke vereinfachen die anonyme Instanziierung von funktionalen Interfaces
- **Ziel:** definiere vereinfachte Form um bereits existierende Methoden zu referenzieren

Methoden-Referenzen (Goetz, 2013)

```
Comparator<Person> byName
    = Comparator.comparing(Person::getName);
Arrays.sort(people, byName);
```

- Parametertypen der Methode des funktionalen Interfaces fungieren als Argumente eines impliziten Methodenaufrufs

Methoden-Referenzen Parametertypen

- Parametertypen der Methode des funktionalen Interfaces fungieren als Argumente eines impliziten Methodenaufrufs
- Manipulation der Parametertypen durch widening, boxing, etc. erlaubt

Methoden-Referenzen (Goetz, 2013)

```
// void exit(int status)
Consumer<Integer> b1 = System::exit;
// void sort(Object[] a)
Consumer<String[]> b2 = Arrays::sort;
// void main(String... args)
Consumer<String> b3 = MyProgram::main;
// void main(String... args)
Runnable r = MyProgram::main;
```

Unterschiedliche Methoden-Typen werden auf verschiedene Arten referenziert:

Statische Methode (Goetz, 2013)

```
KlassenName::methodenName
```

- Klassenbezeichner steht vor dem Trennzeichen.

Statische Methode (Goetz, 2013)

```
KlassenName::methodenName
```

- Klassenbezeichner steht vor dem Trennzeichen.

`super`-Methode (Goetz, 2013)

```
super::methodenName
```

- `super`-Schlüsselwort steht vor dem Trennzeichen

Methode eines bestimmten Objekts (Goetz, 2013)

```
objektName::methodenName
```

- Objektbezeichner steht vor dem Trennzeichen.

Methode eines bestimmten Objekts (Goetz, 2013)

```
objektName::methodenName
```

- Objektbezeichner steht vor dem Trennzeichen.
- bietet bequeme Art um zwischen verschiedenen funktionalen Interfaces zu konvertieren:

Interface-Konvertierung (Goetz, 2013)

```
Callable<Path> c = ...
PrivilegedAction<Path> a = c::call;
```

Methode eines beliebigen Objekts (Goetz, 2013)

```
KlassenName::methodenName
```

- Klasse des beliebigen Objekts steht vor dem Trennzeichen

Methode eines beliebigen Objekts (Goetz, 2013)

```
KlassenName::methodenName
```

- Klasse des beliebigen Objekts steht vor dem Trennzeichen
- Objekt auf dem die Methode ausgeführt wird, ist erster Parameter

Methode eines beliebigen Objekts (Goetz, 2013)

```
KlassenName::methodenName
```

- Klasse des beliebigen Objekts steht vor dem Trennzeichen
- Objekt auf dem die Methode ausgeführt wird, ist erster Parameter

Mehrdeutigkeit mit statischer Methode (Gosling et al., 2014)

```
class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    void test() { Function<C, Integer> f1 = C::size; }
}
```

Klassen-Konstruktor (Goetz, 2013)

```
KlassenName::new
```

- Klassenbezeichner steht vor und `new`-Schlüsselwort nach dem Trennzeichen

Klassen-Konstruktor (Goetz, 2013)

```
KlassenName::new
```

- Klassenbezeichner steht vor und `new`-Schlüsselwort nach dem Trennzeichen
- wenn Konstruktor überladen $\rightarrow$ Compiler wählt Konstruktor der beste Übereinstimmung mit der Zieltyp hat.
- Typ von generischen Klassen kann explizit angegeben oder inferiert werden

Array-Konstruktor (Goetz, 2013)

```
TypeName[]::new
```

- Typ-Bezeichner des Arrays steht vor und `new`-Schlüsselwort nach dem Trennzeichen

Array-Konstruktor (Goetz, 2013)

```
TypeName[]::new
```

- Typ-Bezeichner des Arrays steht vor und `new`-Schlüsselwort nach dem Trennzeichen
- werden behandelt wie Konstruktor mit einem einzelnen `int`-Parameter

Spezieller Array-Konstruktor (Goetz, 2013)

```
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10);
```

Standard-Methoden für Interfaces

- **Ziel:** Integration neuer Lambda-Funktionalität in vorhandene Frameworks

Standard-Methoden für Interfaces

- **Ziel:** Integration neuer Lambda-Funktionalität in vorhandene Frameworks
- **Problem:** Erweiterung von bereits existierenden Interfaces oder abstrakten Basisklassen problematisch

Standard-Methoden für Interfaces

- **Ziel:** Integration neuer Lambda-Funktionalität in vorhandene Frameworks
- **Problem:** Erweiterung von bereits existierenden Interfaces oder abstrakten Basisklassen problematisch

→ `default`-Methoden, die das “Standardverhalten” des Interfaces festlegen

`default`-Methode (Goetz, 2013)

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();

    default void skip(int i) {
        for (; i > 0 && hasNext(); i--) next();
    }
}
```

Ableiten von Standard-Methoden

- Ableitungsregeln äquivalent zu anderen Methoden

Ableiten von Standard-Methoden

- Ableitungsregeln äquivalent zu anderen Methoden
- **Problem:** Konfliktpotential wenn mehrere Supertypen `default`-Methoden mit identischer Signatur definieren

Ableiten von Standard-Methoden

- Ableitungsregeln äquivalent zu anderen Methoden
- **Problem:** Konfliktpotential wenn mehrere Supertypen `default`-Methoden mit identischer Signatur definieren
- Compiler versucht Konflikt aufzulösen, ansonsten Fehler beim kompilieren

Ableiten von Standard-Methoden

- Ableitungsregeln äquivalent zu anderen Methoden
- **Problem:** Konfliktpotential wenn mehrere Supertypen `default`-Methoden mit identischer Signatur definieren
- Compiler versucht Konflikt aufzulösen, ansonsten Fehler beim kompilieren

→ Programmierer wählt bevorzugte Implementierung aus entsprechendem Supertypen

Neue `super`-Syntax (Goetz, 2013)

```
interface Robot implements Artist, Gun {
    default void draw() { Artist.super.draw(); }
}
```

Ableiten von Standard-Methoden

- Klassenmethoden haben Priorität vor Interface-Methoden

Ableiten von Standard-Methoden

- Klassenmethoden haben Priorität vor Interface-Methoden
- Interface-Methoden die bereits einmal überschrieben wurden, werden vom Compiler ignoriert

Ableiten von Standard-Methoden

- Klassenmethoden haben Priorität vor Interface-Methoden
- Interface-Methoden die bereits einmal überschrieben wurden, werden vom Compiler ignoriert

Behandlung von Ableitungskonflikten (Gallardo et al., 2014)

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}
public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}
public interface FireBreather extends Animal {}
public class Dragon implements EggLayer, FireBreather {}
```

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  $\rightarrow$ Klassenmethode

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  → `default`-Methode

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  $\rightarrow$ `default`-Methode

- Funktionalität, die nicht Instanz-spezifisch ist

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  → `default`-Methode

- Funktionalität, die nicht Instanz-spezifisch ist
  → statische Klassenmethode

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  → `default`-Methode

- Funktionalität, die nicht Instanz-spezifisch ist
  → statische Interface-Methode

Statische Interface-Methoden

- Funktionalität, die auf allen Instanzen zur Verfügung stehen soll
  → `default`-Methode

- Funktionalität, die nicht Instanz-spezifisch ist
  → statische Interface-Methode

`default`-Methode (Goetz, 2013)

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

→ Reduziert Notwendigkeit für Nebenklassen, die als Sammlung von statischen Methoden fungieren

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung
- Lambda-Ausdrücke vereinfachen Syntax für anonyme Instanziierung

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung
- Lambda-Ausdrücke vereinfachen Syntax für anonyme Instanziierung
  - erweiterte Typinferenz um Lambda-Syntax zu ermöglichen

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung
- Lambda-Ausdrücke vereinfachen Syntax für anonyme Instanziierung
  - erweiterte Typinferenz um Lambda-Syntax zu ermöglichen
  - Vereinfachung des Variablen-Scopings

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung
- Lambda-Ausdrücke vereinfachen Syntax für anonyme Instanziierung
  - erweiterte Typinferenz um Lambda-Syntax zu ermöglichen
  - Vereinfachung des Variablen-Scopings
  - Lambda-kompatible Referenzierung nicht-anonymer Funktionen

Zusammenfassung

- **Ziel:** Definiere neuen, kompakten Standard für anonyme Methoden in Java
- Funktions-Interfaces als Instanz nominaler Typisierung
- Lambda-Ausdrücke vereinfachen Syntax für anonyme Instanziierung
  - erweiterte Typinferenz um Lambda-Syntax zu ermöglichen
  - Vereinfachung des Variablen-Scopings
  - Lambda-kompatible Referenzierung nicht-anonymer Funktionen
- Kompatibilität mit vorhandenen Frameworks durch statische und `default`-Methoden in Interfaces

References

Gallardo, R., S. Hommel, S. Kannan, J. Gordon, and S. B. Zakhour
2014. *The Java Tutorial: A Short Course on the Basics (6th Edition)*, 6th edition. Addison-Wesley Professional.

Goetz, B.
2013. State of the lambda. Website. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html`; abgerufen am 25.04.2019.

Gosling, J., B. Joy, G. L. Steele, G. Bracha, and A. Buckley
2014. *The Java Language Specification, Java SE 8 Edition*, 1st edition. Addison-Wesley Professional.

Kyle, J.
2016. Type systems: Structural vs. nominal typing explained. Website. `https://medium.com/@thejameskyle/type-systems-structural-vs-nominal-typing-explained-56511dd969f4`; abgerufen am 25.04.2019.

Fragen und Diskussion