HAW HAMBURG

B-AI2 PM2

Lambdas

Thanks to Marty Hall,hall@coreservlets.com

Übersicht dieses Abschnitts

- Einführung
  - Motivation
  - Die Grundidee
- Neue Option: Lambda-Ausdrücke
  - Interpretation
  - Die einfachste Form
  - Typ-Erkennung
  - Ausdrücke für den Lambda-Body
  - Weglassen von Klammern
  - Lambda und Alternativen: Vergleich
- Beispiele
  - Numerische Integration
  - Utilities für Zeitmessungen

Lernziele

- Grundlegende funktionale Programmierprinzipien beherrschen
- Lambda-Ausdrücke in Java verwenden können
- Transferkompetenz entwickeln (hier: Ruby-Wissen über Blöcke nach Java transferieren)
- Übliche Stile bei der Verwendung von Lambda-Ausdrücken kennen
- Notwendige Teile der dazu genutzten Java Klassenbibliothek kennen

Funktionen als Parameter sind Standard

- In dynamischen (meist schwach) typisierten Sprachen
  - JavaScript, Lisp, Scheme, etc.
- In stark typisierten (dynamisch oder nicht)
  - Ruby, Scala, Clojure, ML, etc.
- Funktionaler Ansatz ist bewährt: präzise, flexibel und parallelisierbar
  - Sortieren mit JavaScript

```
var testStrings = ["one", "two", "three", "four"];
testStrings.sort(function(s1, s2) {
                    return(s1.length - s2.length);});
testStrings.sort(function(s1, s2) {
                    return(s1.charCodeAt(s1.length - 1) -
                           s2.charCodeAt(s2.length - 1));});
```

Warum Lambdas in Java jetzt?

- Präzise Syntax
  - Prägnanter und klarer als anonyme innere Klassen
- Schwächen anonymer innerer Klassen
  - Schwerfällig, Verwechslungsgefahr bei „this” und generell bei Namen, kein Zugriff auf nicht finale lokale Variablen, schwer zu optimieren
  - Darum brauchen Sie sich nicht mehr zu kümmern
- Sehr praktisch für die neue Stream-Bibliothek
  - shapes.forEach(s -> s.setColor(Color.RED));
- Viele Programmierer kennen den Ansatz
  - Callbacks, Closures, map/reduce Idiom
- Vereinfachen den Code

Lambdas: Prägnant und ausdrucksstark

- Früher

```
button.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
     doSomethingWith(e);
   }
});
```

- Jetzt

```
button.addActionListener(e -> doSomethingWith(e));
```


"Vigorous writing is concise... This requires not that the writer make all sentences short, or avoid all details and treat subjects only in outline, but that every word should tell." – Strunk and White, *The Elements of Style.*

Lambdas: Neue Denkweisen

- Ermutigen zu funktionaler Programmierung
  - Mit funktionaler Programmierung sind manche Arten von Problemen einfacher zu lösen und der Code besser zu lesen und einfacher zu warten
    - Funktionale Programmierung ersetzt keinesfalls objekt-orientierte Programmierung in Java. OOP ist weiterhin *der* Ansatz zur Darstellung von Typen. Funktionale Elemente ergänzen und verbessern aber viele Methoden und Algorithmen.
- Unterstützung von Streams
  - Streams sind Hüllen um Datenquellen (Arrays, Collections, etc.), die Lambdas verwenden, map/filter/reduce unterstützen, lazy evaluation verwenden und können automatisch parallelisiert werden.
    - For each Schleife kann *nicht* automatisch parallelisiert werden

      for(Employee e: employees) { e.giveRaise(1.15); }
    - forEach Methode kann automatisch parallelisiert werden

      employees.stream().parallel().forEach(e -> e.giveRaise(1.15));//Schön wär's

Kernpunkte

- Sie schreiben etwas, das wie eine Funktion aussieht
  - `Arrays.sort(testStrings, (s1, s2) -> Integer.compare(s1.length(), s2.length());`
  - `taskList.execute(() -> downloadSomeFile());`
  - `someButton.addActionListener(event -> handleButtonClick());`
  - `double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);`
- Sie bekommen aber ein Objekt einer Klasse, die das erwartete Interface implementiert
  - Der erwartete Typ muss ein Interface mit *genau einer* (abstrakten) Methode sein
    - Nennt man “Functional Interface” oder “Single Abstract Method (SAM) Interface”
      - Die Bedingung einer einzelnen ABSTRAKTEN Methode ist nicht überflüssig, denn in Java 8 können Interfaces konkrete Methoden haben (“default Methoden”). Java Interfaces können auch statische Methoden haben.

Zusammenfassung Syntax

- Ersetze eine anonyme innere Klasse

```
doSomething(new SomeOneMethodInterface() {
  @Override
  public SomeType methodOfInterface(args) {
    return(value);
  }
});
```

- Durch einen Lambda-Ausdruck

```
doSomething((args) -> value);
```

Beispiel: Sortiere Strings nach Länge

- Anonyme Klasse

```
Arrays.sort(testStrings, new Comparator<String>() {
   @Override
   public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
});
```

- Lambda

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() -
s2.length());
```

Hier wird die anonyme Klasse einfach durch ein Lambda ersetzt. Es gibt aber weitere Methoden, z.B. Comparator.comparing.
Folgendes geht also auch: Arrays.sort(testStrings, Comparator.comparing(String::length));
Methodenreferenzen wie diese werden im nächsten Abschnitt besprochen.. Comparator.comparing und ähnliche Methoden werden im Abschnitt Lambdas und Functionen höherer Ordnung besprochen..

Von hier nach dort: Schritt 1 – Entferne Interface und Methodennamen

- Idee
  - Aus dem API ist der zweite Parameter von Arrays.sort bekannt: Es ist ein Comparator, also brauchen wir das nicht anzugeben. Comparator hat nur eine abstrakte Methode, wir brauchen also nicht anzugeben, dass sie „compare" heißt.
  - Ergänze „->" (-, dann > Zeichen) zwischen Methodenparametern und -rumpf
- Java-Beispiel vor Version 8

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
```

- Java 8 Alternative (legal, aber noch nicht ideal)

```
Arrays.sort(testStrings,
  (String s1, String s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

Von hier nach dort: Schritt 2 – Entferne Deklarationen der Parameter-Typen

- Idee
  - Aus dem ersten Argument (testStrings), kann Java den Typparameter beim zweiten Argument erkennen: Comparator<String>. Also sind beide Parameter von compare vom Typ String und das muss nicht hingeschrieben werden.
  - Java macht weiterhin starke, compile-time Typprüfung. Der Compiler erschließt aber einige Typen. Ähnlich wie beim „diamond operator".
    - `List<String> words = new ArrayList<>();`
  - In wenigen Fällen ist dies nicht möglich. Dann warnt der Compiler, und Sie können die Typen nicht weglassen.
- Frühere Version

```
Arrays.sort(testStrings,
  (String s1, String s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

- Verbesserte Version (legal, aber noch nicht ideal)

```
Arrays.sort(testStrings,
  (s1, s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

Von hier nach dort: Schritt 3 – Verwende Ausdrücke statt Block

- Idee
  - Kann der Methodenrumpf als ein einzelnes return Statement geschreiben werden, können Sie die geschweiften Klammern und return weglassen und nur den return Wert als Ausdruck hinschreiben.
  - Das geht nicht immer, etwa bei Schleifen oder if. Lambdas werden aber meist verwendet, wenn der Methodenrumpf kurz ist, dann macht man das so. Wenn nicht, ist es legal, Klammern und return zu verwenden. Sie können aber auch erwägen weiterhin eine innere Klasse oder auch eine Methodenreferenz zu verwenden.
- Frühere version

```
Arrays.sort(testStrings,
  (s1, s2) -> { return Integer.compare(s1.length(), s2.length())); });
```

- Verbesserte Version (empfohlen)

```
Arrays.sort(testStrings, (s1, s2)
-> Integer.compare(s1.length(), s2.length()));
```

Optionaler Schritt 4 – Wenn genau ein Parameter: Keine Klammern

- Idee
  - Hat die (abstrakte) Methode des Interfaces *genau* einen Parameter, so sind die Klammern optional
- Java vor Version 8

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- Java ab 8 mit Klammern

```
button.addActionListener((e) -> doSomethingWith(e));
```

- Java ab 8 ohne Klammern

```
button.addActionListener(e -> doSomethingWith(e));
```

Lambda Syntax - Zusammenfassung

- Lasse Interface- und Methodennamen weg

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override public int compare(String s1, String s2) { return Integer.compare(s1.length() , s2.length())); }
});
```

Ersetzt durch

```
Arrays.sort(testStrings, (String s1, String s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

- Lasse Parametertypen weg

```
Arrays.sort(testStrings, (String s1, String s2) -> { return Integer.compare(s1.length() ,s2.length()); });
```

Ersetzt durch

```
Arrays.sort(testStrings, (s1, s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

- Verwende Ausdrücke statt Blöcke

```
Arrays.sort(testStrings, (s1, s2) -> { return Integer.compare(s1.length(), s2.length()); });
```

Ersetzt durch

```
Arrays.sort(testStrings, (s1, s2) ->Integer.compare( s1.length(), s2.length()));
```

- Lasse Klammern bei nur einem Parameter weg

```
button1.addActionListener((event) -> popUpSomeWindow(...));
```

Ersetzt durch

```
button1.addActionListener(event -> popUpSomeWindow(...));
```

Java vor und ab Version 8

- Java vor Version 8

```
taskList.execute(new Runnable() {
   @Override
   public void run() {
     processSomeImage(imageName);
   }
 });
 button.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent event) {
     doSomething(event);
   }
 });
```

- Java 8

```
taskList.execute(() -> processSomeImage(imageName));
button.addActionListener(event -> doSomething(event));
```

Java vs. JavaScript

- Java

```
String[] testStrings = {"one", "two", "three", "four"};
Arrays.sort(testStrings,
            (s1, s2) -> s1.length() - s2.length());
Arrays.sort(testStrings,
            (s1, s2) -> s1.charAt(s1.length() - 1) -
                        s2.charAt(s2.length() - 1));
```

- JavaScript

```
var testStrings = ["one", "two", "three", "four"];
testStrings.sort(function(s1, s2) {
                    return(s1.length - s2.length);});
testStrings.sort(function(s1, s2) {
                  return(s1.charCodeAt(s1.length - 1) -
                         s2.charCodeAt(s2.length - 1));
                  });
```

Blick unter die „Motorhaube”

- Sie schreiben
  - `Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());`
- Tatsächlich passiert dies:
  - Sie benutzen eine Kurzschreibweise für ein Objekt einer Klasse, die Comparator<T>. implementiert. Sie liefern den Rumpf der compare Methode nach dem „->”.
- Sie stellen sich vor:
  - Sie übergeben die Vergleichsfunktion.
- Funktionstypen
  - Java hat technisch *keine* Funktionstypen. Intern werden Lambdas zu Objekten von Klassen, die das erwartete (funktionale) Interface implementieren. Trotzdem stellt man sich Lambdas als Funktionen vor.

Verwendung von Lambdas

- Finde irgend eine Stelle (Variable, Parameter) wo ein Interface mit genau einer (abstrakten) Methode erwartet wird
    - Technisch: Eine abstrakte Methode. In Java vor Version 8 kein Unterschied zwischen 1-method interface und 1-abstract-method interface. Letztere heißen “funktionale Interfaces” oder “SAM (Single Abstract Method) interfaces”.
  - public interface Blah { String foo(String someString); }
- Code der der Interface benutzt ist der gleiche
  - public void someMethod(**Blah** b) { … b.**foo**(…) …}
    - Code, der das Interface verwendet muss immer noch den Methodennamen kennen
- Code, der die Methode Aufrufen will, kann erwarten, dass das Interface die Methode spezifiziert
  - someMethod(**s -> s.toUpperCase() + "!"**);

Beispiel: Numerische Integration

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

Beispiel: Numerische Integration

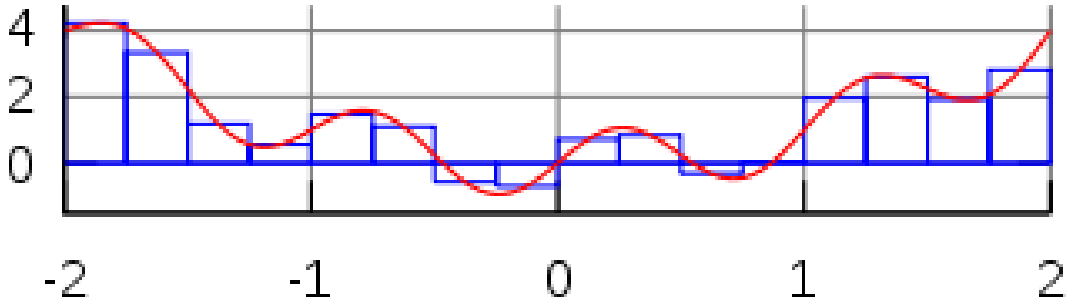
- Ziel
  - Einfache numerische Integration mit Mittelpunktsregel
  - 

    Diagramm aus http://en.wikipedia.org/wiki/Numerical_integration
  - Lambda-Ausdrücke sind das Mittel der Wahl, um die Funktion anzugeben.
    - Brauchen ein funktionales (SAM) Interface mit einer „double eval(double x)” Methode für die zu integrierende Funktion.

Interface

```
public interface Integrable {
   double eval(double x);
}
```

Im Folgenden, werden wir dies Beispiel weiterentwickeln:.

- Erstens werden wir die optionale, aber nützliche Annotation @FunctionalInterface ergänzen.
- Second, we will observe that there is already a compatible interface built into the java.util.function package, and use it instead.

Einfache numerische Integration

```
public static double integrate(Integrable function,
                               double x1, double x2,
                               int numSlices){
  if (numSlices < 1) {
    numSlices = 1;
  }
  double delta = (x2 - x1)/numSlices;
  double start = x1 + delta/2;
  double sum = 0;
  for(int i=0; i<numSlices; i++) {
    sum += delta * function.eval(start + delta * i);
  }
  return(sum);
}
```

Methode zum Testen

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
  for(int i=1; i<7; i++) {
    int numSlices = (int)Math.pow(10, i);
    double result =
      MathUtilities.integrate(function, x1, x2, numSlices);
    System.out.printf("  For numSlices =%,10d
                      result = %,.8f%n",
                      numSlices, result);
  }
}
```

Testergebnisse

```
MathUtilities.integrationTest(x -> x*x, 10, 100);
MathUtilities.integrationTest(x -> Math.pow(x,3), 50, 500);
MathUtilities.integrationTest(x -> Math.sin(x), 0,
Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2, 20);
```

Output

```
Estimating integral of x^2 from 10.000 to 100.000.
Exact answer = 100^3/3 - 10^3/3.
                                    ~= 333,000.00000000
  For numSlices =         10 result = 332,392.50000000
  For numSlices =        100 result = 332,993.92500000
  For numSlices =      1,000 result = 332,999.93925000
  For numSlices =     10,000 result = 332,999.99939250
  For numSlices =    100,000 result = 332,999.99999393
  For numSlices = 1,000,000 result = 332,999.99999994

... // Similar for other three integrals
```

Allgemeine Lambda Prinzipien

- Interfaces in Java 8 unverändert
  - Integrable wäre in Java 7 genauso, außer, dass Sie @FunctionalInterface verwenden sollten
    - Diese Annotation wird im nächsten Abschnitt behandelt
- Code, der Interfaces *benutzt* ist in Java 8 und früheren Versionen identisch
  - Die Definition von integrate ist genau so, wie sie es in Java vor Version 8 wäre. Der Autor von integrate muss wissen, das der Methodenname eval ist.
- Code, der Methoden *aufruft*, die SAM-Interfaces erwarten, kann nun Lambdas verwenden

```
MathUtilities.integrate(x -> Math.sin(x), 0, Math.PI, …);
```

Statt: new Integrable() { public void eval(double x) { return(Math.sin(x)); } }

Ein wiederverwendbarer Zeitmesser

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Zeitmessung

- Ziele
  - Übergebe eine „Funktion” als Parameter
  - Führe die Funktion aus
  - Ermittle die verstrichene Zeit
- Problem: Java wertet die Parameter eines Aufrufs aus
  - TimingUtils.timeOp(someSlowCalculation());
    - Die Berechnung ist beendet, *bevor* timeOp aufgerufen wird!
- Lösung: Lambdas
  - TimingUtils.timeOp(() -> someSlowCalculation());
    - timeOp kann die Berechnung intern ausführen
- Geht auch mit inneren Klassen
  - Siehe z.B. den Abschnitt über fork-join
  - Aber, der entsprechende Code timeOp ist, lang mühselig und schwer lesbar

Das Interface Op

```
public interface Op {
   void runOp();
}
```

In späteren Abschnitten, werden wir das Beispiel weiterentwickeln:

- Erstens, die optionale, aber nützliche Annotation @FunctionalInterface ergänzen.
- Zweitens, da Java Interfaces statische Methoden haben können, werden wir die statiische timeOp Methode aus TimingUtils in in dieses Interface verschieben.
- Dirttens, da Java Interfaces konkrete Methoden (“default methods”) haben können, werden wir eine Methode combinedOp einführen, die es uns ermöglicht, zwei Ops zu einer zusammenzufassen..

Die Klasse TimingUtils

```
public class TimingUtils {
  private static final double ONE_BILLION = 1_000_000_000;

  public static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    operation.runOp();
    long endTime = System.nanoTime();
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf("  Elapsed time: %.3f seconds.%n",
                      elapsedSeconds);
  }
}
```

Kern Test-Code

```
public class TimingTests {
  public static void main(String[] args) {
    for(int i=3; i<8; i++) {
      int size = (int)Math.pow(10, i);
      System.out.printf("Sorting array of length %,d.%n
size);
      TimingUtils.timeOp(
        () -> sortArray(size));
    }
  }
```

Output

```
Sorting array of length 1,000.
  Elapsed time: 0.002 seconds.
Sorting array of length 10,000.
  Elapsed time: 0.004 seconds.
Sorting array of length 100,000.
  Elapsed time: 0.020 seconds.
Sorting array of length 1,000,000.
  Elapsed time: 0.148 seconds.
Sorting array of length 10,000,000.
  Elapsed time: 1.339 seconds.
```

Unterstützender Test-Code

```
public static double[] randomNums(int length) {
  double[] nums = new double[length];
  for(int i=0; i<length; i++) {
    nums[i] = Math.random();
  }
  return(nums);
}

public static void sortArray(int length) {
  double[] nums = randomNums(length);
  Arrays.sort(nums);
}
```

Abschließende Lambda Beispiele

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Ein paar mehr Beispiele

- Als Argumente von Methoden

```
Arrays.sort(testStrings,
            (s1, s2) -> s1.length() - s2.length());
taskList.execute(() -> downloadSomeFile());
button.addActionListener(event ->
handleButtonClick());
double d = MathUtils.integrate(x -> x*x, 0, 100,
1000);
```

Ein paar mehr Beispiele (Forts.)

- Als Variablen (Zeigt den Typ leichter erkennbar)

```
AutoCloseable c = () -> cleanupForTryWithResources();
Thread.UncaughtExceptionHandler handler =
  (thread, exception) -> doSomethingAboutException();
Formattable f =
  (formatter, flags, width, precision) ->
makeFormattedString();
ContentHandlerFactory fact =
  mimeType -> createContentHandlerForMimeType();
CookiePolicy policy =
  (uri, cookie) -> decideIfCookieShouldBeAccepted();
Flushable toilet = () -> writeBufferedOutputToStream();
TextListener t = event -> respondToChangeInTextValue();
```

Zusammenfassung: Grundideen

- Toll! Jetzt haben wir λ‘s
  - Kurz und prägnant
  - Passen zu vorhandenen APIs
  - Wer funktionale Programmierung kennt, kann das schon
  - Passen gut zum neuen Streams API
  - Es gibt Methodenreferenzen und bereitgestellte funktionale Interfaces
- Buh! Warum keine “volle” funktionale Programmierung (?)
  - Typ eines λ ist eine Klasse, die ein Inter-face implementiert, keine “echte” Funktion
    - Interface müssen Sie haben, Methodennamen müssen Sie (manchmal) kennen
  - Mutable lokale Variablen nicht verwendbar

Themen dieses Abschnitts

- Die Annotation @FunctionalInterface
- Methodenreferenzen
- Gültigkeits regeln für Lambdas
- Effektive finale lokale Variablen

Annotation @FunctionalInterface

- Findet Fehler zur compile-Zeit
  - Fügt ein Entwickler eine weitere abstrakte Methode ein, gibt es einen Compiler-Fehler für das Interface
- Bring die Entwurfsabsicht klar zum Ausdruck
  - Teilt anderen Entwicklern mit, dass dieses Interface ein Ziel für Lambdas sein soll
- Ist aber nicht zwingend erforderlich, wie @Override
  - Lambdas können überall verwendet werden, wo ein SAM-Interface erartet wird, unabhängig von der Annotation @FunctionalInterface

Interface für numerische Integration

- Letzter Abschnitt

```
public interface Integrable {
  double eval(double x);
}
```

- Nun

```
@FunctionalInterface
public interface Integrable {
  double eval(double x);
}
```

Interface in TimingUtilities

- Letzter Abschnitt

```
public interface Op {
  void runOp();
}
```

- Nun

```
@FunctionalInterface
public interface Op {
  void runOp();
}
```

Method References

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Methodenreferenzen - Grundlagen

- Einfachster Fall: statische Methoden
  - Ersetze

    ```
    (args) -> ClassName.staticMethodName(args)
    ```

  - durch

    ```
    ClassName::staticMethodName
    ```

    - Beispiele: Math::cos, Arrays::sort, String::valueOf
  - Das heißt: Gibt es die benötige Methode bereits, so können Sie sie einfach aufrufen, ohne einen Lambda-Ausdruck hinschreiben zu müssen
  - Die Signatur der referenzierten Methode muss der der abstrakten Methoden im SAM Interface entsprechen
- Andere Methodenreferenzen werden Sie bald kennen lernen
  - variable::instanceMethod (e.g., System.out::println)
  - Class::instanceMethod (e.g., String::toUpperCase)
  - ClassOrType::new (e.g., String[]::new)

Beispiel: Numerische Integration

- In früherem Beispiel erste

```
MathUtilities.integrationTest(x -> Math.sin(x), 0,
Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2,
20);
```

- durch

```
MathUtilities.integrationTest(Math::sin, 0, Math.PI);
MathUtilities.integrationTest(Math::exp, 2, 20);
```

Typ einer Methodenreferenz

- Frage: Welchen Typ hat Math::sin?
  - Double? Function? Math?
- Antwort: Kann nur aus dem Kontext bestimmt werden
  - Eine bessere Frage ist: „welchen Typ hat Math::sin *in folgendem Code*?"
    - MathUtilities.integrationTest(Math::sin, 0, Math.PI);
  - Dazu müssen wir uns nur das API ansehen.
  - Ergebnis: Der Typ ist hier Integrable
    - In anderem Kontext, kann Math::sin einen anderen Typ haben!
- Dies gilt für alle Lambdas, nicht nur Methodenreferenzen
  - Der Type kann nur aus dem Kontext bestimmt werden

Der Typ von Lambdas bzw. Methoden-referenzen

- Interfaces
  - public interface Foo { double method1(double d); }
  - public interface Bar { double method2(double d); }
  - public interface Baz { double method3(double d); }
- Methoden, die das Interface verwenden
  - public void blah1(Foo f) { … f.method1(…)… }
  - public void blah2(Bar b) { … b.method2(…)… }
  - public void blah3(Baz b) { … b.method3(…)… }
- Aufruf der Methode (λs oder Methodenreferenzen)
  - blah1(Math::cos) *or* blah1(d -> Math.cos(d))
  - blah2(Math::cos) *or* blah2(d -> Math.cos(d))
  - blah3(Math::cos) *or* blah3(d -> Math.cos(d))
    - Geht auch mit Math::sin, Math::log, Math::sqrt, Math::abs, etc.

Bedeutung von Methodenreferenzen

- Niedrig!
  - Verstehen Sie dies nicht, so können Sie immer lambdas verwenden
  - Ersetze foo(Math::cos) durch foo(d -> Math.cos(d))
  - Ersetze bar(System.out::println) durch bar(s -> System.out.println(s))
  - Ersetze baz(Class::twoArgMethod) durch (a, b) -> Class.twoArgMethod(a, b)
- Aber Methodenreferenzen sind populär und die Empfehlung ist: Bevorzugen Sie Methodenreferenzen gegenüber Lambdas!
  - Prägnanter
  - Aus anderen Sprachen bereits bekannt, wie JavaScript, C in denen man existierende Methoden/Funktionen direkt ansprechen kann

```
function square(x) { return x*x; }
var f = square;
f(10); → 100
```

Vier Arten von Methodenreferenzen

Art	Beispiel	< = > Lambda
SomeClass::staticMethod	Math::cos	x -> Math.cos(x)
someObject::instanceMethod	someString::toUpperCase	() -> someString.toUpperCase()
SomeClass::instanceMethod	String::toUpperCase	s -> s.toUpperCase()
SomeClass::new	Employee::new	() -> new Employee()

var::instanceMethod vs. Class::instanceMethod

- someObject::instanceMethod
  - Erzeugt ein lambda mit *genauso vielen* Argumenten wie die Methode braucht

```
String test = "PREFIX:";
String result1 = transform(someString, test::concat);
```

- The concat method takes one arg
- This lambda takes one arg, passing s as argument to test.concat
- Equivalent lambda is s -> test.concat(s)

- SomeClass::instanceMethod
  - Erzeugt ein lambda, mit *einem* Argument *mehr,* als die Methode erwartet. Das erste Argument ist das Objekt, für das die Methode aufgerufen wird; der Rest sind die Parameter für die Methode.

```
String result2= transform(someString, String::toUpperCase);
```

- The toUpperCase method takes zero args
- This lambda takes one arg, invoking toUpperCase on that argument
- Equivalent lambda is s -> s.toUpperCase()

Methodenreferenz: Beispiel Helper Interface

```
@FunctionalInterface
public interface StringFunction {
   String applyFunction(String s);
}
```

Methodenreferenz: Beispiel Helper Interface

```
public class Utils {
  public static String transform(String s, StringFunction f) {
    return(f.applyFunction(s));
  }

  public static String makeExciting(String s) {
    return(s + "!!");
  }

  private Utils() {}
}
```

Methodenreferenzen: Code ausführen

```
public static void main(String[] args) {
  String s = "Test";

  // SomeClass::staticMethod
  String result1 = Utils.transform(s, Utils::makeExciting);
  System.out.println(result1);

  // someObject::instanceMethod
  String prefix = "Blah";
  String result2 = Utils.transform(s, prefix::concat);
  System.out.println(result2);

  // SomeClass::instanceMethod
  String result3 = Utils.transform(s, String::toUpperCase);
  System.out.println(result3);
}
```

```
Test!!
```

```
BlahTest
```

```
TEST
```

Vorschau: Constructor References

- In Java war es bisher etwas komplizierter zur Laufzeit über die Klasse zur Erzeugung eines Objekts zu entscheiden
  - Wollen Sie etwa ein Array oder eine Liste mit zufälligen Formen füllen (Tetris) Circle, Square, Rectangle
  - Dann brauchen Sie reflexiven Code (siehe hierzu später)
- Das geht jetzt einfach
  - Lege ein Array von Konstruktor Referenzen an und wähle daraus zufällig
    - `{ Circle::new, Square::new, Rectangle::new }`
  - Siehe hierzu auch das Interface Supplier

Vorschau: Generiere Zufallsdaten

```
private final static Supplier[] peopleGenerators =
  { Person::new, Writer::new, Artist::new, Consultant::new,
    EmployeeSamples::randomEmployee,
    () -> { Writer w = new Writer();
            w.setFirstName("Ernest");
            w.setLastName("Hemingway");
            w.setBookType(Writer.BookType.FICTION);
            return(w); }
  };

public static Person randomPerson() {
  Supplier<Person> generator =
    RandomUtils.randomElement(peopleGenerators);
  return(generator.get());
}
```

Vorschau: Array Constructor References

- Bald lernen Sie Streams kennen. Die haben toArray
  - ```
    Employee[] employees =
       employeeStream.toArray(Employee[]::new);
    ```
- Dies ist ein Spezialfall einer Konstruktor Referenz
  - toArray erhält eine IntFunction, die abstrakte Methode bekommt ein int als Argument, es wird also „new Employee[n]” aufgerufen. Das leere Employee Array wird mit den Elementen des Streams gefüllt.
- Allgemein formuliert
  - toArray erhält ein lambda oder eine Methodenreferenz auf irgend etwas, das ein int erhält und erzeugt ein Array von korrektem Typ und korrekter Länge
    - Das Array wird dann durch toArray gefüllt
    ```

Gültigkeitsbereich und Lambdas

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Das Wichtigste

- Lambdas are lexically scoped
  - Es wird keine neue Ebene eingeführt
- Konsequenzen
  - „this” referenziert die umschließende (outer) Klasse
  - Es gibt keinen Zugriff auf das umschließende Objekt
    - Es sei denn, das lambda ist Bestandteil einer Inneren Klasse
  - Innerhalb von Lambdas können keine „neuen” Variablen mit dem gleichen Namen vorkommen, wie solche im umschließenden Code
    - Aber Lambdas können auf lokale Variablen der umschließende Methode lesend, aber nicht ändernd zugreifen
  - Lambdas können auf Instanzvariablen der umschließenden Klassen lesend und ändernd zugreifen

Beispiele

- Illegal: repeated variable name

```
double x = 1.2;
someMethod(x -> doSomethingWith(x));
```

- Illegal: repeated variable name

```
double x = 1.2;
someMethod(y -> { double x = 3.4; … });
```

- Illegal: lambda modifying local var from the outside

```
double x = 1.2;
someMethod(y -> x = 3.4);
```

- Legal: modifying instance variable

```
private double x = 1.2;
public void foo() {  someMethod(y -> x = 3.4); }
```

- Legal: local name matching instance variable name

```
private double x = 1.2;
public void bar() {  someMethod(x -> x + this.x); }
```

Effectively Final Local Variables

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

Das Wichtigste

- Lambdas können auf lokale Variablen zugreifen, die nicht final sind, aber niemals verändert werden
  - Diese nennt man „effectively final” – Variable bei denen es legal wäre, sie als final zu deklarieren
  - In Lambdas kann auf mutable *Instanz*variablen zugegriffen werden
    - „this” referenziert Objekt der umschließenden Klasse
    - Keine Referenz auf Objekt der um schließenden Klasse.
- Mit expliziter Deklaration (explizit final)

```
final String s = "...";
doSomething(someArg -> use(s));
```

- Effectively final (ohne explizite Deklaration)

```
String s = "...";
doSomething(someArg -> use(s));
```

  - Seit Java 8 gilt das analog auch für anonyme innere Klassen

Beispiel: Button Listeners

```
public class SomeClass ... {
  private Container contentPane;

  private void someMethod() {
    button1.addActionListener(event ->
contentPane.setBackground(Color.BLUE));
    Color b2Color = Color.GREEN;
    button2.addActionListener(event -> setBackground(b2Color));
    button3.addActionListener(event -> setBackground(Color.RED));
    ...
  }
  ...
}
```

Instance variable: same rules as with anonymous inner classes in older Java versions; they can be modified.

Local variable: need not be explicitly declared final, but cannot be modified; i.e., must be "effectively final".

Beispiel: Paralleler Bild Download

- Idea
  - Verwende Java Threads um einige Bilder (Internet Cafes weltweit) herunterzuladen und blätterbar anzuzeigen
- Java 8
  - ExecutorService.execute erwartet ein Runnable, Runnable ist ein funktionales Interface => Lambdas sinnvoll einsetzbar
  - So haben Sie Zugriff auf lokale Variablen (so diese effectively final sind)

Main Code

```
  ...
  ExecutorService taskList =
Executors.newFixedThreadPool(poolSize);
  for(int i=1; i<=numImages; i++) {
    JLabel label = new JLabel();
    URL location = new URL(String.format(imagePattern, i));
    taskList.execute(() -> {
      ImageIcon icon = new ImageIcon(location);
      label.setIcon(icon);
    });
    imagePanel.add(label);
  }
  ...
```

Full code can be downloaded from
http://www.coreservlets.com/java-8-tutorial/

Multithreaded version takes less than half the time of the single-threaded version.
Speedup could be much larger if the images were taken from different servers.

Zusammenfassung

- @FunctionalInterface
  - Verwenden für jedes Interface, das dauerhaft nur eine abstrakte Methode hat
- Methodenreferenz
  - `arg -> Class.method(arg)` → `Class::method`
- Gültigkeitsbereiche
  - Lambdas führen keine neue Ebene ein
  - „this” bezieht sich immer auf das Objekt der umschließenden Klasse
- Effektiv finale lokale Variablen
  - Lambdas können lesend aber nicht ändernd auf lokale Variable der umschließenden Methode zugreifen
  - Diese Variablen müssen nicht explizit als final deklariert werden
  - Dies gilt seit Java 8 auch für anonyme innere Klassen

Lambda Expressions in Java : Lambda Bausteine in java.util.function

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

λ

Themen in diesem Abschnitt

- Lambdas: Basis im Paket java.util.function
  - Versionen für einige primitive Typen
    - *Blah*UnaryOperator, *Blah*BinaryOperator, *Blah*Predicate, *Blah*Consumer
  - Generische Versionen
    - Predicate
    - Function
    - BinaryOperator
    - Consumer
    - Supplier

Das Wichtigste

- java.util.function: Viele wiederverwendbare Interfaces
  - Obwohl sie „nur“ Interfaces sind, werden sie behandelt, als wenn sie Funktionen seien
- Nicht generische Interfaces
  - IntPredicate, LongUnaryOperator, DoubleBinaryOperator, etc.
- Generische Interfaces
  - Predicate<T> — T in, boolean out
  - Function<T,R> — T in, R out
  - Consumer<T> — T in, nothing (void) out
  - Supplier<T> — Nothing in, T out
  - BinaryOperator<T> — Two T’s in, T out

Einfache Bausteine

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Das Wichtigste

- Interfaces wie Integrable sind weit verbreitet
  - In Java wurden viele übliche Fälle aufgenommen
- Können in vielen Situationen direkt verwendet werden
  - Sie brauchen also einen allgemeineren Namen, als “Integrable”
- java.util.function definiert viele einfache funktionale (SAM) Interfaces
  - Namen entsprechen den Argumenten und Rückgabewerten
    - So können Sie „Integrable“ z.B. durch „DoubleUnaryOperator“ ersetzen
  - Sie müssen das API lesen, wg. der Methodennamen
    - Die Lambdas verwenden den Methodennamen nicht, aber Ihr Code, der die Lambdas verwendet, muss die Methode explizit aufrufen

Einfache und generische Interfaces

- Typen vorgegeben
  - Beispiele (es gibt viel mehr)
    - IntPredicate (int in, boolean out)
    - LongUnaryOperator (long in, long out)
    - DoubleBinaryOperator(two doubles in, double out)
  - Beispiel

```
DoubleBinaryOperator f = (d1, d2) -> Math.cos(d1 + d2);
```

- Generisch
  - Viele der Interfaces (Function<T,R>, Predicate<T>, etc.) haben breites Einsatzspektrum
    - Und default Methoden, z.B. “and”, „or“, „negate”

Interface aus Integrationsbeispiel

```
@FunctionalInterface
public interface Integrable {
   double eval(double x);
}
```

Numerische Integrationsmethode

```
public static double integrate(Integrable function,
                               double x1, double x2,
                               int numSlices){
  if (numSlices < 1) {
    numSlices = 1;
  }
  double delta = (x2 - x1)/numSlices;
  double start = x1 + delta/2;
  double sum = 0;
  for(int i=0; i<numSlices; i++) {
    sum += delta * function.eval(start + delta * i);
  }
  return(sum);
}
```

Method for Testing

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
  for(int i=1; i<7; i++) {
    int numSlices = (int)Math.pow(10, i);
    double result =
      MathUtilities.integrate(function, x1, x2, numSlices);
    System.out.printf("  For numSlices =%,10d
                      result = %,.8f%n",
                      numSlices, result);
  }
}
```

Verwendung Num. Integration

```
MathUtilities.integrationTest(x -> x*x, 10, 100);
MathUtilities.integrationTest(x -> Math.pow(x,3), 50, 500);
MathUtilities.integrationTest(Math::sin, 0, Math.PI);
MathUtilities.integrationTest(Math::exp, 2, 20);
```

Verwendung der Basisbausteine

- Ersetze

```
public static double integrate(Integrable function, …) {
  ... function.eval(...);  ...
}
```

- durch

```
public static double integrate(DoubleUnaryOperator function, …)
{
  ... function.applyAsDouble(...); ...
}
```

- Anschließend können wir Integrable entfernen
  - Denn „DoubleUnaryOperator“ ist ein funktionales Interface, dessen Methode die gleiche Signatur wie „eval“ hat

Entscheidungsregeln

- Geraten Sie in Versuchung, ein funktionales Interface ausschließlich als Ziel für Lambdas zu verwenden:
  - Prüfen Sie, ob stattdessen eines der funktionalen Interfaces aus java.util.function verwendet werden kann
    - DoubleUnaryOperator, IntUnaryOperator, LongUnaryOperator
      - double/int/long in, same type out
    - DoubleBinaryOperator, IntBinaryOperator, LongBinaryOperator
      - Two doubles/ints/longs in, same type out
    - DoublePredicate, IntPredicate, LongPredicate
      - double/int/long in, boolean out
    - DoubleConsumer, IntConsumer, LongConsumer
      - double/int/long in, void return type
    - Generische Interfaces: Function, Predicate, Consumer, etc.
      - Siehe folgende Abschnitte

Generische Bausteine: Predicate

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Predicate: Das Wichtigste

- Vereinfachte Definition

```
public interface Predicate<T> {
  boolean test(T t);
}
```

Vereinfacht, da Predicate einige nicht-abstrakte Methoden hat, die später erläutert werden.

- Idee
  - Baue eine „Funktion” um eine Bedingung zu prüfen
- Nutzen
  - Suchen oder prüfen Sie auf Elemente in einer Collection ohne (mit viel weniger) redundantem Code
- Syntax Beispiel

```
Predicate<Employee> matcher = e -> e.getSalary() > 50_000;
if(matcher.test(someEmployee)) {
  doSomethingWith(someEmployee);
}
```

Beispiel: Findenden passender Einträge in Liste

- Idee
  - Sehr oft haben Sie eine Liste, wollen aber nur eine Teilmenge verarbeiten und den Rest ignorieren
- Ohne Lambdas
  - Gefahr redundanten Codes für verschiedene Arten von Tests
- Mit Lambdas, Schritt 1
  - Verwende Predicate<TypeInOurList> um den Test zu verallgemeinern
- Mit Lambdas, Schritt 2
  - Verwende Predicate<T> um auf verschiedene Arten von Listen zu verallgemeinern
- Mit Lambdas (siehe späteren Abschnitt)
  - Verwende die filter-Methode von Streams um die Vorteile von Verkettung, lazy evaluation und Parallelisierung nutzen zu können

Ohne Predicate: Finde Employee nach Vorname

```
public static Employee findEmployeeByFirstName(
        List<Employee> employees, String firstName) {
            for(Employee e: employees) {
                if(e.getFirstName().equals(firstName)) {
                    return(e);
                }
            }
        return null;
}
```

Ohne Predicate: Finde Employee nach Salary

```
public static Employee findEmployeeBySalary(
        List<Employee> employee
        double salaryCutoff) {
            for(Employee e: employees) {
               if(e.getSalary() >= salaryCutoff) {
                  return(e);
            }
        }
    return null;
}
```

Viel wiederholter Code. Für weitere Suchoptionen geht es entsprechend weiter.

Refactoring 1: Finde ersten Employee, der dem Test genügt

```
public static Employee firstMatchingEmployee(
        List<Employee> candidates,
        Predicate<Employee> matchFunction) {
            for(Employee possibleMatch: candidates) {
                if(matchFunction.test(possibleMatch)) {
                    return(possibleMatch);
                }
            }
         return null; //Achtung, Hierzu später mehr!
}
```

Refactoring 1: Nutzen

- Jetzt
  - Verschiedenen match-Funktionen können einfach und gut lesbar verwendet werden.
    - `firstMatchingEmployee(employees, e -> e.getSalary() > 500_000);`
    - `firstMatchingEmployee(employees, e -> e.getLastName().equals("…"));`
    - `firstMatchingEmployee(employees, e -> e.getId() < 10);`
- Vorher
  - Unhandliches Interface.
    - Ohne Lambdas, hätte man ein Interface instanziieren können. Das würde einigen redundanten Code eliminieren. Ist aber unhandlich.
- Es geht noch besser
  - Der Code hängt noch an der Employee Klasse.

Refactoring 2: Finde erstes Element, das dem Test genügt

```
public static <T> T firstMatch(
        List<T> candidates,
        Predicate<T> matchFunction) {
            for(T possibleMatch: candidates) {
                if(matchFunction.test(possibleMatch)) {
                    return(possibleMatch);
                }
            }
        return null;
}
```

Nun können wir Tests für Elemente einer beliebigen Klasse durchführen

Verwendung von firstMatch

- firstMatchingEmployee Beispiele funktionieren weiterhin
  - `firstMatch(employees, e -> e.getSalary() > 500_000);`
  - `firstMatch(employees, e -> e.getLastName().equals("…"));`
  - `firstMatch(employees, e -> e.getId() < 10);`
- Auch allgemeinerer Code geht nun
  - ```
    Country firstBigCountry =
      firstMatch(countries, c -> c.getPopulation() > 10_000_000);
    ```
  - ```
    Car firstCheapCar =
      firstMatch(cars, c -> c.getPrice() < 15_000);
    ```
  - ```
    Company firstSmallCompany =
      firstMatch(companies, c -> c.numEmployees() <= 50);
    ```
  - ```
    String firstShortString =
      firstMatch(strings, s -> s.length() < 4);
    ```

Test Lookup nach Vorname

```
private static final List<Employee> EMPLOYEES = EmployeeSamples.getSampleEmployees();
private static final String[] FIRST_NAMES = { "Archie", "Amy", "Andy" };

@Test
public void testNames() {
  assertThat(findEmployeeByFirstName(EMPLOYEES, FIRST_NAMES[0]),
             is(notNullValue()));
  for(String firstName: FIRST_NAMES) {
    Employee match1 =
      findEmployeeByFirstName(EMPLOYEES, firstName);
    Employee match2 =
      firstMatchingEmployee(EMPLOYEES, e -> e.getFirstName().equals(firstName));
    Employee match3 =
      firstMatch(EMPLOYEES, e -> e.getFirstName().equals(firstName));
    assertThat(match1, allOf(equalTo(match2), equalTo(match3)));
  }
}
```

Testziele:

- Beide Versionen liefern das gleiche Ergebnis.

Test suche nach Gehalt (Salary)

```
private static final List<Employee> EMPLOYEES = EmployeeSamples.getSampleEmployees();
private static final int[] SALARY_CUTOFFS = { 200_000, 300_000, 400_000 };

@Test
public void testSalaries() {
  assertThat(findEmployeeBySalary(EMPLOYEES, SALARY_CUTOFFS[0]),
             is(notNullValue()));
  for(int cutoff: SALARY_CUTOFFS) {
    Employee match1 =
      findEmployeeBySalary(EMPLOYEES, cutoff);
    Employee match2 =
      firstMatchingEmployee(EMPLOYEES, e -> e.getSalary() >= cutoff);
    Employee match3 =
      firstMatch(EMPLOYEES, e -> e.getSalary() >= cutoff);
    assertThat(match1, allOf(equalTo(match2), equalTo(match3)));
  }
}
```

Definition of Predicate Revisited

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Die Definition ist vereinfacht, da Predicate default und static Methoden hat.
Diese habe wir hier (noch) nicht benötigt.

Lambda Grundprinzipien Revisited

- Interfaces sind weitgehend unverändert
  - Predicate ist in Java 8 genau wie es früher gewesen wäre, bis auf @FunctionalInterface (optional)
    - Um Fehler zur Compile-Zeit zu finden
    - Macht die Designabsicht explizit: Entwickler sollen Lambdas verwenden
- Code, der Interfaces verwendet ist ansonsten unverändert
  - Die Definition von firstMatch ist genau wie vor Java 8. Der Autor von firstMatch wissen, das die Methode „test“ heißt.
- Code, der Methoden aufruft, die ein funktionales Interfaces erwarten, kann nun Lambdas verwenden
  - `firstMatch(employees, e -> e.getSalary() > 500_000);`

Generische Bausteine: Function

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Function: Das Wichtigste

- Vereinfachte Definition

```
public interface Function<T,R> {
  R apply(T t);
}
```

- Idee
  - Sie bauen eine „Function”, die ein T erhält und ein R liefert
    - BiFunction ist ähnlich, bekommt aber zwei Argumente
- Nutzen
  - Sie können eine Collection von Objekten sehr einfach bearbeiten
- Syntax Beispiel

```
Function<Employee, Double> raise = e -> e.getSalary() * 1.1;
for(Employee employee: employees) {
  employee.setSalary(raise.apply(employee));
}
```

Beispiel 1: Refactoring des StringUtility Codes

- Vorgeschichte
  - StringFunction Interface und transform Methode Methodenrreferenzen zu illustrieren.
- Refactoring 1
  - Ersetze StringFunction durch Function<String,String>
  - Auch die transform Methode muss angepasst werden. Prinzip: Bis auf den tatsächlichen Methodennamen ändert sich nichts.
- Refactoring 2
  - Verwende Function<T,R> statt Function<String,String>
  - Verallgemeinere transform auf ein Argument T und Rückgabe R

Vorher: Transformation mit StringFunction

- Unser altes Interface

```
@FunctionalInterface
public interface StringFunction {
  String applyFunction(String s);
}
```

- Unser alte Methode

```
public static String transform(String s, StringFunction f) {
  return(f.applyFunction(s));
}
```

- Verwendungsbeispiel

```
String result = Utils.transform(someString,
String::toUpperCase);
```

Refactoring 1: Verwende Function

- Eigenes Interface
  - Keines!
- Eigene Methode

```
public static String transform(String s,
Function<String,String> f) {
  return f.apply(s);
}
```

- Verwendungsbeispiel (unverändert)

```
String result = Utils.transform(someString,
String::toUpperCase);
```

Refactorin 2: Typen verallgemeinern

- Eigenes Interface
  - Keines
- Eigene Methode

```
public static <T,R> R transform(T value, Function<T,R> f) {
  return(f.apply(value));
}
```

- Verwendungsbeispiel (verallgemeinert)

```
String result = Utils.transform(someString,
String::toUpperCase);
List<String> words = Arrays.asList("hi", "bye");
int size = Utils.transform(words, List::size);
```

Beispiel 2: Summe allgemeiner Eigenschaft

- Idee
  - Oft hat man eine Liste (Employees) und addiere etwas (Salary)
  - Viele weitere Beispiele
- Java vor Version 8
  - Gefahr wiederholten Codes
- Java 8
  - Verwende „Function“ um die Transformationsmethode (salary, population, price) zu verallgemeinern

Ohne „Function“: Gehaltssumme

```
  public static int salarySum(List<Employee> employees)
{
    int sum = 0;
    for(Employee employee: employees) {
      sum += employee.getSalary();
    }
    return(sum);
  }
```

Ohne „Function“: Gesamtpopulation

```
  public static int populationSum(List<Country>
countries) {
    int sum = 0;
    for(Country country: countries) {
      sum += country.getPopulation();
    }
    return(sum);
  }
```

Mit „Function“: Summe numerischer Eigenschaft

```
public static <T> int mapSum(
                    List<T> entries,
                    Function<T, Integer> mapper) {
  int sum = 0;
  for(T entry: entries) {
    sum += mapper.apply(entry);
  }
  return(sum);
}
```

Ergebnisse

- Sie können „salarySum“ reproduzieren
  - `int numEmployees = mapSum(employees, Employee::getSalary);`
- Sie können andere Summen bilden:
  - `int totalWeight = mapSum(packages, Package::getWeight);`
  - `int totalFleetPrice = mapSum(cars, Car::getStickerPrice);`
  - `int regionPopulation = mapSum(countries, Country::getPopulation);`
  - `int regionElderlyPopulation = mapSum(listOfCountries, c -> c.getPopulation() - c.getPopulationUnderSixty());`
  - `int sumOfNumbers = mapSum(listOfIntegers, Function.identity());`

Weiter generische Bausteine

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

„BinaryOperator“: Das Wichtigste

- Vereinfachte Definition

```
public interface BinaryOperator<T> {
  T apply(T t1, T t2);
}
```

- Idee
  - Bau dir eine „Function”, die zwei T’s bekommt und ein T liefert
    - Ist eine Spezialisierung von „BiFunction<T,U,R>“.
- Nutzen
  - Siehe „Function“. Da alle Werte den gleichen Typ haben, sehr nützlich für „reduce”-Operationen, die Werte einer Collection zusammenfassen.
- Beispiel für Syntax

```
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;
      // Obiges lambda kann durch Integer::sum ersetzt werden
int sum = adder.apply(num1, num2);
```

„BinaryOperator“: Anwendungen

- Flexiblere Methode mapSum
  - Statt
    - `mapSum(List<T> entries, Function<T, Integer> mapper)`
  - Verallgemeinere weiter und übergebe Zusammenfassungs- (reduce) Operator (hard codiert als „+” in mapSum)
    - `mapReduce(List<T> entries, Function<T, R> mapper, BinaryOperator<R> combiner)`
- Hypothetische Beispiele
  - `int payroll = mapReduce(employees, Employee::getSalary, Integer::sum);`
  - `double lowestPrice = mapReduce(cars, Car::getPrice, Math::min);`
- Problem:
  - Was macht man, wenn es keine Einträge gibt? mapSum liefert 0, aber was soll mapReduce liefern? Siehe Streams und die Klasse Optional.

Consumer: Das Wichtigste

- Vereinfachte Definition

```
public interface Consumer<T> {
  void accept(T t);
}
```

- Idee
  - Baue eine „Function" die ein T bekommt und dies verändert (keine Rückgabe, void)
- Nutzen
  - Sie können eine Funktion (Methode) auf eine Collection von Objekten anwenden, ohne weiteren Code schreiben zu müssen
- Syntax Beispiel

```
Consumer<Employee> raise = e -> e.setSalary(e.getSalary() * 1.1);
for(Employee employee: employees) {
  raise.accept(employee);
}
```

„Consumer“: Anwendung

- Die Methode „forEach“ von Stream verwendet Consumer
  - `employees.forEach(e -> e.setSalary(e.getSalary()*1.1));`
  - `values.forEach(System.out::println);`
  - `textFields.forEach(field -> field.setText(""));`

- Weitere Details
  - Siehe Streams

Supplier: Das Wichtigste

- Vereinfachte Definition

```
public interface Supplier<T> {
  T get();
}
```

- Idee
  - Ermöglicht Ihnen eine “Funktion” ohne Parameter, die ein T liefert. Kann dies irgendwie leisten: Aufruf von „new”, Verwendung existierenden Objekts, etc.
- Nutzen
  - Einfacher Austausch von Objekterzeugungsmethoden. Auch nützlich zum umschalten von Test zu Produktion, etc.
- Syntax Beispiel

```
Supplier<Employee> maker1 = Employee::new;
Supplier<Employee> maker2 = () -> randomEmployee();
Employee e1 = maker1.get();
Employee e2 = maker2.get();
```

Using Supplier to Randomly Make Different Types of Person

```
private final static Supplier[] peopleGenerators =
  { Person::new, Writer::new, Artist::new, Consultant::new,
    EmployeeSamples::randomEmployee,
    () -> { Writer w = new Writer();
            w.setFirstName("Ernest");
            w.setLastName("Hemingway");
            w.setBookType(Writer.BookType.FICTION);
            return(w); }
  };

public static Person randomPerson() {
  Supplier<Person> generator =
    RandomUtils.randomElement(peopleGenerators);
  return(generator.get());
}
```

When randomPerson is called, it first randomly chooses one of the people generators, then uses that Supplier to build an instance of a Person or subclass of Person.

Helper Method: randomElement

```
public class RandomUtils {
  private static Random r = new Random();

  public static int randomInt(int range) {
    return(r.nextInt(range));
  }

  public static int randomIndex(Object[] array) {
    return(randomInt(array.length));
  }

  public static <T> T randomElement(T[] array) {
    return(array[randomIndex(array)]);
  }
}
```

Using randomPerson

- Test code

```
System.out.printf("%nSupplier Examples%n");
for(int i=0; i<10; i++) {
  System.out.printf("Random person: %s.%n", EmployeeUtils.randomPerson());
}
```

- Results (one of many possible outcomes)

```
Supplier Examples
Random person: Andrea Carson (Consultant).
Random person: Desiree Designer [Employee#14 $212,000].
Random person: Andrea Evans (Artist).
Random person: Devon Developer [Employee#11 $175,000].
Random person: Tammy Tester [Employee#19 $166,777].
Random person: David Carson (Writer).
Random person: Andrea Anderson (Person).
Random person: Andrea Bradley (Writer).
Random person: Frank Evans (Artist).
Random person: Erin Anderson (Writer).
```

Wrap-Up

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Summary

- Type-specific building blocks
  - *Blah*UnaryOperator, *Blah*BinaryOperator, *Blah*Predicate, *Blah*Consumer
- Generic building blocks
  - Predicate

```
Predicate<Employee> matcher = e -> e.getSalary() > 50000;
if(matchFunction.test(someEmployee)) { doSomethingWith(someEmployee); }
```

  - Function

```
Function<Employee, Double> raise = e -> e.getSalary() + 1000;
for(Employee employee: employees) { employee.setSalary(raise.apply(employee)); }
```

  - BinaryOperator

```
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;
int sum = adder.apply(num1, num2);
```

  - Consumer

```
Consumer<Employee> raise = e -> e.setSalary(e.getSalary() * 1.1);
for(Employee employee: employees) { raise.accept(employee); }
```

Questions?

More info:

http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial

http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial

http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization

http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training

Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

Slides © 2016 Marty Hall,hall@coreservlets.com,
Übersetzung: Bernd Kahlbrandt, bernd.kahlbrandt@haw-hamburg.de

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.