Fehlerbehandlung

Motivation

Blue Screen

© SkillUp - Shutterstock

Motivation

Jedes komplexere Programm muss mit gewissen Fehlersituationen umgehen können

- Mehrarbeit und zusätzliche Zeilen Sourcecode, um auf unerwartete Situationen reagieren oder Fehler abfangen zu können.
- Moderne Programmiersprachen haben ein Exception Handling zur Verarbeitung (Erkennung und Behandlung oder Propagation) von Fehlerzuständen

Motivation

Ok

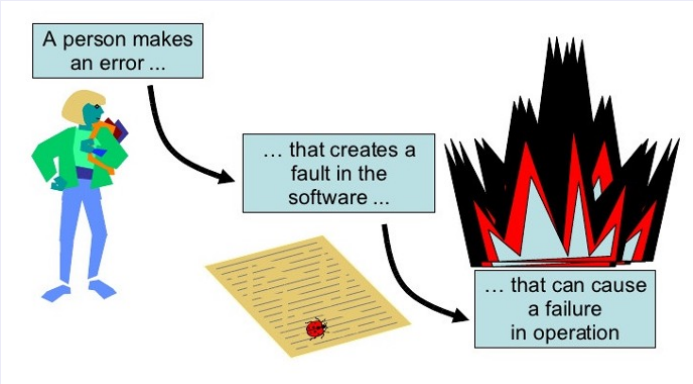
Eine fehlende oder inkorrekte Fehlerbehandlung macht sich im User Interface z.B. dadurch bemerkbar, dass der Wartecursor nicht wieder auf den Standardcursor gesetzt wird.

Gefahr

In der Business-Logik oder der Datenzugriffsschicht kommt es u.a. zu nicht freigegebenen Ressourcen.

Motivation - Fehlerpropagierung

Fault–Error–Failure



Einstieg in die Java Fehlerbehandlung

In Java

- Durch Exceptions ungewöhnliche Situationen behandeln
- Fehlersituationen durch spezielle Rückgabewerte beschreiben
- Mit Assertions gewisse Zustände im Programm zu prüfen.

Beispiel

Die Behandlung von Fehlern besitzt zahlreiche Facetten und ein allgemein richtiges Verhalten gibt es nicht.

```
public class ExceptionExample1 {
  ...
  public Object findById(final long objectId) {
    // Ungueltiger Parameterwert => Exception
    if (objectId < 0)
      throw new IllegalArgumentException("objectId must not be negative");
    if (DbHelper.isValid(objectId)) {
      final Object obj = DbHelper.findById(objectId);
      // Zustandspruefung mit Assertion
      assert (obj != null);
      return obj;
    }
    // Spezieller Rueckgabewert fuer keinen Treffer
    return null;
  }
}
```

Typische Exceptions

In Java

- IllegalArgumentException – Mit einer IllegalArgumentException kön-nen falsche Belegungen von Parametern ausgedrückt werden
- NullPointerException – Sind Eingabewerte null, so kann man darauf mit einer NullPointerException reagieren
- IllegalStateException – Sind benötigte Daten nicht korrekt initialisiert, so kann dies über eine IllegalStateException kommuniziert werden
- UnsupportedOperationException – Auf eine fehlende Implementierung kann mittels einer UnsupportedOperationException hingewiesen werden.

Lösung

```
try{
  // Potenziell Exception ausloesend
  final Object obj = findById(objectId);
  // ...
}
catch (final IllegalArgumentException ex
    )
{
  // Fehlerbehandlung
  errorCounter ++;
  showMessageBox("Ungueltige objectId --
       darf nicht negativ sein");
}
```

Was passiert

- Tritt bei der Abarbeitung der Anweisungen des try-Blocks ein Fehler auf, der eine Exception auslöst, so stoppt die Verarbeitung der Anweisungen im try-Block
- Es wird direkt ein auf die Exception ausgelegter catch-Block angesprungen

Beispiel

Keine geeignete Reaktion auf Fehler

```
catch (final IllegalArgumentException ex)
{
  ex.printStackTrace();
}
```

Spezifische Fehlerbehandlung und abgeleitete Exceptions

Oftmals können durch die Anweisungen in einem try-Block verschiedene Arten von Exceptions ausgelöst werden. Für diesen Fall kann man zur Behandlung mehrere catch-Blöcke wie folgt dafür definieren

```
try {
  final File file = new File(...);
  //...
}
catch (final IOException ioe) {
  // Fehlerbehandlung
  handleIOException(ioe)
}
catch (final ParseException pex) {
  // Fehlerbehandlung
  handleParseException(pex)
}
```

Vorsicht: Es kommt auf die Reihenfolge an!

```
try {
  final File file = new File(...);
  //...
}
catch (final FileNotFoundException fnfe)
    {
  handleFileNotFoundException(fnfe)
}
catch (final IOException ioe) {
  handleIOException(ioe)
}
```

Achtung

- Reihenfolge der in den catch-Blöcken gefangenen Exceptions muss entgegengesetzt zur Ableitungshierarchie erfolgen
- FileNotFoundException ist Spezialisierung von IOException

Abschließende Aktionen

finally-Block

```
try {
  // Hier koennen Exceptions auftreten
}
catch (Exception ex) {
  // Fehlerbehanldung
}
finally {
  // Anweisungen werden immer dann ausgefuehrt, wenn der finally-Block vorhanden ist -
      also egal, ob ein Fehler aufgetreten ist oder nicht
}
```

Automatische Propagation

- Existiert kein catch-Block, so muss eine möglicherweise auftretende Exception in der Signatur der Methode angegeben werden.
- Falls bei der Ausführung des Programms dann bei der Fehlerbehandlung kein catch-Block passt, wird die Exception automatisch an den Aufrufer weiter propagiert.

Bad Smell: „Anstands“ -Null-Prüfungen

Problematisch: Auf den speziellen Zustand (state == null wir nicht hingewiesen)

```
public static void updateSystemState()
{
  final SystemState state = calculateSystemState();
  if (state != null)
  {
    systemStateMap.put(KEY_SYSTEM, state);
  }
}
```

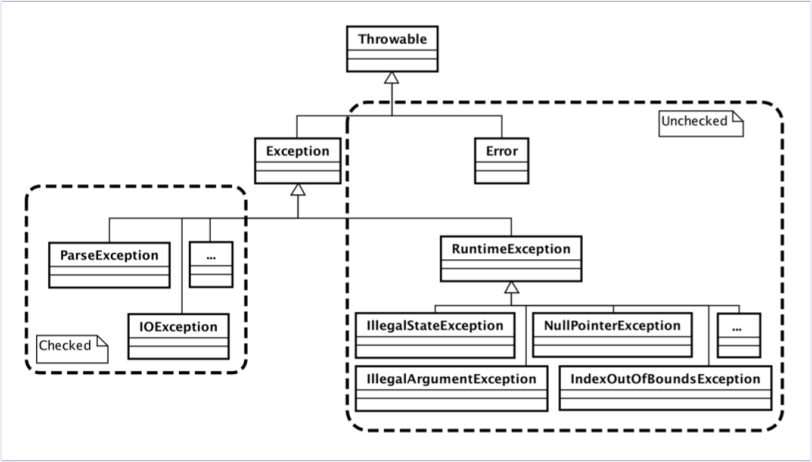
Bad Smell: „Delayed“ Exeption

Problematisch: Mögliche Fehleingabe bleibt sehr lange unerkannt, erst wenn später ein Aufruf von getSystemState() erfolgt, wird die ungültige Eingabe sichtbar

```
public static void setSystemState(final SystemState state)
{
  systemStateMap.put. (KEY_SYSTEM, state);
}
public static SystemState getSystemState()
{
  final SystemState state = systemStateMap.get (KEY_SYSTEM);
  if (state == null)
    throw new IllegalStateException ("No entry for systen state");
  return state;
}
```

Checked Exceptions und Unchecked Exceptions

Exception Hierarchie - Ausschnitt



Exceptions

Checked Exception

- sind Bestandteil des „Vertrags“ zwischen Aufrufer und Bereitsteller einer Methode
- zeigen mögliche, durch Aufrufer zu erwartende Fehlersituationen
- müssen in der Methodensignatur mit dem Schlüsselwort throws angegeben werden
- Müssen durch catch gefangen werden, ansonsten erfolgt automatische Propagation an weitere aufrufende Methoden

Unchecked Exceptions

- vom Basistyp Runtime-Exception und nicht zwingend notwendig zu behandeln
- aber möglich mit einem catch-Block bearbeiten
- Weil aber Unchecked Exceptions normalerweise schwerwiegende Programmierprobleme oder unerwartete Situationen ausdrücken, ist das Verarbeiten mit einem catch-Block eher ungewöhnlich

Regel

Checked Exception und Unchecked Exception

Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so kann eine Checked Exception bevorzugt werden. Ist nicht davon auszugehen, dass ein Aufrufer die Fehlersituation korrigieren soll, oder ist ein Aufrufer dazu höchstwahrscheinlich nicht in der Lage, so ist eine Unchecked Exception die richtige Wahl.

Bad Smell: Verletzung des DRY-Prinzips

Problematisch

```
public static void main (String[] args) {
  try {
    exceptionThrowingMethod();
  }
  catch (final RemoteException ex) {
    reportException(ex);
  }
  catch (final FileNotFoundException ex) {
    reportException(ex);
  }
}
private static void exceptionThrowingMethod() throws RemoteException,
    FileNoteFoundException
{ ... }
```

Seit Java 7

Multi Catch

```
public static void main (String[] args)
{
  try
  {
    exceptionThrowingMethod();
  }
  catch (final RemoteException | FileNotFoundExceptionex)
  {
    reportException(ex);
  }
}
```

Achtung

Multi Catch

```
public class TryCatch {
  public static void main (String[] args
      ) {
    new TryCatch().m();
  }
  public void m() {
    try {
      System.out.println("m(): try");
      int i = 1 / 0;
    } catch (RuntimeException |
        ArithmeticException e) {
      System.out.println("m(): catch
          known Err");
    } catch ()Exception e) {
      System.out.println("m(): catch
          Exception");
}}}
```

Ausgabe

Unresolved compilation problem: The exception ArithmeticException is already caught by the alternative RuntimeException

Final Rethrow - Catch bleibt übersichtlich

- Final Rethrow ermöglicht es, mehrere Exception-Typen aus einem einzigen catch-Block weiter zu propagieren.
- somit ist es möglich, dass verschiedene Typen von Exceptions per catch (Exception ex) gefangen werden

```
private void performCalculation(final String fileName) throws IOException, RemoteException
private void finalRethrowV2(final String fileName) throws IOException, RemoteException {
  try {
    performCalculation(fileName);
  }
  // Das final verhindert fuer erste Versionen von JDK 7 einen Compile-Error
  catch (final Exception ex) {
    log.error("exception occurred", ex);
    throw ex; // Compile-Error vor JDK 7
  }
}
```

Eigene Exception

Eigene Exception-Klasse

```
public class MeineException extends
    Exception {
  public MeineException (String message)
      {
    super(message)
  }
  public MeineException(String message,
      Throwable throwable) {
    super(message, throwable)
}}
```

Regel

- Ableitung eigener Exceptions von Oberklasse Exception
- Definition eines leeren Konstruktors und eines Konstruktors mit String-Parameter, um Exceptions beim Werfen erzeugen zu können

Frage

Wieso muss ein Konstruktor deklariert werden?

Automatic Resource Management (ARM)

- Vor Java 1.7 musste man Ressourcen manuell über den finally Block freigeben.
- Ab Java 1.7 gibt es die Möglichkeit, einen try Block mit Ressourcen zu erstellen welcher sich automatisch um die Freigabe nicht mehr benötigter Ressourcen kümmert

Ohne ARM

```
public static String readFirstLine(final String path) {
  BufferedReader br = null;
  try {
    br = new BufferedReader(new FileReader(path));
    return br.readLine();
  } catch (final IOException ex) {
    // handle or rethrow
  } finally {
    try {
      if (br != null) {
        br.close();
      }
    } catch (final IOException ioe) {
      // ignore
  }}
  return "";}
```

Mit ARM

```
public static String readFirstLine(final String path) {
  try (final FileReader fr = new FileReader(path);
  final BufferedReader br = new BufferedReader(fr)) {
    return br.readLine();
  } catch (final IOException ex) {
    // handle or rethrow
  }
  return "";
}
```

Für jede in den runden Klammern des try-Blocks angegebene Variable wird beim Verlassen des try-Blocks automatisch die Methode close() aufgerufen. Voraussetzung dafür ist, dass die dort definierten Referenzvariablen das Interface java.lang.Auto-Closeable erfüllen.

Assertions

- Ziel: erwartete Zustände abzusichern.
- Allgemein: Zur Formulierung einer solchen Zusicherung wird das Schlüsselwort assert sowie eine boolesche Bedingung angegeben.
- $\Rightarrow$ Wird diese zu false evaluiert, so wird ein java.lang.AssertionError ausgelöst
- $\Rightarrow$ Verarbeitung von Assertions ist für die JVM standardmäßig deaktiviert ist und explizit aktiviert werden muss (JVM-Parameter -ea bzw. –enableassertions)

Assert am Beispiel

```
public class AssertTest {
  public static void main(final String[] args) {
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");
    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1) {
      // Versionen auslesen
      final String majorVersion = tokenizer.nextToken().trim();
      final String minorVersion = tokenizer.nextToken().trim();
      // Sicherstellen, dass Tokens einen Wert enthalten
      assert !majorVersion.isEmpty();
      assert !minorVersion.isEmpty();
      System.out.println("Major: '" + majorVersion + "'");
      System.out.println("Minor: '" + minorVersion + "'");
      System.out.println("#Tokens: '" + tokenCount + "'");
    } else {
      System.err.println("Unexpected version number format => no '.' found");
}}}
```

... aussagekräftiger

```
public class AssertTest2 {
  public static void main(final String[] args) {
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");
    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1) {
      final String majorVersion = tokenizer.nextToken().trim();
      final String minorVersion = tokenizer.nextToken().trim();
      assert !majorVersion.isEmpty() : "Major-Version must not be empty";
      assert !minorVersion.isEmpty() : buildWarnMessage("Minor-Version");
      System.out.println("Major: '" + majorVersion + "'");
      System.out.println("Minor: '" + minorVersion + "'");
      System.out.println("#Tokens: '" + tokenCount + "'");
    } else {
      System.err.println("Unexpected version number format => no '.' found");
  }}
  private static String buildWarnMessage(final String versionName) {
    return versionName + " must not be empty!";
}}
```

Tipps zum Einsatz von Assertions

On - Off

- Problem: Es ist möglich, Assertions zur Laufzeit an- und abzuschalten, daher kann man sie als ein „löchriges Sicherheitsnetz“ auffassen.
- Da Assertions jederzeit ohne unsere Kontrolle ein- bzw. ausgeschaltet werden können, stellen sie kein geeignetes Mittel dar, Eingabeparameter öffentlicher Methoden zu prüfen. Zur Prüfung von Werten innerhalb privater Methoden ist der Einsatz von Assertions vertretbar, da hier der eigene Objektzustand eigentlich immer gesichert sein sollte, indem bereits zuvor durch Parameterprüfungen in öffentlichen Methoden Fehleingaben verhindert wurden.

Tipps zum Einsatz von Assertions

Tests

Assertions für Situationen einzusetzen, die „niemals“ auftreten können bzw. sollen, z.B. „default“ in switch-case

Assertions als semantischer Kommentar

Mit Assertions lassen sich Bedingungen in den Sourcecode einbringen, die wie ein Kommentar zu lesen sind, aber zusätzlich die Validierung gewisser Zusicherungen erlauben. Dadurch können Annahmen klarer als lediglich mit einem reinen Kommentar formuliert werden.

Assertions und Seiteneffekte

Die Ausführung von Assertions sollte keine Änderung an Variablen oder Attributen vornehmen, da diese je nach Aktivierung abgearbeitet werden oder eben nicht. Es ist zudem empfehlenswert, dass die in Assertions aufgerufenen Methoden keine Seiteneffekte verursachen.

Lambda

Lambda–Ausdrücke

Intention: häufig vorkommenden Java-Code zu vereinfachen

- Lambda-Ausdrücke in Java sind quasi Methoden ohne Namen.
- Sie bestehen aus einer Liste von formalen Parametern, einem Pfeil $\rightarrow$ und einem Funktionsrumpf
- Im Gegensatz zu Methoden werden der Rückgabetyp und Exceptions nicht spezifiziert, sondern vom Compiler inferiert.
- Die größten praktischen Auswirkungen haben Lambda-Ausdrücke auf die Collection-Klassen und -Schnittstellen

Nomenklatur

Lambda-Ausdrücke werden auch als anonyme Methoden oder als Closures bezeichnet. Closures sind Methoden, die später auf Variablen oder Objekte der Code-Umgebung zugreifen, in der sie definiert wurden.

Beispiel

```
public class HelloLambda {
  public static void main(String[] args) {
    String[] ar = { "Hello ", "Lambda ", "World!" };
    Arrays.asList(ar).forEach(s -> System.out.print(s));
  }
}
```

Hinter den Kulissen

forEach erwartet als Parameter ein Consumer-Objekt

```
void Iterable.forEach(Consumer<? super T> action) { ...}
```

Die Consumer-Schnittstelle ist wiederum so definiert

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

$\Rightarrow$ Lambda-Ausdruck stellt Code für die accept-Methode zur Verfügung

Alternative

Anonyme Klasse

```
Arrays.asList(ar).forEach(
  new Consumer<String>() {
    public void accept(String t) {
      System.out.print(t);
    }
} );
```

- Lambda-Ausdrücke besser lesbar
- Mehr Arbeit für den Compiler

Welche Informationen sind das?

- Aufgrund der Definition der forEach-Methode weiß der Compiler, dass die Methode ein Consumer-Objekt erwartet.
- Die Consumer-Schnittstelle ist eine funktionale Schnittstelle, d. h., sie enthält genau eine Methode ohne Defaultimplementierung. Das macht dem Compiler klar, dass der Lambda-Ausdruck den Code für die accept-Methode liefert.
- Der Compiler erkennt sogar den Datentyp für den Parameter s des Lambda-Ausdrucks s $\rightarrow$ ... Es muss sich um String-Objekte handeln, weil forEach auf eine Liste mit String-Elementen angewendet wird.

Die Syntax von Lambda-Ausdrücken

Beispiele

```
// Lambda-Ausdruck ohne Parameter
() -> 7;
() -> "Ergebnis";

// Lambda-Ausdruck mit einem Parameter
(int i) -> i*i;
(i) -> i*i;
i -> i*i;

// Lambda-Ausdruck mit mehreren Parametern
(int i, String s) -> s.substring(i, i+1);
(i, s) -> s.substring(i, i+1);
```

Beispiel

Beispiele

```
@FunctionalInterface
interface Square {
  int calculate(int q, int r);
}
public class LambdaBeispiel {
  public static void main(String[] args) {
    Square i = (int q,int r) -> q*r;
    System.out.println(i.calculate(7,1));
  }
}
```

Wichtig

Parameter

Die Namen der Parameter einer anonymen Funktion dürfen nicht mit jenen bereits definierter Variablen übereinstimmen!

Klammerung

```
FilenameFilter (f, s) -> {String lower = s.toLowerCase(); return lower.endsWith(".pdf");};
```

- Wenn der Lambda-Ausdruck aus mehreren Java-Kommandos besteht, muss er wie üblich geklammert werden
- Ergebnisse müssen dann wie bei Methoden mit return zurückgegeben werden.
- Es sind aber auch Lambda-Ausdrücke ohne Ergebnis zulässig

Anwendungsgebiete

Funktionale Schnittstellen

- Lambda-Ausdrücke können nur zur Implementierung von funktionellen Schnittstellen verwendet werden. (z. B. Runnable, Callable, ActionListener und Comparator).
- Lambda-Ausdrücke sind ungeeignet, um Schnittstellen mit mehreren Methoden (default-Methoden werden nicht gezählt!) oder abstrakte Klassen zu implementieren.

this und super

- Die Lambda-Funktion kann ebenso wie eine anonyme Klasse direkt auf Variablen zugreifen, die in derselben Code-Ebene zugänglich sind
- Achtung: Bei this und super verhält sich Code in Lambda-Ausdrücken allerdings anders als Code in anonymen Klassen
- $\Rightarrow$ In Lambda-Ausdrücken haben this und super dagegen dieselbe Bedeutung wie im Code außerhalb
- $\Rightarrow$ this bezieht sich also auf Elemente der Klasse, in der der Lambda-Ausdruck definiert wird, super auf deren Basisklasse

Referenzen auf Methoden

Allgemein: Mit Klasse::statischeMethode bzw. objekt::Methode können Referenzen auf Methoden übergeben werden.

1. Fall: Statische Methoden

```
List<Integer> data = Arrays.asList(7, 2, 5, 4);
Optional <Integer > minimum = data.stream().reduce(Math::min);
System.out.println(minimum);
```

Mit Lambda-Ausdruck

```
minimum = data.stream().reduce( (i1, i2) -> Math.min(i1, i2) );
```

Referenzen auf Methoden

Allgemein: Mit Klasse::statischeMethode bzw. objekt::Methode können Referenzen auf Methoden übergeben werden.

2. Fall: Nichtstatische Methode mit Objekt

```
List <Double > ld = new ArrayList < >();
ld.add(1.0);
ld.add(2.2);
ld.add(0.3);
ld.forEach(System.out::println);
```

Mit Lambda-Ausdruck

```
ld.forEach(d -> System.out.println(d));
```

Referenzen auf Methoden

Allgemein: Mit Klasse::statischeMethode bzw. objekt::Methode können Referenzen auf Methoden übergeben werden.

3. Fall: Nichtstatische Methode mit Klasse/Typ

```
public class Kontakt {
  public String vorname , nachname , telnr; ...
  public void prettyPrint() {
    System.out.format("%s %s (%s)\n",vorname , nachname , telnr);
  }
  Kontakt[] kontakte = new Kontakt[4];
  kontakte[0] = new Kontakt("Martin", "Muueller", "123");
  kontakte[1] = ... Arrays.asList(kontakte).forEach(Kontakt::prettyPrint);
```

Mit Lambda-Ausdruck

```
Arrays.asList(kontakte).forEach(k -> k.prettyPrint());
```

Referenzen auf Methoden

Allgemein: Mit Klasse::statischeMethode bzw. objekt::Methode können Referenzen auf Methoden übergeben werden.

Spezialfall: Referenz auf einen Konstruktor

- Klasse::new vor bzw. bei generischen Klassen Klasse::<Typ>new
- $\Rightarrow$ Konstruktor ist keine Methode

Beispiel für Referenzen auf Methoden

```
public class DoubleTriplet {
  public double a, b, c;
  public DoubleTriplet(double data1, double data2, double data3) {
    a = data1;
    b = data2;
    c = data3;
  }
  public void printout() {
    System.out.format("[%f, %f, %f]\n", a, b, c);
  }
  public void printoutDetail() {
    System.out.format("Triplet a=%f, b=%f, c=%f, len=%f\n", a, b, c, Math.sqrt(a * a + b *
        b + c * c));
  }
}
```

Beispiel für Referenzen auf Methoden

```
public class TestDoubleTriplet {
  public static void main(String[] args) {
    var lst = new ArrayList<DoubleTriplet>();
    lst.add(new DoubleTriplet(1, 1, 1.2));
    lst.add(new DoubleTriplet(1, 2, 3));
    lst.add(new DoubleTriplet(2, 2, 4));

    lst.forEach(d -> d.printout());
    lst.forEach(DoubleTriplet::printout);
  }
}
```

Entstehungsgeschichte Defaultmethoden

Defaultmethoden werden oft in einem Atemzug mit Lambda-Ausdrücken genannt, obwohl sie mit diesen unmittelbar gar nichts zu tun haben!

Iterable, wurde in Java 8 um die neue Methode forEach erweitert

$\Rightarrow$ Erwartet Lambda-Ausdruck als Parameter

Problem: Alle „älteren“ Klassen, die Iterable implementieren, müsste noch angepasst werden

$\Rightarrow$Es gäbe praktisch kein älteres Programm, das durch das Update auf Java 8 oder eine neuere Version nicht grundlegend geändert werden müsste.

Lösung

Defaultmethode

bei der Definition von Schnittstellen einzelne Methoden mit dem Schlüsselwort default zu deklarieren und mit Code zu versehen. Implementierung kann übernommen oder überschrieben werden.

Beispiel Iterable

```
public interface Iterable <T> ... {
  ...
  default void forEach(Consumer<? super T> action){
    Iterables.forEach(this , action);
  }
}
```

⇒ Sämtliche Klassen, die Iterable implementieren, funktionieren weiterhin.

Problem: Mehrfachvererbung

Die Mehrfachvererbung von Typen ist in Java seit jeher möglich

Ableitung von einer Klasse und Implementierung zusätzlicher Interfaces

Durch Default-Methoden wird eine Mehrfachvererbung von Zustand nicht möglich und ist unkritisch.

Default-Methoden ermöglichen Mehrfachvererbung von Verhalten

Default-Methoden werde als virtuelle Methoden behandelt

Welche Ausgabe?

```
interface A {
  default void m(int i, int j) {
    int result = calculate(i, j);
    System.out.println("A.m(i=" + i + ",j=" + j + ")
          := " + result);
  }
  default int calculate(int i, int j) {
    System.out.println("A.calculate aufgerufen");
    return i + j;
  }
}
interface B extends A {
  default int calculate(int i, int j) {
    System.out.println("B.calculate aufgerufen");
    return i * j;
  }
  default void m(int i, int j) {
    int result = calculate(i, j);
    System.out.println("B.m(i=" + i + ",j=" + j + ")
          := " + result);
  }
}
```

```
interface C extends A {
  default void m(int i, int j) {
    int result = calculate(i, j);
    System.out.println("C.m(i=" + i + ",j=" + j + ")
          := " + result);
  }
}
public class D implements B, C {
  public static void main(String[] args) {
    D d = new D();
    d.m(3, 4);
  }
  @Override
  public void m(int i, int j) {
    C.super.m(i, j);
  }

}
```

Wichtigsten Lambda-Schnittstellen im Paket „java.util.function“

java.util.function enthallt eine ganze Sammlung von Schnittstellen

- Collection.forEach(Consumer)
- Collection.removeIf(Predicate)
- List.replaceAll(Function)
- Stream.anyMatch(Predicate)

Predicate-Schnittstelle

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

Schnittstelle	Aufgabe
`Predicate<T>`	Überprüft, ob ein Objekt vom Typ `T` ein Kriteriun erfüllt.
`Supplier<T>`	Liefert Objekte vom Typ `T` (z. B. für `get`-Methoden).
`Consumer<T>`	Verarbeitet ein Objekt vom Typ `T`, gibt kein Ergebnis zurück.
`BiConsumer<T, U>`	Verarbeitet zwei Objekte vom Typ `T` und `U`, gibt kein Ergebnis zurück.
`Function<T, R>`	Verarbeitet ein Objekt vom Typ `T` und liefert als Ergebnis ein Objekt vom Typ `R` zurück.
`BiFunction<T, U, R>`	Verarbeitet zwei Objekte vom Typ `T` und `U` und liefert als Ergebnis ein Objekt vom Typ `R` zurück.
`UnaryOperator<T>`	Entspricht `Function<T, T>`, d. h., die zu verarbeitenden Daten und die Ergebnisse weisen denselben Typ auf.
`BinaryOperator<T>`	Entspricht `BiFunction<T, T, T>`, d. h., die zu verarbeitenden Daten und die Ergebnisse weisen denselben Typ auf.

Welche Ausgabe?

```
  public static void main(String[] args) {
    Integer[] zahlen = new Integer[100];
    for (int i = 0; i < 100; i++)
    zahlen[i] = i + 1;
    printNumbers(zahlen, z -> z < 10);
    printNumbers(zahlen, z -> z % 2 == 0);
    printNumbers(zahlen, z -> z == (int) Math.sqrt(z) * (int) Math.sqrt(z));
  }
  public static void printNumbers(Integer[] daten, Predicate<Integer> kriterium) {
    for (Integer i : daten)
    if (kriterium.test(i))
    System.out.print(i + " ");
    System.out.println();
  }
}
```