Kapitel 1
Neues in Java 9

»Jede Lösung eines Problems ist ein neues Problem.«
– Johann Wolfgang von Goethe (1749–1832)

Dieses Kapitel fasst die wesentlichen Änderungen von Java 9 kompakt zusammen.

1.1 Klassenlader (Class Loader) und Modul-/Klassenpfad

Ein Klassenlader ist dafür verantwortlich, die Binärrepräsentation einer Klasse aus einem Hintergrundspeicher oder Hauptspeicher zu laden. Aus der Datenquelle (im Allgemeinen die *.class*-Datei) liefert der Klassenlader ein Byte-Array mit den Informationen, die im zweiten Schritt dazu verwendet werden, die Klasse ins Laufzeitsystem einzubringen; das ist *Linking*. Es gibt vordefinierte Klassenlader und die Möglichkeit, eigene Klassenlader zu schreiben, um etwa verschlüsselte vom Netzwerk zu beziehen oder komprimierte *.class*-Dateien aus Datenbanken zu laden.

1.1.1 Klassenladen auf Abruf

Nehmen wir zu Beginn ein einfaches Programm mit zwei Klassen:

```
class Person {
  static String NOW = java.time.LocalDateTime.now().toString();

  public static void main( String[] args ) {
    new Dog();
  }
}

class Dog {
  Person master;
}
```

Wenn die Laufzeitumgebung das Programm `Person` startet, muss sie eine Reihe von Klassen laden. Das tut sie dynamisch zur Laufzeit. Sofort wird klar, dass es zumindest `Person` sein

muss. Wenn aber die statische `main(String[])`-Methode aufgerufen wird, muss auch `Dog` geladen sein. Und da beim Laden einer Klasse auch die statischen Variablen initialisiert werden, wird auch die Klasse `LocalDateTime` geladen.

Zwei weitere Dinge werden nach einiger Überlegung deutlich:

- Wenn `Dog` geladen wird, bezieht es sich auf `Person`. Da `Person` aber schon geladen ist, muss es nicht noch einmal geladen werden.
- Unsichtbar stecken noch andere referenzierte Klassen dahinter, die nicht direkt sichtbar sind. So wird zum Beispiel `Object` geladen, da implizit in der Klassendeklaration von `Person` steht: `class Person extends Object`. Auch `String` muss geladen werden, weil `String` einmal in der Signatur von `main(String[])` vorkommt und es der Typ von `now` ist. Intern ziehen die Typen viele weitere Typen nach sich. `String` implementiert `Serializable`, `CharSequence` und `Comparable`, also müssen diese drei Schnittstellen auch geladen werden. Und so geht das weiter, je nachdem, welche Programmpfade abgelaufen werden. Wichtig ist aber, zu verstehen, dass diese Klassendateien so spät wie möglich geladen werden.

Im Beispiel mit den Klassen `Person` und `Dog` lädt die Laufzeitumgebung selbstständig die Klassen (*implizites Klassenladen*). Klassen lassen sich auch mit `Class.forName(String)` über ihren Namen laden (*explizites Klassenladen*).

1.1.2 Klassenlader bei der Arbeit zusehen

Um zu sehen, welche Klassen überhaupt geladen werden, lässt sich der virtuellen Maschine beim Start der Laufzeitumgebung ein Schalter mitgeben: `-verbose:class`. Dann gibt die Maschine beim Lauf alle Typen aus, die sie lädt.

Nehmen wir das Beispiel, lassen jedoch die Variable `NOW` erst einmal heraus:

```
class Person {
//  static String NOW = java.time.LocalDateTime.now().toString();

  public static void main( String[] args ) {
    new Dog();
  }
}

class Dog {
  Person master;
}
```

Die Ausgabe mit dem aktivierten Schalter ist über 500 Zeilen lang; ein Ausschnitt:


```
[0.015s][info][class,load] opened: C:\Program Files\Java\jdk-9\lib\modules
[0.029s][info][class,load] java.lang.Object source: jrt:/java.base
[0.029s][info][class,load] java.io.Serializable source: jrt:/java.base
[0.029s][info][class,load] java.lang.Comparable source: jrt:/java.base
[0.029s][info][class,load] java.lang.CharSequence source: jrt:/java.base
[0.029s][info][class,load] java.lang.String source: jrt:/java.base
[0.030s][info][class,load] java.lang.reflect.AnnotatedElement source: jrt:/java.base
...
[0.197s][info][class,load] sun.security.util.Debug source: jrt:/java.base
[0.197s][info][class,load] Person source: file:/C:/Users/Christian/Dropbox/ ↵
Eigene%20Dokumente/Insel/programme/Spielwiese/bin/
[0.198s][info][class,load] java.lang.NamedPackage source: jrt:/java.base
[0.198s][info][class,load] java.lang.PublicMethods$MethodList source: jrt:/java.base
[0.198s][info][class,load] java.lang.PublicMethods$Key source: jrt:/java.base
[0.198s][info][class,load] java.lang.Void source: jrt:/java.base
[0.199s][info][class,load] Dog source: file:/C:/Users/Christian/Spielwiese/bin/
[0.199s][info][class,load] java.lang.Shutdown source: jrt:/java.base
[0.199s][info][class,load] java.lang.Shutdown$Lock source: jrt:/java.base
```

Nehmen wir nun die `NOW`-Zeile mit hinein, so führt das zu 200 zusätzlich geladenen Klassen.

1.1.3 JMOD-Dateien und JAR-Dateien

Der Klassenlader bezieht *.class*-Dateien nicht nur aus Verzeichnissen, sondern in der Regel aus Containern. So müssen keine Verzeichnisse ausgetauscht werden, sondern nur einzelne Dateien. Als Container-Formate finden wir JMOD (neu in Java 9) und JAR. Wenn Java-Software ausgeliefert wird, bieten sich JAR- oder JMOD-Dateien an, denn es ist einfacher und platzsparender, nur ein komprimiertes Archiv weiterzugehen als einen großen Dateibaum.

JAR-Dateien

Sammlungen von Java-Klassendateien und Ressourcen werden in der Regel in *Java-Archiven*, kurz *JAR-Dateien*, zusammengefasst. Diese Dateien sind im Grunde ganz normale ZIP-Archive mit einem besonderen Verzeichnis *META-INF* für Metadateien. Das JDK bringt im *bin*-Verzeichnis das Werkzeug *jar* zum Aufbau und Extrahieren von JAR-Dateien mit.

JAR-Dateien behandelt die Laufzeitumgebung wie Verzeichnisse von Klassendateien und Ressourcen. Zudem haben Java-Archive den Vorteil, dass sie signiert werden können und illegale Änderungen auffallen. JAR-Dateien können Modulinformationen beinhalten, dann heißen sie englisch *modular JAR*.

JMOD-Dateien

Das Format JMOD ist speziell für Module und neu in Java 9 – es organisiert Typen und Ressourcen. Zum Auslesen und Packen gibt es im *bin*-Verzeichnis des JDK das Werkzeug *jmod*.

Hinweis

Die JVM greift selbst nicht auf diese Module zurück. Achten wir auf die Ausgabe vom letzten Programm, dann steht in der ersten Zeile:

```
[0.015s][info][class,load] opened: C:\Program Files\Java\jdk-9\lib\modules
```

Die Datei *module* ist ca. 170 MiB groß und in einem proprietären Dateiformat.

JAR vs. JMOD

Module können in JMOD- und JAR-Container gepackt werden. Wenn ein JAR kein Modular JAR ist, also keine Modulinformationen enthält, so fehlen zentrale Informationen, wie Abhängigkeiten oder eine Version; ein JMOD ist immer ein benanntes Modul.

JMOD-Dateien sind nicht so flexibel wie JAR-Dateien, denn sie können nur zur Übersetzungszeit und zum Linken eines Runtime-Images – dafür gibt es das Kommandozeilenwerkzeug `jlink` – genutzt werden. JMOD-Dateien können nicht wie JAR-Dateien zur Laufzeit verwendet werden. Das Dateiformat ist proprietär und kann sich jederzeit ändern, es ist nichts Genaues spezifiziert.[1] Einziger Vorteil von JMOD: Native Bibliotheken lassen sich standardisiert einbinden.

1.1.4 Woher die kleinen Klassen kommen: die Suchorte und spezielle Klassenlader

Die Laufzeitumgebung nutzt zum Laden nicht nur einen Klassenlader, sondern mehrere. Das ermöglicht, unterschiedliche Orte für die Klassendateien festzulegen. Ein festes Schema bestimmt die Suche nach den Klassen:

1. Klassentypen wie `String`, `Object` oder `Point` stehen in einem ganz speziellen Archiv. Wenn ein eigenes Java-Programm gestartet wird, so sucht die virtuelle Maschine die angeforderten Klassen zuerst in diesem Archiv. Da es elementare Klassen sind, die zum Hochfahren eines Systems gehören, werden sie *Bootstrap-Klassen* genannt. Die Implementierung dieses *Bootstrap-Klassenladers* ist Teil der Laufzeitumgebung.
2. Ist eine Klasse keine Bootstrap-Klasse, beginnt der *System-Klassenlader* bzw. *Applikations-Klassenlader* die Suche im *Modulpfad* (ehemals *Klassenpfad*/*Classpath*). Diese Pfadangabe besteht aus einer Aufzählung von Modulen, in denen die Laufzeitumgebung nach den Klassendateien und Ressourcen sucht.

1 Die *http://openjdk.java.net/jeps/261* macht die Aussage, dass es ein ZIP ist.

1.1.5 Setzen des Modulpfades

Wo die JVM die Klassen findet, muss ihr mitgeteilt werden, und das ist in der Praxis elementar für die Auslieferung, auch englisch *deployment* genannt. Java wartet mit dem Laden der Klassen so lange, bis sie benötigt werden. Es gibt zum Beispiel Programmabläufe nur zu besonderen Bedingungen, und wenn dann erst spät ein neuer Typ referenziert wird, der nicht vorhanden ist, fällt dieser Fehler erst sehr spät auf. Dem Compiler müssen folglich nicht nur die Quellen für Klassen und Ressourcen der eigenen Applikation mitgeteilt werden, sondern alle vom Programm referenzierten Typen aus zum Beispiel quelloffenen und kommerziellen Bibliotheken. Sollen in einem Java-Projekt Dateien aus einem Verzeichnis oder einem externen Modul geholt werden, so ist der übliche Weg, diese Dateien im Modulpfad anzugeben. Diese Angabe ist für alle SDK-Werkzeuge notwendig – am häufigsten ist sie beim Compiler und bei der Laufzeitumgebung zu sehen.

Setzen des Klassenpfades

Vor Java 9 gab es nur JAR-Dateien und Verzeichnisse im Klassenpfad. Auch wenn es ab Java 9 weiterhin den Klassenpfad gibt, sollte er auf lange Sicht leer sein. Es gibt zwei Möglichkeiten zur Aufnahme von Verzeichnissen und JAR-Dateien in den Klassenpfad:

- ein Schalter
- eine Umgebungsvariable

Schalter -classpath

Die Suchorte lassen sich flexibel angeben, wobei die erste Variante einem SDK-Werkzeug über den Schalter `-classpath` (kurz `-cp`) die Klassendateien bzw. Archive liefert:

```
$ java -classpath classpath1;classpath2 mein.paket.MainClass
```

Der Klassenpfad enthält Wurzelverzeichnisse der Pakete und JAR-Dateien, also Archive von Klassendateien und Ressourcen.

Beispiel

Nimm ein Java-Archiv *library.jar* im aktuellen Verzeichnis, die Ressourcen unter dem *bin*-Verzeichnis und alle JAR-Dateien im Verzeichnis *lib* in den Klassenpfad mit auf:

```
$ java -cp "library.jar;bin/.;lib/*" mein.paket.MainClass
```

Unter Windows ist der Trenner ein Semikolon, unter Unix ein Doppelpunkt! Das Sternchen steht für *alle* JAR-Dateien, es ist *keine* übliche Wildcard, wie z. B. *parser*.jar*.[2] Sehen Kommandozeilen der Betriebssysteme ein *, beginnen sie in der Regel eine eigene Verarbeitung; daher muss die gesamte Pfadangabe in doppelten Anführungszeichen stehen.

2 Weitere Details unter *https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html*

Umgebungsvariable CLASSPATH

Eine Alternative zum Schalter `-cp` ist das Setzen der Umgebungsvariablen `CLASSPATH` mit einer Zeichenfolge, die Pfadangaben spezifiziert:

```
$ SET CLASSPATH=classpath1;classpath2
$ java mein.paket.MeinClass
```

Problematisch ist der globale Charakter der Variablen, sodass lokale `-cp`-Angaben besser sind. Außerdem »überschreiben« die `-cp`-Optionen die Einträge in `CLASSPATH`. Zu guter Letzt: Ist weder `CLASSPATH` noch eine `-cp`-Option gesetzt, besteht der Klassenpfad für die JVM nur aus dem aktuellen Verzeichnis, also ».«.

Um in Eclipse den Klassenpfad zu erweitern, damit etwa die Klassendateien von Java-Archiven berücksichtigt werden, ist Folgendes zu tun: Im Projekt das Kontaktmenü öffnen und PROPERTIES aufrufen, dann links unter JAVA BUILD PATH gehen und anschließend im Reiter LIBRARIES entweder ADD JARS... (JARs sind im Projekt) oder ADD EXTERNAL JARS... (JAR-Dateien liegen nicht im Projekt, sondern irgendwo anders im Dateisystem) nutzen.

Hinweis

Die so genannten Bootstrap-Klassen aus den Paketen `java(x).*` (wie `Object`, `String`) stehen nicht im `CLASSPATH`.

Classpath-Hell

Java-Klassen in JAR-Dateien auszuliefern, ist der übliche Weg, es gibt aber zwei Probleme:

1. Aus Versehen können zwei JAR-Dateien mit unterschiedlichen Versionen im Klassenpfad liegen. Nehmen wir an, es sind *parser-1.2.jar* und *parser-2.0.jar*, wobei sich bei der neuen Version API und Implementierung leicht geändert haben. Das fällt vielleicht am Anfang nicht auf, denn einen Ladefehler gibt es für den Typ nicht, er ist ja da – die JVM nimmt den ersten Typ, den sie findet. Nur wenn ein Programm auf die neue API zurückgreift, aber die geladene Klasse vom alten JAR stammt, knallt es zur Laufzeit. Bei doppelten JARs mit unterschiedlichen Versionen führt eine Umsortierung im Klassenpfad zu einem ganz anderen Ergebnis. Zum Glück lässt sich das Problem relativ schnell lösen.
2. Zwei Java-Bibliotheken – nennen wir sie *vw.jar* und *audi.jar* – benötigen je eine Neben-JAR zum Arbeiten. Doch während *vw.jar* die Version *bosch-1.jar* benötigt, benötigt *audi.jar* die Version *bosch-2.jar*. Das ist ein Problem, denn JARs sind im Standard-Klassenpfad immer global, aber nicht hierarchisch, es kann also kein JAR ein »lokales« Unter-JAR haben.

Lösungen für das zweite Problem gibt es einige, wobei zu neuen Klassenladern gegriffen wird. Bekannt ist OSGi, das in der Java-Welt aber etwas an Fahrt verloren hat.

1.2 Module entwickeln und einbinden

Das *JPMS (Java Platform Module System*), auch unter dem Projektnamen *Jigsaw* bekannt, ist eine der größten Neuerungen in Java 9. Im Mittelpunkt steht die starke Kapselung: Implementierungsdetails kann ein Modul geheim gehalten. Selbst Hilfscode innerhalb des Moduls, auch wenn er öffentlich ist, darf nicht nach außen dringen. Zweitens kommt eine Abstraktion von Verhalten über Schnittstellen hinzu, die interne Klassen aus dem Modul implementieren können, wobei dem Nutzer die konkreten Klassen nicht bekannt sind. Als dritten Punkt machen explizite Abhängigkeiten die Interaktion mit anderen Modulen klar. Eine grafische Darstellung hilft auch bei großen Architekturen, die Übersicht über Nutzungsbeziehungen zu behalten.

1.2.1 Wer sieht wen

Klassen, Pakete und Module lassen sich als Container mit unterschiedlichen Sichtbarkeiten sehen:

- Ein Typ, sei es Klasse oder Schnittstelle, enthält Attribute und Methoden.
- Ein Paket enthält Typen.
- Ein Modul enthält Pakete.
- Private Eigenschaften in einem Typ sind nicht in anderen Typen sichtbar.
- Nicht öffentliche Typen sind in anderen Paketen nicht sichtbar.
- Nicht exportierte Pakete sind außerhalb eines Moduls nicht sichtbar.

Ein Modul ist definiert

1. durch einen Namen,
2. durch die Angabe, was es exportiert möchte und
3. welches Modul es zur Arbeit selbst benötigt.

Interessant ist der zweite Aspekt, also dass ein Modul etwas exportiert. Wenn nichts exportiert wird, ist auch nichts sichtbar nach außen. Alles, was Außenstehende sehen sollen, muss in der Modulbeschreibung aufgeführt sein – nicht alle öffentlichen Typen des Moduls sind standardmäßig öffentlich, dann wäre das kein Fortschritt zu JAR-Dateien. Mit dem neuen Modulsystem haben wir also eine ganz andere Sichtbarkeit. Aus der Viererbande `public`, `private`, paketsichtbar, `protected` bekommt `public` eine viel feinere Abstufung. Denn was `public` ist, bestimmt das Modul, und das sind:

- Typen, die das Modul für alle exportiert
- Typen für explizit aufgezählte Module
- alle Typen im gleichen Modul

Der Compiler und die JVM achten auf die Einhaltung der Sichtbarkeit, und auch Tricks mit Reflection sind nicht mehr möglich, wenn ein Modul keine Freigabe erteilt hat.

Modultypen

Wir wollen uns in dem Abschnitt intensiver mit drei Modultypen beschäftigen. Wenn wir neue Module schreiben, dann sind das *benannte Module*. Daneben gibt es aus Kompatibilitätsgründen *automatische Module* und *unbenannte Module*, mit denen wir vorhandene JAR-Dateien einbringen können. Die Bibliothek der Java SE ist selbst in Module unterteilt, wir nennen sie *Plattform-Module*.

Die Laufzeitumgebung zeigt mit einem Schalter `--list-modules` alle Plattform-Module an.

Beispiel

Liste die ca. 70 Module auf:

```
$ java --list-modules
java.activation@9
java.base@9
java.compiler@9
…
oracle.desktop@9
oracle.net@9
```

Im Ordner *C:\Program Files\Java\jdk-9\jmods* liegen JMOD-Dateien.

1.2.2 Plattform-Module und JMOD-Beispiel

Das Kommandozeilenwerkzeug `jmod` zeigt an, was ein Modul exportiert und benötigt. Nehmen wir die JDBC-API für Datenbankverbindungen als Beispiel; die Typen sind in einem eigenen Modul mit den Namen `java.sql`.

```
C:\Program Files\Java\jdk-9\bin>jmod describe ..\jmods\java.sql.jmod
java.sql@9
exports java.sql
exports javax.sql
exports javax.transaction.xa
requires java.base mandated
```



```
requires java.logging transitive
requires java.xml transitive
uses java.sql.Driver
platform windows-amd64
```

Wir können ablesen:

- den Namen
- die Pakete, die das Modul exportiert: `java.sql`, `javax.sql` und `javax.transation.xa`
- die Module, die `java.sql` benötigt: `java.base` ist hier immer drin, dazu kommen `java.logging` und `java.xml`.

Die Meldung mit »uses« steht im Zusammenhang mit dem Service-Locator – wir können das vorerst ignorieren. Die Kennung über die Plattform (`windows-amd64`) schreibt `jmod` mit hinein, es ist die Belegung der System-Property `os.arch` auf dem Build-Server.

1.2.3 Verbotene Plattformeigenschaften nutzen, --add-exports

Als Sun von vielen Jahren mit der Entwicklung der Java-Bibliotheken begann, kamen viele interne Hilfsklassen mit in die Bibliothek. Viele beginnen mit den Paketpräfixen `com.sun` und `sun`. Die Typen wurden immer als interne Typen kommuniziert, doch bei einigen Entwicklern waren die Neugierde und das Interesse so groß, dass die Warnungen von Sun/Oracle ignoriert wurden. In Java 9 kommt der große Knall, da `public` nicht mehr automatisch `public` für alle Klassen außerhalb des Moduls ist; die internen Klassen werden nicht mehr exportiert, sind also nicht mehr benutzbar. Es kommt zu einem Compilerfehler, wie in folgendem Beispiel:

```
public class ShowRuntimeArguments {
  public static void main( String[] args ) throws Exception {
    System.out.println( java.util.Arrays.toString(
                          jdk.internal.misc.VM.getRuntimeArguments() ) );
  }
}
```

Unser Programm greift auf die VM-Klasse zurück, um die eigentliche Belegung der Kommandozeile zu erfragen. In der `main(String[] args)`-Methode sind in `args` keine VM-Argumente enthalten.

Übersetzen wir das Programm, gibt es einen Compilerfehler (nicht bei einem Java 8-Compiler):

```
$ javac ShowRuntimeArguments.java
ShowRuntimeArguments.java:3: error: package jdk.internal.misc is not visible
    System.out.println( java.util.Arrays.toString( jdk.internal.misc.VM.getRuntime
```

```
Arguments() ) );
                                                          ^
  (package jdk.internal.misc is declared in module java.base, which does not export

it to the unnamed module)
1 error

The module java.base does not export the package jdk.internal.misc., so the type

jdk.internal.misc.Unsafe is not accessible - as a consequence compilation fails.
```

Das Problem dokumentiert der Compiler. Es ist dadurch zu lösen, indem mit dem Schalter `--add-exports` aus dem Modul `java.base` das Paket `jdk.internal.misc` unserer Klasse bereitgestellt wird. Die Angabe ist für den Compiler und für die Laufzeitumgebung zu setzen:

```
$ javac --add-exports java.base/jdk.internal.misc=ALL-UNNAMED
ShowRuntimeArguments.java
$ java  ShowRuntimeArguments
Exception in thread "main" java.lang.IllegalAccessError: class ShowRuntimeArguments ( ↩
in unnamed module @0x77afea7d) cannot access class jdk.internal.misc.VM (in module ↩
java.base) because module java.base does not export jdk.internal.misc to unnamed ↩
module @0x77afea7d
        at ShowRuntimeArguments.main(ShowRuntimeArguments.java:3)
$ java --add-exports java.base/jdk.internal.misc=ALL-UNNAMED ShowRuntimeArguments
[--add-exports=java.base/jdk.internal.misc=ALL-UNNAMED]
```

Wir sehen die Ausgabe, das Programm funktioniert.

Eine Angabe wie `java.base/jdk.internal.misc`, bei der vorne das Modul steht und hinter dem / der Paketname, ist oft ab Java 9 anzutreffen. Hinter dem Gleichheitszeichen steht entweder unser Paket, welches die Typen in `jdk.internal.misc` sehen kann, oder – wie in unserem Fall – `ALL-UNNAMED`.

jdeps

Hätten wir das Programm schon erfolgreich unter Java 9 übersetzt, würde es zur Laufzeit ebenfalls knallen. Da es nun sehr viel Programmcode gibt, haben die Java-Entwickler bei Oracle das Kommandozeilenprogramm `jdeps` entwickelt. Es meldet, wenn interne Typen im Programm vorkommen:

```
$ jdeps ShowRuntimeArguments.class
ShowRuntimeArguments.class -> java.base
```



```
<unnamed>            -> java.io                  java.base
<unnamed>            -> java.lang                java.base
<unnamed>            -> java.util                java.base
<unnamed>            -> jdk.internal.misc        JDK internal API (java.base)
```

Die Meldung »JDK internal API« bereitet uns darauf vor, dass es gleich Ärger geben wird.

So kann relativ leicht eine große Codebasis untersucht werden, und Entwickler können proaktiv den Stellen auf den Grund gehen, die problematische Abhängigkeiten haben.

1.2.4 Plattformmodule einbinden, --add-modules und --add-opens

Jedes Java SE-Projekt basiert auf dem Modul `java.se`, was diverse Modulabhängigkeiten nach sich zieht.

Diverse Module sind nicht Teil vom Modul Java SE, unter anderem sind das das Java Activation Framework, CORBA, Transaction-API, JAXB, Web-Services und interne Module, die mit `jdk` beginnen.

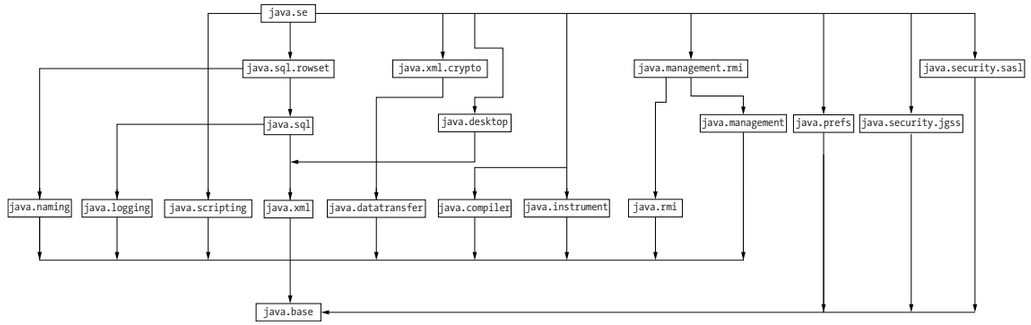

Abbildung 1.1 Modulabhängigkeiten von java.se

Starten wir ein Programm mit Bezug zu einer dieser Bibliotheken, gibt es einen Fehler. Zunächst zu dem Programm, das ein Objekt automatisch in XML konvertieren soll:

```
public class Person {
  public String name = "Chris";
  public static void main( String[] args ) {
    javax.xml.bind.JAXB.marshal( new Person(), System.out );
  }
}
```

Ausgeführt auf der Kommandozeile folgt ein Fehler:

```
$ java Person
Exception in thread "main" java.lang.NoClassDefFoundError: javax/xml/bind/JAXB
 at Person.main(Person.java:6)
Caused by: java.lang.ClassNotFoundException: javax.xml.bind.JAXB
 at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(Unknown Source)
 at java.base/
jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Unknown Source)
 at java.base/java.lang.ClassLoader.loadClass(Unknown Source)
 ... 1 more
```

Wir müssen das Modul `java.xml.bind` (oder auch das »Übermodul« `java.se.ee`) mit angeben; dafür dient der Schalter `--add-modules`.

```
$ java --add-modules java.xml.bind Person
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
    <name>Chris</name>
</person>
```

Öffnen für Reflection

Jetzt gibt es allerdings ein anderes Problem, das auffällt, wenn wir einen anderen Typ in XML umwandeln wollen:

```
public class Today {
  public static void main( String[] args ) {
    javax.xml.bind.JAXB.marshal( new java.util.Date(), System.out );
  }
}
```

Auf der Kommandozeile zeigt sich:

```
$ java --add-modules java.xml.bind Today
Exception in thread "main" javax.xml.bind.DataBindingException:
javax.xml.bind.JAXBException: Package java.util with JAXB class java.util.Date
defined in a module java.base must be open to at least java.xml.bind module.
 at java.xml.bind@9/javax.xml.bind.JAXB._marshal(Unknown Source)
 at java.xml.bind@9/javax.xml.bind.JAXB.marshal(Unknown Source)
 at Today.main(Today.java:4)
Caused by: javax.xml.bind.JAXBException: Package java.util with JAXB class java.util.
Date defined in a module java.base must be open to at least java.xml.bind module.
 at java.xml.bind@9/javax.xml.bind.ModuleUtil.delegateAddOpensToImplModule(

Unknown Source)
```



```
 at java.xml.bind@9/javax.xml.bind.ContextFinder.newInstance(Unknown Source)
 at java.xml.bind@9/javax.xml.bind.ContextFinder.newInstance(Unknown Source)
...
```

Die zentrale Information ist »Package java.util with JAXB class java.util.Date defined in a module java.base must be open to at least java.xml.bind module.« Wir müssen etwas öffnen; dazu verwenden wir den Schalter `--add-opens`:

```
$ java --add-modules java.xml.bind --add-opens java.base/java.util=java.xml.bind Box
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<date>2017-09-02T23:20:52.170+02:00</date>
```

Die Option öffnet für Reflection das Paket `java.util` aus dem Modul `java.base` für `java.xml.bind`. Neben `--add-opens` gibt es das ähnliche `--add-exports`, das alle öffentlichen Typen und Eigenschaften zur Übersetzungs-/Laufzeit öffnet; `--add-opens` geht für Reflection einen Schritt weiter.

1.2.5 Projektabhängigkeiten in Eclipse

Um Module praktisch umzusetzen, wollen wir in Eclipse zwei neue Java-Projekte aufbauen: *com.tutego.greeter* und *com.tutego.main*. Wir legen im Projekt *com.tutego.greeter* eine Klasse `com.tutego.insel.greeter.Greeter` an und in *com.tutego.main* die Klasse `com.tutego.insel.main.Main`. Im Package-Explorer sieht das so aus:

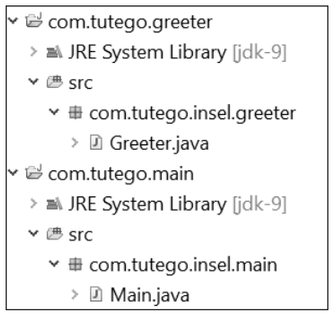

Abbildung 1.2 Java-Projekte com.tutego.greeter und com.tutego.main im Package-Explorer

Jetzt ist eine wichtige Vorbereitung in Eclipse nötig: Wir müssen einstellen, dass *com.tutego.main* das Java-Projekt *com.tutego.greeter* benötigt. Dazu gehen wir auf das Projekt *com.tutego.main* und rufen im Kontextmenü PROJECT auf, alternativ im Menüpunkt PROJECT • PROPERTIES oder über die Tastenkombination Alt + ↵. Im Dialog navigiere links auf JAVA BUILD PATH und aktiviere den Reiter PROJECTS. Wähle ADD..., und im Dialog wähle aus der Liste COM.TUTEGO.GREETER. OK schließt den kleinen Dialog, und unter REQUIRED PROJECTS IN BUILD PATH taucht eine Abhängigkeit auf.

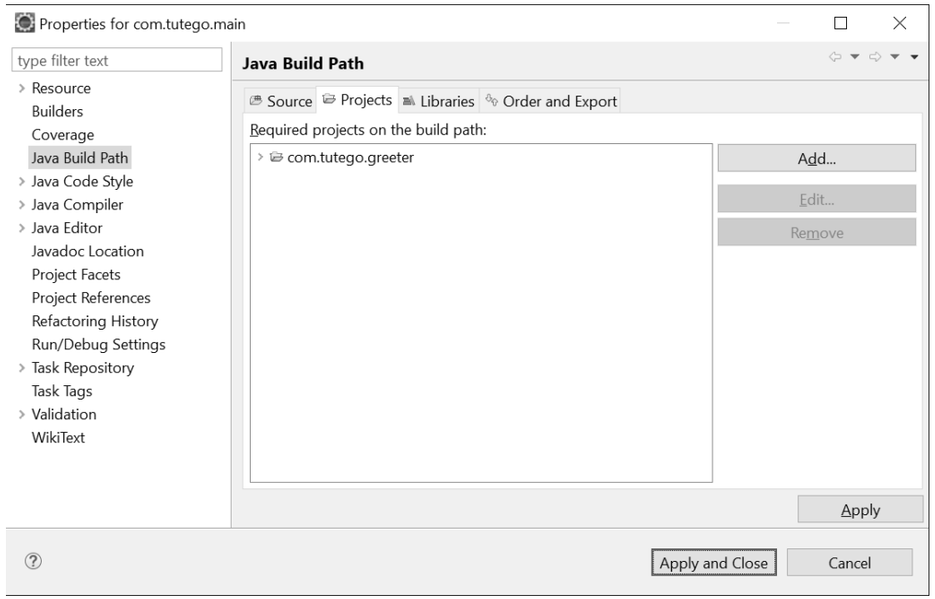

Abbildung 1.3 Abhängigkeit des Eclipse-Projektes com.tutego.main von com.tutego.greeter

Wir können jetzt zwei einfache Klassen implementieren. Zunächst für das Projekt *com.tutego.greeter*:

Listing 1.1 com/tutego/insel/greeter/Greeter.java

```
package com.tutego.insel.greeter;

public class Greeter {

  private Greeter() { }

  public static Greeter instance() {
    return new Greeter();
  }

  public void greet( String name ) {
    System.out.println( "Hey "+ name );
  }
}
```

Und die Hauptklasse im Projekt *com.tutego.main*:

Listing 1.2 com/tutego/insel/main/Main

```
package com.tutego.insel.main;

import com.tutego.insel.greeter.Greeter;

public class Main {

  public static void main( String[] args ) {
    Greeter.instance().greet( "Chris" );
  }
}
```

Da wir in Eclipse vorher die Abhängigkeit gesetzt haben, gibt es keinen Compilerfehler.

1.2.6 Benannte Module und module-info.java

Die Modulinformationen werden über eine Datei *module-info.java* (kurz Modulinfodatei) deklariert, Annotationen kommen nicht zum Einsatz. Diese zentrale Datei ist der Hauptunterschied zwischen einem Modul und einer einfachen JAR-Datei. In dem Moment, in dem die spezielle Klassendatei *module-info.class* im Modulpfad ist, beginnt die Laufzeitumgebung, das Projekt als Modul zu interpretieren.

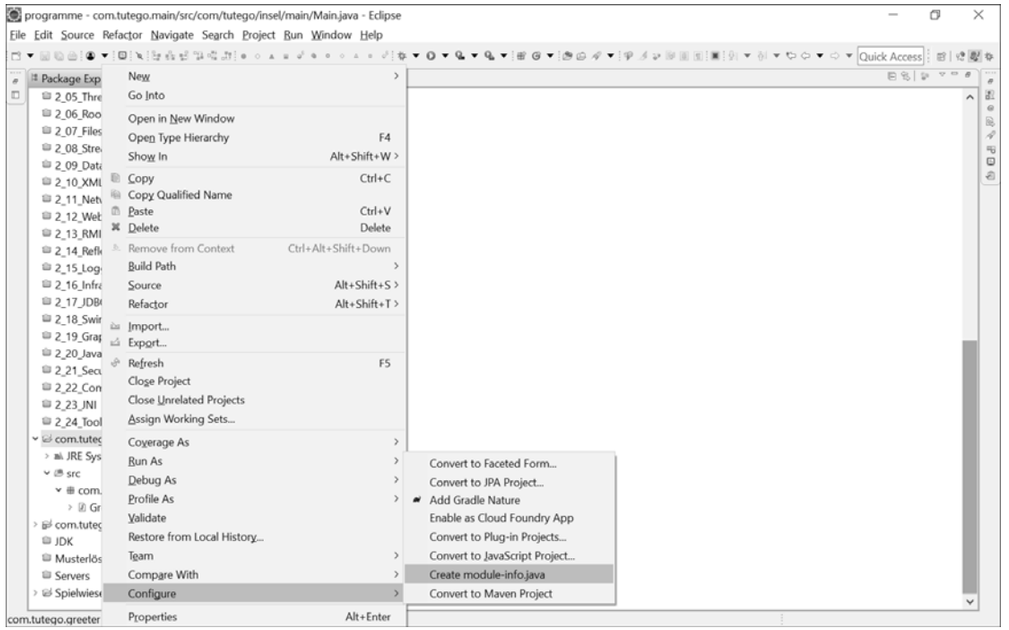

Abbildung 1.4 Datei module-info anlegen

Testen wir das, indem wir in unsere Projekte *com.tutego.greeter* und *com.tutego.main* eine Modulinfodatei anlegen. Das kann Eclipse über das Kontextmenü CONFIGURE • CREATE MODULE-INFO.JAVA für uns machen.

Für das erste Modul `com.tutego.greeter` entsteht:

Listing 1.3 module-info.java

```
/**
 *
 */
/**
 * @author Christian
 *
 */
module com.tutego.greeter {
  exports com.tutego.insel.greeter;
  requires java.base;
}
```

Und für die zweite Modulinfodatei – Kommentare ausgeblendet:

Listing 1.4 module-info.java

```
module com.tutego.main {
  exports com.tutego.insel.main;
  requires com.tutego.greeter;
  requires java.base;
}
```

Hinter dem Schlüsselwort `module` steht der Name des Moduls, den Eclipse automatisch so wählt, wie das Eclipse-Projekt heißt.[3] Es folgt ein Block in geschweiften Klammern.

Zwei Schlüsselwörter fallen ins Auge, die wir schon vorher bemerkt haben: `exports` und `requires`.

- Das Projekt/Modul `com.tutego.greeter` exportiert das Paket `com.tutego.insel.greeter`. Andere Pakete nicht. Es benötigt (`requires`) `java.base`, wobei das Modul Standard ist und die Zeile gelöscht werden kann.
- Das Projekt/Modul `com.tutego.main` exportiert das Paket `com.tutego.insel.main`, und es benötigt `com.tutego.greeter` – diese Information nimmt sich Eclipse selbstständig aus den Projektabhängigkeiten.

3 Zur Benennung von Modulen gibt es Empfehlungen in dem englischsprachigen Beitrag *http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2017-May/000687.html*.

Info

Ein Modul `required` ein anderes Modul, aber `exports` ein Paket.

Beginnen wir mit den Experimenten in den beiden `module-info.java`-Dateien:

Modul	Aktion	Ergebnis
`com.tutego.greeter` `com.tutego.main`	`// requires java.base;`	Auskommentieren führt zu keiner Änderung, da `java.base` immer required wird.
`com.tutego.greeter`	`// exports com.tutego.insel.greeter;`	Compilerfehler im `main`-Modul: »The type com.tutego.insel.greeter.Greeter is not accessible«
`com.tutego.greeter`	`exports com.tutego.insel.greeter to god;`	Nur das Modul `god` bekommt Zugriff auf `com.tutego.insel.greeter`. Das `MAIN`-Modul meldet »The type com.tutego.insel.greeter.Greeter is not accessible«.
`com.tutego.greeter`	`exports com.tutego.insel.closer;`	Hinzufügen führt zum Compilerfehler »The package com.tutego.insel.closer does not exist or is empty«.
`com.tutego.main`	`// requires com.tutego.greeter;`	Compilerfehler »The import com.tutego.insel.greeter cannot be resolved«.
`com.tutego.main`	`// exports com.tutego.insel.main;`	Keins, denn `c.t.i.m` wird von keinem Modul required

Tabelle 1.1 Modulsyntax und ihre Effekte

Die Zeile mit `exports com.tutego.insel.greeter to god` zeigt einen qualifizierten Export.

Übersetzen und Packen von der Kommandozeile

Setzen wir ins Wurzelverzeichnis des Moduls `com.tutego.greeter` ein Batch-Skript *compile.bat*:

Listing 1.5 compile.bat

```
set PATH=%PATH%;C:\Program Files\Java\jdk-9\bin
rmdir /s /q lib
mkdir lib
javac -d bin src\module-info.java src\com\tutego\insel\greeter\Greeter.java
jar --create --file=lib/com.tutego.greeter@1.0.jar --module-version=1.0 -C bin .
jar --describe-module --file=lib/com.tutego.greeter@1.0.jar
```

Folgende Schritte führt das Skript aus:

1. Setzen der `PATH`-Variablen für die JDK-Tools
2. Löschen eines vielleicht schon angelegten *lib*-Ordners
3. Anlegen eines neuen *lib*-Orders für die JAR-Datei
4. Übersetzen der zwei Java-Dateien in den Zielordner *bin*
5. Anlegen einer JAR-Datei. `--create` (abkürzbar zu `-c`) instruiert das Werkzeug, eine neue JAR-Datei anzulegen. `-file` (oder kurz `-f`) bestimmt den Zielnamen, `-module-version` unsere Versionsnummer, und `-C` wechselt das Verzeichnis und beginnt ab dort, die Dateien einzusammeln. Die Kommandozeilensyntax beschreibt Oracle auf der Webseite *https://docs.oracle.com/javase/9/tools/jar.htm*.
6. Die Option `--describe-module` (oder kurz `-d`) zeigt die Modulinformation und führt zu folgender (vereinfachten) Ausgabe: `com.tutego.greeter@1.0 jar:file:///C:/…/com.tutego.greeter/lib/com.tutego.greeter@1.0.jar/!module-info.class exports com.tutego.insel.greeter requires java.base`.

Für das zweite Projekt ist die *compile.bat* sehr ähnlich, dazu kommt ein Aufruf der JVM, um das Programm zu starten.

Listing 1.6 compile.bat

```
set PATH=%PATH%;C:\Program Files\Java\jdk-9\bin
rmdir /s /q lib
mkdir lib
javac -d bin --module-path ..\com.tutego.greeter\lib src\module-info.java ⮌
  src\com\tutego\insel\main\Main.java
jar -c -f=lib/com.tutego.main@1.0.jar --main-class=com.tutego.insel.main.Main ⮌
  --module-version=1.0 -C bin .
java -p lib;..\com.tutego.greeter\lib -m com.tutego.main
```

Änderungen gegenüber dem ersten Skript sind:

1. Beim Compilieren müssen wir den Modulpfad mit `--module-path` (oder kürzer mit `-p`) angeben, weil ja das Modul `com.tutego.greeter` required ist.

2. Beim Anlegen der JAR-Datei geben wir über `-main-class` die Klasse mit der `main(…)`-Methode an.
3. Startet die JVM das Programm, lädt sie das Hauptmodul und alle abhängigen Module. Wir geben beide *lib*-Order mit den JAR-Dateien an und mit `-m` das sogenannte *initiale Modul* für die Hauptklasse.

1.2.7 Automatische Module

JAR-Dateien spielen seit 20 Jahren eine zentrale Rolle im Java-System; sie vom einen zum anderen Tag abzuschaffen, würde große Probleme bereiten. Ein Blick auf *https://mvnrepository.com/repos* offenbart über 7,7 Millionen Artefakte; es gehen auch Dokumentationen und andere Dateien in die Statistik ein, doch es gibt eine Größenordnung, wie viele JAR-Dateien im Umlauf sind.

Damit JAR-Dateien unter Java 9 eingebracht werden können, gibt es zwei Lösungen: das JAR in den Klassenpfad oder in den neuen Modulpfad zu setzen. Kommt ein JAR in den Modulpfad und hat es keine Modulinfodatei, entsteht ein *automatisches Modul*. Bis auf eine kleine Einschränkung funktioniert das für die meisten existierenden Java-Bibliotheken.

Ein automatisches Modul hat gewisse Eigenschaften für den Modulnamen und Konsequenzen in den Abhängigkeiten:

- Ohne Modulinfo haben die automatischen Module keinen selbstgewählten Namen, sondern sie bekommen vom System einen Namen zugewiesen, der sich aus dem Dateinamen ergibt.[4] Vereinfacht gesagt: Angehängte Versionsnummern und die Dateiendung werden entfernt und alle nicht-alphanummerischen Zeichen durch Punkte ersetzt, jedoch nicht zwei Punkte hintereinander.[5] Die Version wird erkannt. Die Dokumentation gibt das Beispiel *foo-bar-1.2.3-SNAPSHOT.jar* an, was zum Modulnamen `foo.bar` und der Version `1.2.3-SNAPSHOT` führt.
- Automatische Module exportieren immer alle ihre Pakete. Wenn es also eine Abhängigkeit zu diesem automatischen Modul gibt, kann der Bezieher alle sichtbare Typen und Eigenschaften verwenden.
- Automatische Module können alle anderen Module lesen, auch die unbenannten.

Auf den ersten Blick scheint eine Migration in Richtung Java 9 einfach: alle JARs auf den Modulpfad, und nacheinander Modulinfodateien anlegen. Allerdings gibt es JAR-Dateien, die von der JVM als automatisches Modul abgelehnt werden, wenn sie nämlich Typen eines Paketes enthalten und dieses Paket sich schon in einem anderen aufgenommen Modul befindet. Module dürfen keine »split packages« enthalten, also das gleiche Paket noch ein-

4 `Automatic-Module-Name` in die *META-INF*-Datei zu setzen, ist eine Alternative, dazu später mehr.

5 *http://download.java.net/java/jigsaw/docs/api/java/lang/module/ModuleFinder.html#of-java.nio.file.Path...-*.

mal enthalten. Die Migration erfordert dann entweder a) das Zusammenlegen der Pakete zu einem Modul, b) die Verschiebung in unterschiedliche Pakete oder c) die Nutzung des Klassenpfades.

1.2.8 Unbenanntes Modul

Eine Migration auf eine neue Java-Version sieht in der Regel so aus, dass zuerst die JVM gewechselt und geprüft wird, ob die vorhandene Software weiterhin funktioniert. Laufen die Testfälle durch und gibt es keine Auffälligkeiten im Testbetrieb, kann der Produktivbetrieb unter der neuen Version erfolgen. Gibt es keine Probleme, können nach einiger Zeit die neuen Sprachmittel und Bibliotheken verwendet werden.

Übertragen wir das auf den Wechsel von Java 8 auf Java 9: Eine vorhandene Java-Software muss inklusive aller Einstellungen und Einträge im Klassenpfad weiterhin laufen. Das heißt, eine Java 9-Laufzeitumgebung kann den Klassenpfad nicht ignorieren. Da es intern nur einen Modulpfad gibt, müssen auch diese JAR-Dateien zu Modulen werden. Die Lösung ist das *unbenannte Modul* (engl. *unnamed module*): Jedes JAR im Klassenpfad – dabei spielt es keine Rolle, ob es eine *modul-info.class* enthält – kommt in das unbenannte Modul. Davon gibt es nur eines, wir sprechen also im Singular, nicht Plural.

»Unbenannt« sagt schon, dass das Modul keinen Namen hat und folglich auch keine Abhängigkeit zu den JAR-Dateien im unbenannten Modul existieren kann; das ist der Unterschied zu einem automatischen Modul. Ein unbenanntes Modul hat die gleiche Eigenschaft wie ein automatisches Modul, dass es alle Pakete exportiert. Und weil es zur Migration gehört, hat ein unbenanntes Modul auch Zugriff auf alle anderen Module.

1.2.9 Lesbarkeit und Zugreifbarkeit

Die Laufzeitumgebung sortiert Module in einen Graphen ein. Die Abhängigkeit der Module führt dabei zur sogenannten *Lesbarkeit* (engl. *readability*): Benötigt Modul A Modul B, so *liest* A Modul B, und B wird von A *gelesen*. Für die Funktionsweise des Modulsystems ist dies elementar, denn so werden zur Übersetzungszeit schon Fehler ausgeschlossen, wie Zyklen oder gleiche Pakete in unterschiedlichen Modulen. Die Lesbarkeit ist zentral für eine zuverlässige Konfiguration, engl. *reliable configuration*.

Einen Schritt weiter geht der Begriff der *Erreichbarkeit*/Zugänglichkeit (engl. *accessibility*). Wenn ein Modul ein anderes Modul grundsätzlich lesen kann, bedeutet das noch nicht, dass es an alle Pakete und Typen kommt, denn nur diejenigen Typen sind sichtbar, die exportiert worden sind. Lesbare und erreichbare Typen nennen sich *erreichbar*.

Die nächste Frage ist, welcher Modultyp auf welchen anderen Modultyp Zugriff hat. Eine Tabelle fasst die Lesbarkeit am besten zusammen:

Modultyp	Ursprung	Exportiert Pakete	Hat Zugriff auf
Plattform-modul	JDK	Explizit	
Benannte Module	Container mit Modul-info im Modulpfad	Explizit	Plattformmodule, andere benannte Module, automatische Module
Automatische Module	Container ohne Modulinfo im Modul-pfad	Alle	Plattformmodule, andere benannte Module, automatische Module, unbenanntes Modul
Unbenanntes Modul	Klassendateien und JARs im Klassenpfad	Alle	Plattformmodule, benannte Module, automatische Module

Tabelle 1.2 Lesbarkeit der Module

Der Modulinfodatei kommt dabei die größte Bedeutung zu, denn sie macht aus einem JAR ein Modular JAR; fehlt die Modulinformation, bleibt es ein normales JAR, wie sie Java-Entwickler seit 20 Jahren kennen. Die JAR-Datei kann neu in den Modulpfad kommen oder in den bekannten Klassenpfad. Das ergibt vier Kombinationen:

	Modulpfad	Klassenpfad
JAR mit Modulinformation	Wird benanntes Modul.	Wird unbenanntes Modul.
JAR ohne Modulinformation	Wird automatisches Modul.	Wird unbenanntes Modul.

Tabelle 1.3 JARs im Pfad

JAR-Archive im Klassenpfad sind das bekannte Verhalten, weswegen auch ein Wechsel von Java 8 auf Java 9 möglich sein sollte.

1.2.10 Modul-Migration

Nehmen wir an, unsere monolithische Applikation hat keine Abhängigkeiten zu externen Bibliotheken und soll modularisiert werden. Dann besteht der erste Schritt darin, die gesamte Applikation in ein großes benanntes Modul zu setzen. Als Nächstes müssen die einzelnen Bereiche identifiziert werden, damit nach und nach die Bausteine in einzelne Module wandern. Das ist nicht immer einfach, zumal zyklische Abhängigkeiten nicht unwahrscheinlich sind. Bei der Modularisierung des JDK hatten die Oracle-Entwickler viel Mühe.

Das Problem mit automatischen Modulen

Traditionell generieren Build-Werkzeuge wie Maven oder Gradle JAR-Dateien, und ein Dateiname hat sich irgendwie ergeben. Werden jedoch diese JAR-Dateien zu automatischen Modulen, spielt der Dateiname plötzlich eine große Rolle. Doch bewusst wurde der Dateiname vermutlich nie gewählt. Referenziert ein benanntes Modul ein automatisches Modul, bringt das zwei Probleme mit sich: Ändert sich der Dateiname – lassen wir die Versionsnummer einmal außen vor –, heißt auch das automatische Modul anders, und die Abhängigkeit kann nicht mehr aufgelöst werden. Das zweite Problem ist größer: Viele Java-Bibliotheken haben noch keine Modulinformationen, und folglich werden Entwickler eine Abhängigkeit zu diesem automatischen Modul über den abgeleiteten Namen ausdrücken. Nehmen wir z. B. die beliebte Open-Source-Bibliothek Google Guava. Die JAR-Datei hat den Dateinamen *guava-23.0.jar* – `guava` heißt folglich das automatische Modul. Ein benanntes Modul (nennen wir es `M1`) kann über `required guava` eine Abhängigkeit ausdrücken. Konvertiert Google die Bibliothek in ein echtes Java 9-Modul, dann wird sich der Name ändern – geplant ist `com.google.guava`. Und ändert sich der Name, führen alle Referenzierungen in Projekten zu einem Compilerfehler; ein Alias wäre eine tolle Idee, das gibt es jedoch nicht. Und das Problem besteht ja nicht nur im eigenen Code, der Guava referenziert; referenziert das eigene Modul `M1` ein Modul `M2`, das wiederum Guava referenziert, so gibt es das gleiche Problem – wir sprechen von einer *transitiven Abhängigkeit*. Die Änderung des Modulnamens von Guava wird zum Problem, denn wir müssen warten, bis `M2` den Namen korrigiert, damit `M1` wieder gültig ist.

Eine Lösung mildert das Problem ab: In der JAR-Manifest-Datei kann ein Eintrag `Automatic-Module-Name` gesetzt werden – das »überschreibt« den automatischen Modulnamen.

Beispiel

Apache Commons setzt den Namen so:

```
Automatic-Module-Name: org.apache.commons.lang3
```

Benannte Module, die Abhängigkeiten zu automatischen Module besitzen, sind also ein Problem. Es ist zu hoffen, dass die zentralen Java-Bibliotheken, auf die sich so viele Lösungen stützen, schnell Modulinformationen einführen. Das wäre eine Lösung von unten nach oben, englisch *bottom-up*. Das ist das Einzige, was erfolgversprechend ist, aber wohl auch eine lange Zeit benötigen wird. Im Monat vom Java 9-Release hat noch keine wichtige Java-Bibliothek eine Modulinformation, `Automatic-Module-Name` kommt häufiger vor.

1.3 Sprachänderungen in Java 9

Eine Änderung in Java 9 ist klein, aber nützlich: Vor Java 9 mussten die in einem `try` mit Ressourcen verwendeten `AutoClosable`-Variablen immer deklariert werden. Das ist seit Java 9

nicht mehr nötig. In der neuen Syntax kann eine Variable stehen, die vorher deklariert ist. Die einzige Voraussetzung ist: Sie muss effektiv final sein.

Zur Implementierung innerer anonymer Klassen war bisher der Diamantoperator im Zusammenhang mit Generics nicht möglich. Das ist ab Java 9 erlaubt.

Eine etwas merkwürdige Änderung betrifft den Bezeichnernamen _ (Unterstrich). Er ist nun ein reserviertes Schlüsselwort und kann folglich nicht mehr als Bezeichner verwendet werden. Das betrifft einigen Programmcode, denn häufig nutzen Autoren den Unterstrich, um nicht benutzte Methodenparameter oder nicht benutzte Lambda-Parameter zu schreiben. Da der Compiler ab Version 8 jedoch schon vorgewarnt hat, blieb genug Zeit, den Unterstrich aus dem Quellcode zu verbannen.

Was in Java 8 geplant wurde, hat Oracle in Java 9 umgesetzt: Es gibt private Methoden in Schnittstellen. Somit kann doppelter Quellcode einfacher ausgelagert werden. Vor Java 9 musste doppelter Quellcode entweder in den Default-Methoden oder in statischen Methoden akzeptiert oder weitergeleitet werden an eine Hilfsmethode einer anderen Klasse.

Die Annotation `@SafeVarargs` ist nun auch an privaten Objektmethoden möglich.

Und wo wir schon einmal bei Annotationen sind – die Annotation `Deprecation` hat zwei neue Eigenschaften: `forRemoval()` und `since()`.

1.4 Bibliotheksänderungen in Java 9

In dem aktualisierten Band sind die Änderungen in Java 9 eingeflossen und auf die unterschiedlichen Kapitel verteilt. Die gesamte Liste der Änderungen findet sich unter *http://openjdk.java.net/projects/jdk9/*.

1.5 Änderungen in den Werkzeugen von Java 9

Die Standardausgabe des Werkzeugs Javadoc ist HTML5. Neu ist ebenfalls eine Suche in der API-Dokumentation. Sie ist nicht mit einem Webservice verbunden, sondern arbeitet lokal. Das ist gut, denn es muss keine Internetverbindung vorhanden sein. Die Suche arbeitet auf Teilzeichenfolgen von Bezeichnernamen und zeigt die Ergebnisse in einer Liste an. Das ist auch für Nutzer von Java 8 interessant, denn ist es ein deutlicher Mehrwert, und die Änderungen in der API sind unter Java 9 nicht so gravierend.

Das Java-Modulsystem beschreibt präzise, was ein Programm zur Compilezeit und zur Laufzeit benötigt. Das schließt bestimmte Pakete explizit ein, und so ist es möglich, auch zu bestimmen, was von den Kernmodulen nicht gebraucht wird. Mit diesem Wissen kann eine sehr kompakte individuelle Laufzeitumgebung zusammengestellt werden, die nur das integriert, was das eigene Programm referenziert. Das Werkzeug, das Oracle zum Zusammenbin-

den einer individuellen Laufzeitumgebung mit dem eigenen Programm mitbringt, heißt *jlink*: The Java Linker.

Ebenfalls ein neues Tool, das jedoch exklusiv für Linux-x64-Systeme ist, ist der »Java Ahead-Of-Time Compiler« *jaotc*. Die Dokumentation unter *http://openjdk.java.net/jeps/295* spricht deutliche Worte, wenn sie sagt: »experimental-only«.

1.6 JDK 9-HotSpot-JVM-Änderungen

Wenn es um Java geht, müssen wir immer unterscheiden, ob es um die Sprache selbst geht, um die Bibliotheken, um die JVM oder um die Implementierung von Oracle, die das JDK darstellt.

Auf der Seite der virtuellen Maschine ist die größte Änderung die Festlegung auf G1 für den Standard Garbage Collector. Er ist optimiert auf kurze Pausen anstatt auf hohen Durchsatz.

Die Java-Version für ARM 32/64 Bit war bis Java 8 nicht quelloffen. Das hat Oracle nachgeholt.

z Systems (auch bekannt unter den Namen zSeries, System z, z/Architecture) ist eine Mainframe-Architektur von IBM. Verschiedene Linux-Distributionen laufen auf s390x, unter anderem Ubuntu, RHEL/Fedora und SUSE-Linux-Produkte. Neu für Java 9 ist ein Linux/s390x-Port.

1.7 Zum Weiterlesen

Im Eclipse Project Oxygen.1a (4.7.1a) gibt es Unterstützung für Java 9. Die Webseiten *https://www.eclipse.org/eclipse/news/4.7.1a* und *https://wiki.eclipse.org/Java9/Examples* geben eine Übersicht.

Kapitel 4
Datenstrukturen und Algorithmen

»Überlege einmal, bevor du gibst, zweimal, bevor du annimmst, und tausendmal, bevor du verlangst.«
– Marie von Ebner-Eschenbach (1830–1916)

Algorithmen[1] sind ein zentrales Thema der Informatik. Ihre Erforschung und Untersuchung nimmt dort einen bedeutenden Platz ein. Algorithmen operieren nur dann effektiv mit Daten, wenn diese geeignet strukturiert sind. Schon das Beispiel Telefonbuch zeigt, wie wichtig die Ordnung der Daten nach einem Schema ist. Die Suche nach einer Telefonnummer bei gegebenem Namen gelingt schnell, während die Suche nach einem Namen bei bekannter Telefonnummer ein mühseliges Unterfangen darstellt. Datenstrukturen und Algorithmen sind also eng miteinander verbunden, und die Wahl der richtigen Datenstruktur entscheidet über effiziente Laufzeiten; beide erfüllen allein nie ihren Zweck. Leider ist die Wahl der »richtigen« Datenstruktur nicht so einfach, wie es sich anhört, und eine Reihe von schwierigen Problemen in der Informatik ist wohl deswegen noch nicht gelöst, weil eine passende Datenorganisation bis jetzt nicht gefunden wurde.

Die wichtigsten Datenstrukturen, wie Listen, Mengen, Kellerspeicher und Assoziativspeicher, sollen in diesem Kapitel vorgestellt werden. In der zweiten Hälfte des Kapitels wollen wir uns dann stärker den Algorithmen widmen, die auf diesen Datenstrukturen operieren.

4.1 Datenstrukturen und die Collection-API

Dynamische Datenstrukturen passen ihre Größe der Anzahl der Daten an, die sie aufnehmen. Schon in Java 1.0 brachte die Standardbibliothek fundamentale Datenstrukturen mit, aber erst mit Java 1.2 wurde mit der so genannten *Collection-API* der Umgang mit Datenstrukturen und Algorithmen auf eine gute Basis gestellt. In Java 5 gab es große Anpassungen durch Einführung der Generics und in Java 8 Anpassungen durch das neue Sprachkonstrukt der Lambda-Ausdrücke. Die Datenstrukturen finden sich allesamt im Java-Paket `java.util` respektive Unterpaket `java.util.concurrent`.

1 Das Wort *Algorithmus* geht auf den Namen des persisch-arabischen Mathematikers Ibn Mûsâ Al-Chwârismî zurück, der im 9. Jahrhundert lebte.

Sprachregelung

Ein Container ist ein Objekt, das wiederum andere Objekte aufnimmt und die Verantwortung für die Elemente übernimmt. Wir werden die Begriffe *Container*, *Sammlung* und *Collection* synonym[2] verwenden.

4.1.1 Designprinzip mit Schnittstellen, abstrakten und konkreten Klassen

Das Design der Collection-Klassen folgt vier Prinzipien:

- Schnittstellen legen Gruppen von Operationen für die verschiedenen Behältertypen fest. So gibt es zum Beispiel mit `List` eine Schnittstelle für Sequenzen (Listen) und mit `Map` eine Schnittstelle für Assoziativspeicher, die Schlüssel-Wert-Paare verbinden.
- Konkrete Klassen für bestimmte Behältertypen beerben die entsprechende abstrakte Basisklasse und ergänzen die unbedingt erforderlichen Grundoperationen (und ergänzen einige die Performance steigernde Methoden gegenüber der allgemeinen Lösung in der Oberklasse). Sie sind in der Nutzung unsere direkten Ansprechpartner. Für eine Liste können wir zum Beispiel die konkrete Klasse `ArrayList` und als Assoziativspeicher die Klasse `TreeMap` nutzen.
- Algorithmen, wie die Suche nach einem Element, gehören zum Teil zur Schnittstelle der Datenstrukturen. Zusätzlich gibt es mit der Klasse `Collections` eine Utility-Klasse mit weiteren Algorithmen.
- Weniger interessant für Anwender, aber für Implementierer einer Datenstruktur sind eine Vielzahl abstrakte Basisklassen, die die Operationen der Schnittstellen so weit wie möglich realisieren und auf eine minimale Zahl von als abstrakt deklarierten Grundoperationen zurückführen. So greift etwa `addAll(…)` auf mehrere `add(…)`-Aufrufe zurück oder `isEmpty()` auf `getSize()`. (Mit den abstrakten Basisimplementierungen wollen wir uns nicht weiter beschäftigen. Sie sind interessanter, wenn eigene Datenstrukturen auf der Basis der Grundimplementierung entworfen werden.)

4.1.2 Die Basisschnittstellen Collection und Map

Alle Datenstrukturen aus der Collection-API fußen auf zwei Schnittstellen:

- `java.util.Collection` (für Listen, Mengen, Schlangen)
- `java.util.Map` (für Assoziativspeicher)

Durch die gemeinsame Schnittstelle erhalten alle implementierenden Klassen einen gemeinsamen Rahmen. Die Operationen lassen sich grob einteilen in:

2 Wie war noch mal das andere Wort für synonym?

- Basisoperationen zum Erfragen der Elementanzahl und zum Hinzufügen, Löschen, Selektieren und Finden von Elementen
- Mengenoperationen, die etwa andere Sammlungen einfügen
- Array-Operationen bei `Collection`, um die Sammlung in ein Array zu konvertieren, und bei `Map` in Operationen, um alternative Ansichten von Schlüsseln oder Werten zu bekommen

4.1.3 Die Utility-Klassen Collections und Arrays

Datenstrukturen implementieren Algorithmen, etwa die Verwaltung von Elementen in einem Binärbaum oder einem Array. Java bietet zudem zwei besondere Klassen insbesondere für Such- und Sortieralgorithmen: Die Utility-Klasse `Collections` bietet Hilfsmethoden für `Collection`-Objekte – und einige wenige Methoden für Mengen –, und `Arrays` bietet statische Hilfsmethoden für eben Arrays. Wichtig ist, auf die Schreibweise zu achten: `Collection` versus `Collections`. Die Namensgebung ist jedoch einheitlich, denn in der gesamten Java-API gibt es mehrere Beispiele für Utility-Klassen, die ein »s« am Ende bekommen und ausschließlich statische Methoden bereitstellen. Einige Algorithmen sind auch direkt bei den Klassen zu finden, so etwa `sort(…)` bei Listen, aber auch `Collections` bietet eine `sort(…)`-Methode.

4.1.4 Das erste Programm mit Container-Klassen

Bis auf Assoziativspeicher implementieren alle Container-Klassen das Interface `Collection` und haben dadurch schon wichtige Methoden, die Daten aufnehmen, manipulieren und auslesen. Das folgende Programm erzeugt als Datenstruktur eine *verkettete Liste*, fügt Strings ein und gibt zum Schluss die Sammlung auf der Standardausgabe aus:

Listing 4.1 src/main/java/com/tutego/insel/util/MyFirstCollection.java, MyFirstCollection

```
public class MyFirstCollection {
  private static void fill( Collection<String> c ) {
    c.add( "Juvy" );
    c.add( "Tina" );
    c.add( "Joy" );
  }

  public static void main( String[] args ) {
    List<String> c = new LinkedList<>();
    fill( c );
    System.out.println( c );     // [Juvy, Tina, Joy]
    Collections.sort( c );
```

```
    System.out.println( c );     // [Joy, Juvy, Tina]
  }
}
```

Das Beispiel zeigt unterschiedliche Aspekte der Collection-API:

- Alle Datenstrukturen sind generisch deklariert. Statt `new LinkedList()` schreiben wir zum Beispiel `new LinkedList<String>()`, weil die Sammlung `String`-Objekte aufnehmen soll und durch die Generics typsicher wird.
- Unserer eigenen statischen Methode `fill(Collection<String>)` ist es egal, welche konkrete `Collection` wir ihr geben. Statt einer `LinkedList` hätten wir auch eine `ArrayList` übergeben können, denn auch sie ist vom Typ `Collection`. Das Gleiche gilt für Mengen (`Set`-Objekte), denn alle `Set`-Klassen implementieren ebenfalls `Collection`.
- Eine Liste lässt sich mit `add(…)` füllen. Die Methode schreibt die Schnittstelle `Collection` vor, und `LinkedList` realisiert die Operation aus der Schnittstelle.
- Während `Collection` eine Schnittstelle ist, die von unterschiedlichen Datenstrukturen implementiert wird, ist `Collections` eine Utility-Klasse mit vielen Hilfsmethoden, etwa zum Sortieren mit `Collections.sort(…)`.

Tipp

Nutze immer den kleinstnötigen Typ! Wir haben das an zwei Stellen getan. Statt `fill (LinkedList<String> c)` deklariert das Programm `fill(Collection<String> c)`, und statt `LinkedList<String> c = new LinkedList<>()` nutzt es `List<String> c = new LinkedList<>()`. Mit der Nutzung des Basistyps lassen sich unter softwaretechnischen Gesichtspunkten leicht die konkreten Datenstrukturen ändern, aber etwa die Methodensignatur ändert sich nicht und ist breiter aufgestellt. Es ist immer schön, wenn wir – etwa aus Gründen der Geschwindigkeit oder Speicherplatzbeschränkung – auf diese Weise leicht die Datenstruktur ändern können und der Rest des Programms unverändert bleibt. Das ist die Idee der schnittstellenorientierten Programmierung, und es ist in Java selten nötig, den konkreten Typ einer Klasse direkt anzugeben.

4.1.5 Die Schnittstelle Collection und Kernkonzepte

Unterschnittstellen erweitern `Collection` und schreiben Verhalten vor, ob etwa der Container die Reihenfolge des Einfügens beachtet, Werte doppelt beinhalten darf oder die Werte sortiert hält; `List`, `Set`, `Queue`, `Deque` und `NavigableSet` sind dabei die wichtigsten.

Es folgt eine Übersicht über alle Methoden:

```
interface java.util.Collection<E>
extends Iterable<E>
```


- `boolean add(E o)`
  Optional. Fügt dem Container ein Element hinzu und gibt `true` zurück, falls sich das Element einfügen lässt. Gibt `false` zurück, wenn schon ein Objekt gleichen Werts vorhanden ist und doppelte Werte nicht erlaubt sind. Diese Semantik gilt etwa bei Mengen. Erlaubt der Container das Hinzufügen grundsätzlich nicht, löst er eine `UnsupportedOperationException` aus.
- `boolean addAll(Collection<? extends E> c)`
  Fügt alle Elemente der Collection `c` dem Container hinzu.
- `void clear()`
  Optional. Löscht alle Elemente im Container. Wird dies vom Container nicht unterstützt, wird eine `UnsupportedOperationException` ausgelöst.
- `boolean contains(Object o)`
  Liefert `true`, falls der Container ein inhaltlich gleiches Element enthält.
- `boolean containsAll(Collection<?> c)`
  Liefert `true`, falls der Container alle Elemente der Collection `c` enthält.
- `boolean isEmpty()`
  Liefert `true`, falls der Container keine Elemente enthält.
- `Iterator<E> iterator()`
  Liefert ein `Iterator`-Objekt über alle Elemente des Containers.
- `boolean remove(Object o)`
  Optional. Entfernt das angegebene Objekt aus dem Container, falls es vorhanden ist.
- `boolean removeAll(Collection<?> c)`
  Optional. Entfernt alle Objekte der Collection `c` aus dem Container.
- `default boolean removeIf(Predicate<? super E> filter)`
  Entfernt alle Elemente aus dem Container, bei denen das Prädikat erfüllt ist. (Sollte eigentlich auch als optional gekennzeichnet sein, weil es intern auf das `remove()` vom Iterator zurückgreift.)
- `boolean retainAll(Collection<?> c)`
  Optional. Entfernt alle Objekte, die nicht in der Collection `c` vorkommen.
- `int size()`
  Gibt die Anzahl der Elemente im Container zurück.
- `Object[] toArray()`
  Gibt ein Array mit allen Elementen des Containers zurück.
- `<T> T[] toArray(T[] a)`
  Gibt ein Array mit allen Elementen des Containers zurück. Verwendet das als Argument übergebene Array als Ziel-Container, wenn es groß genug ist. Sonst wird ein Array passender Größe angelegt, dessen Laufzeittyp `a` entspricht.

- `boolean equals(Object o)`
  Prüft, a) ob das angegebene Objekt `o` ein kompatibler Container ist und b) ob alle Elemente aus dem eigenen Container `equals(…)`-gleich den Elementen des anderen Containers sind und c) ob sie – falls vorhanden – die gleiche Ordnung haben.
- `int hashCode()`
  Liefert den Hashwert des Containers. Dies ist wichtig, wenn der Container als Schlüssel in Assoziativspeichern verwendet wird. Dann darf der Inhal,t aber nicht mehr geändert werden, da der Hashwert von allen Elementen des Containers abhängt.

Hinweis

Der Basistyp `Collection` ist typisiert, genauso wie die Unterschnittstellen und implementierenden Klassen. Auffällig sind die Methoden `remove(Object)` und `contains(Object)`, die gerade nicht mit dem generischen Typ `E` versehen sind, was zur Konsequenz hat, dass diese Methoden mit beliebigen Objekten aufgerufen werden können. Fehler schleichen sich schnell ein, wenn der Typ der eingefügten Objekte ein anderer ist als der beim Löschversuch, etwa bei `HashSet<Long> set` mit anschließendem `set.add(1L)` und `remove(1)`.

Die Schnittstelle `Collection` erweitert die Schnittstelle `Iterable`, sodass jede Datenstruktur vom Typ `Collection` rechts vom Doppelpunkt eines erweiterten `for` stehen kann – mehr dazu gleich in Abschnitt 4.1.9, »Die Schnittstelle Iterable und das erweiterte for«.

Anzeige der Veränderungen durch boolesche Rückgaben

Der Rückgabewert einiger Methoden wie `addXXX(…)` oder `removeXXX(…)` ist ein `boolean` und könnte natürlich auch `void` sein. Doch die Collection-API signalisiert über die Rückgabe, ob eine Änderung der Datenstruktur erfolgte oder nicht. Bei Mengen liefert `add(…)` etwa `false`, wenn ein gleiches Element schon in der Menge ist; `add(…)` ersetzt das alte nicht durch das neue.

Optionale Methoden und UnsupportedOperationException

Einige Methoden aus der Schnittstelle `Collection` und Unterschnittstellen sind in der API-Dokumentation als *optional* notiert, weil konkrete Container oder Realisierungen die Operationen nicht realisieren wollen oder können. Da eine Schnittstellenimplementierung aber auf jeden Fall die Operation als Methode implementieren muss, lösen die Methoden eine `UnsupportedOperationException` aus, wenn sie die Operation nicht durchführen. Es gibt zum Beispiel Nur-Lese-Container, die Lesezugriffe zulassen, aber Veränderungen abblocken. Die Reaktion auf den Änderungsversuch zeigt folgendes Beispiel:

Listing 4.2 src/main/java/com/tutego/insel/util/UnsupportedOperationExceptionDemo.java, main()

```
Collection<Integer> set = new HashSet<>();
set.add( 1 );
Collection<Integer> c = Collections.unmodifiableCollection( set );
```



```
System.out.println( c );              // [1]
c.clear(); // Exception in thread "main" java.lang.UnsupportedOperationException
```

Die Optional-Anmerkungen erschrecken zunächst und lassen die Klassen bzw. Schnittstellen irgendwie unzuverlässig oder nutzlos erscheinen. Die konkreten Standardimplementierungen der Collection-API bieten diese Operationen jedoch vollständig an.

Das Konzept der optionalen Operationen ist umstritten, wenn Methoden zur Laufzeit eine Exception auslösen. Besser wären natürlich kleinere separate Schnittstellen, die nur die Leseoperationen enthalten und zur Übersetzungszeit überprüft werden können; dann gäbe es jedoch deutlich mehr Schnittstellen im `java.util`-Paket.

Vergleiche mit equals(…)

Der Methode `equals(…)` kommt bei den Elementen, die in die Datenstrukturen wandern, eine besondere Rolle zu. Jedes Objekt, das eine `ArrayList`, `LinkedList`, ein `HashSet` oder alle anderen Datenstrukturen[3] aufnehmen soll, muss zwingend `equals(…)` implementieren. Denn Methoden wie `contains(…)`, `remove(…)` vergleichen die Elemente mit `equals(…)` auf Gleichheit und nicht mit `==` auf Identität.

Beispiel

Ein neues Punkt-Objekt kommt in die Datenstruktur. Nun wird es mit einem anderen `equals(…)`-gleichen Objekt auf das Vorkommen in der `Collection` geprüft und gelöscht:

```
Collection<Point> list = new ArrayList<>();
list.add( new Point(47, 11) );
System.out.println( list.size() );                      // 1
System.out.println( list.contains( new Point(47, 11) ) ); // true
list.remove( new Point(47, 11) );
System.out.println( list.size() );                      // 0
```

Eigene Klassen müssen folglich `equals(…)` aus der absoluten Oberklasse `Object` überschreiben. Umgekehrt heißt das auch, dass Objekte, die kein sinnvolles `equals(…)` besitzen, nicht von den Datenstrukturen aufgenommen werden können; ein Beispiel hierfür ist `StringBuilder`/`StringBuffer`.

Besonderheit

Die Datenstrukturen selbst deklarieren eine `equals(…)`-Methode. Zwei Datenstrukturen sind `equals(…)`-gleich, wenn sie die gleichen Elemente – gleich nach der `equals(…)`-Relation – besitzen und die gleiche Ordnung haben. Ein Detail in der Implementierung überrascht jedoch. Exemplarisch:

3 Lassen wir die besondere Klasse `IdentityHashMap` außen vor.

```
LinkedList<String> l1 = new LinkedList<>( Arrays.asList( "" ) );
ArrayList<String>  l2 = new ArrayList<>( Arrays.asList( "" ) );
System.out.println( l1.equals( l2 ) );          // true
```

Die beiden Datenstrukturen sind gleich, obwohl ihre Typen unterschiedlich sind. Das ist einmalig in der Java-API. Dahinter steht, dass die `equals(…)`-Implementierung von etwa `ArrayList`, `LinkedList` nur betrachtet, ob das an `equals(…)` übergebene Objekt vom Typ `List` ist. Das Gleiche gilt im Übrigen auch für `Set` und `Map`. Nach dem Typtest folgen die Tests auf die Gleichheit der Elemente.

4.1.6 Schnittstellen, die Collection erweitern, und Map

Es gibt einige elementare Schnittstellen, die einen Container weiter untergliedern, etwa in der Art, wie Elemente gespeichert werden.

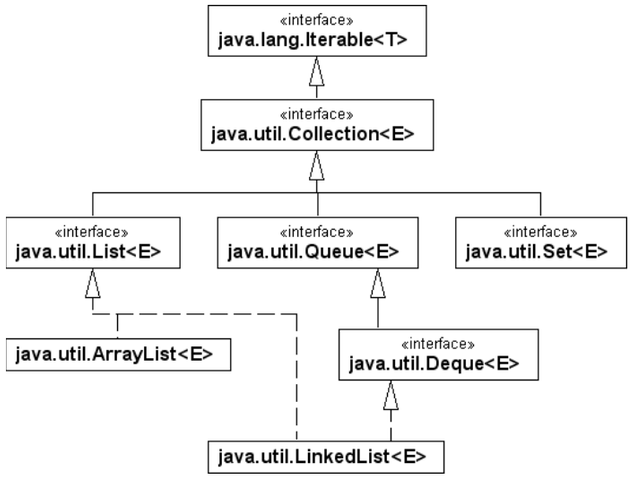

Abbildung 4.1 Zentrale Schnittstellen und Klassen der Collection-API

Die Schnittstelle List für Sequenzen

Die Schnittstelle `List`[4], die die `Collection`-Schnittstelle erweitert, enthält zusätzliche Operationen für eine *geordnete Liste* (auch *Sequenz* genannt) von Elementen. Auf die Elemente einer Liste lässt sich über einen ganzzahligen Index zugreifen, und es kann linear nach Elementen gesucht werden. Doppelte Elemente sind erlaubt, auch beliebig viele `null`-Einträge.

4 Wie in der Collection-Design-FAQ unter *https://docs.oracle.com/javase/9/docs/api/java/util/doc-files/coll-designfaq.html#a11* nachzulesen ist, hätte die Schnittstelle durchaus `Sequence` heißen können.

Zwei bekannte implementierende Klassen sind `LinkedList` sowie `ArrayList`. Weil das AWT-Paket eine Klasse mit dem Namen `List` deklariert, muss bei der `import`-Deklaration darauf geachtet werden, das richtige `java.util.List` statt `java.awt.List` zu verwenden.

Die Schnittstelle Set für Mengen

Ein `Set` ist eine im mathematischen Sinne definierte Menge von Objekten. Wie von mathematischen Mengen bekannt, darf ein `Set` keine doppelten Elemente enthalten. Für zwei nicht identische Elemente `e1` und `e2` eines `Set`-Objekts liefert der Vergleich `e1.equals(e2)` also immer `false`. Genauer gesagt: Aus `e1.equals(e2)` folgt, dass `e1` und `e2` identische Objektreferenzen sind, sich also auf dasselbe Mengenelement beziehen.

Besondere Beachtung muss Objekten geschenkt werden, die ihren Wert nachträglich ändern, da so zunächst ungleiche Mengenelemente inhaltlich gleich werden können. Dies kann ein `Set` nicht kontrollieren. Als weitere Einschränkung gilt, dass eine Menge sich selbst nicht als Element enthalten darf. Die wichtigste konkrete Mengen-Klasse ist `HashSet`.

`NavigableSet` – bzw. ihr Muttertyp `SortedSet` – erweitert `Set` um die Eigenschaft, Elemente sortiert auslesen zu können. Das Sortierkriterium wird durch ein Exemplar der Hilfsklasse `Comparator` bestimmt, oder die Elemente implementieren `Comparable`. Die Klassen `TreeSet` und `ConcurrentSkipListSet` implementieren die Schnittstellen und erlauben mit einem Iterator oder einer Array-Repräsentation Zugriff auf die sortierten Elemente.

Die Schnittstelle Queue für (Warte-)Schlangen

Eine Queue arbeitet nach dem FIFO-Prinzip (für engl. *first in, first out*); zuerst eingefügte Elemente werden zuerst wieder ausgegeben, getreu dem Motto: »Wer zuerst kommt, mahlt zuerst.« Die Schnittstelle `Queue` deklariert Operationen für alle Warteschlangen und wird etwa von den Klassen `LinkedList` und `PriorityQueue` implementiert.

Queue mit zwei Enden

Während die Queue Operationen bietet, die an einem Ende Daten anhängen und erfragen, bietet die Datenstruktur `Deque` (vom Englischen *Double-Ended QUEue*) Operationen an beiden Enden. Die Klasse `LinkedList` ist zum Beispiel eine Implementierung von `Deque`. Die Datenstruktur wird wie »Deck« ausgesprochen.

Die Schnittstelle Map

Eine Datenstruktur, die einen *Schlüssel* (engl. *key*) mit einem *Wert* (engl. *value*) verbindet, heißt *assoziativer Speicher*. Sie erinnert an ein Gedächtnis und ist mit einem Wörterbuch oder Nachschlagewerk vergleichbar. Betrachten wir ein Beispiel: Auf einem Personalausweis findet sich eine eindeutige Nummer, eine ID, die einmalig für jeden Bundesbürger ist. Wenn nun in einem Assoziativspeicher alle Passnummern gespeichert sind, lässt sich leicht über

die Passnummer (Schlüssel) die Person (Wert) herausfinden, also der Name der Person, die Gültigkeit des Ausweises usw. In die gleiche Richtung geht ein Beispiel, das ISB-Nummern mit Büchern verbindet. Ein Assoziativspeicher könnte zu der ISB-Nummer zum Beispiel das Erscheinungsjahr assoziieren, ein anderer Assoziativspeicher eine Liste von Rezensionen.

[+]

Hinweis

Gerne wird als Beispiel für einen Assoziativspeicher ein Telefonbuch mit einer Assoziation zwischen Namen und Telefonnummern genannt. Wenn das mit einem Assoziativspeicher realisiert werden muss, reicht natürlich der Name alleine nicht aus, sondern der Ort/das Land müssen dazukommen (ich bin zum Beispiel nicht der einzige Christian Ullenboom; in Erlangen wohnt mein Namensvetter). Auch ist es weniger ein Problem, dass in einem Familienhaushalt mehrere Personen die gleiche Telefonnummer besitzen. Vielmehr wird die Tatsache zum Problem, dass eine Person unterschiedliche Telefonnummern, etwa eine Mobil- und Festnetznummer, besitzen kann. Damit das Modell korrekt bleibt, muss eine Assoziation zwischen einem Namen und einer Liste von Telefonnummern bestehen. Ein Assoziativspeicher ist flexibel genug dafür: Der assoziierte Wert muss kein einfacher Wert wie eine Zahl oder String sein, sondern kann eine komplexe Datenstruktur sein.

In Java schreibt die Schnittstelle `Map` Verhalten für einen Assoziativspeicher vor. `Map` ist ein wenig anders als die anderen Schnittstellen. So erweitert die Schnittstelle `Map` die Schnittstelle `Collection` nicht. Das liegt daran, dass bei einem Assoziativspeicher Schlüssel und Wert immer zusammen vorkommen müssen und die Datenstruktur eine Operation wie `add(Object)` nicht unterstützen kann. Im Gegensatz zu `List` gibt es bei einer `Map` auch keine Position.

Die Schlüssel einer `Map` können mithilfe eines Kriteriums sortiert werden. Ist das der Fall, implementieren diese speziellen Klassen die Schnittstelle `NavigableMap` (bzw. den Muttertyp `SortedSet`), die `Map` direkt erweitert. Das Sortierkriterium wird entweder über ein externes `Comparator`-Objekt festgelegt, oder die Elemente in der `Map` sind vom Typ `Comparable`. Damit kann ein Iterator in einer definierten Reihenfolge einen assoziativen Speicher ablaufen. Bisher implementieren `TreeMap` und `ConcurrentSkipListMap` die Schnittstelle `NavigableMap`.

4.1.7 Konkrete Container-Klassen

Alle bisher vorgestellten Schnittstellen und Klassen dienen der Modellierung und dem Programmierer nur als Basistyp. Die Klassen in Tabelle 4.1 sind konkrete Klassen und können von uns benutzt werden.

Alle Datenstrukturen sind serialisierbar und implementieren `Serializable`, die Basisschnittstellen wie `Set`, `List` ... machen das nicht!

Typ	Implementierung	Erklärung
Listen (List)	`ArrayList`	Implementiert Listen-Funktionalität durch die Abbildung auf ein Array; implementiert die Schnittstelle `List`.
	`LinkedList`	`LinkedList` ist eine doppelt verkettete Liste, also eine Liste von Einträgen mit einer Referenz auf den jeweiligen Nachfolger und Vorgänger. Das ist nützlich beim Einfügen und Löschen von Elementen an beliebigen Stellen innerhalb der Liste.
Mengen (Set)	`HashSet`	Implementierung der Schnittstelle `Set` durch ein schnelles Hash-Verfahren
	`TreeSet`	Implementierung von `Set` durch einen Baum, der alle Elemente sortiert hält
	`LinkedHashSet`	schnelle Mengen-Implementierung, die sich parallel auch die Reihenfolge der eingefügten Elemente merkt
Assoziativspeicher (Map)	`HashMap`	Implementiert einen assoziativen Speicher durch ein Hash-Verfahren.
	`TreeMap`	Exemplare dieser Klasse halten ihre Elemente in einem Binärbaum sortiert; implementiert `NavigableMap`.
	`LinkedHashMap`	schneller Assoziativspeicher, der sich parallel auch die Reihenfolge der eingefügten Elemente merkt
	`WeakHashMap`	Verwaltet Elemente mit schwachen Referenzen, sodass die Laufzeitumgebung bei Speicherknappheit Elemente entfernen kann.
Schlange (Queue)	`LinkedList`	Die verkettete Liste implementiert `Queue` und auch `Deque`.
	`ArrayBlockingQueue`	blockierende Warteschlange
	`PriorityQueue`	Prioritätswarteschlange

Tabelle 4.1 Konkrete Container-Klassen

Welche Container-Klasse nehmen?

Bei der großen Anzahl von Klassen sind Entscheidungskriterien angebracht, nach denen Entwickler Klassen auswählen können. Die folgende Aufzählung soll einige Vorschläge geben:

- Ist eine Sequenz, also eine feste Ordnung, gefordert? Wenn ja, dann nimm eine Liste.
- Soll es einen schnellen Zugriff über einen Index geben? Wenn ja, ist die `ArrayList` gegenüber der `LinkedList` im Vorteil.
- Werden oft am Ende und Anfang Elemente eingefügt? Dann kann `LinkedList` punkten.
- Wenn eine Reihenfolge der Elemente uninteressant ist, aber schnell entschieden werden soll, ob ein Element Teil einer Menge ist, erweist sich `HashSet` als interessant.
- Sollen Elemente nur einmal vorkommen und immer sortiert bleiben? Dann ist `TreeSet` eine gute Wahl.
- Wenn es eine Assoziation zwischen Schlüssel und Elementen geben muss, ist eine `Map` von Vorteil.

4.1.8 Generische Datentypen in der Collection-API

Die Collection-API macht massiv Gebrauch von Generics. Das fällt unter anderem dadurch auf, dass die API-Dokumentation einen parametrisierten Typ erwähnt und die `Collection`-Schnittstelle zum Beispiel nicht `add(Object e)` deklariert, sondern `add(E e)`. Generics gewährleisten bessere Typsicherheit, da nur spezielle Objekte in die Datenstruktur kommen. Mit den Generics lässt sich bei der Konstruktion einer Collection-Datenstruktur angeben, welche Typen zum Beispiel in der Datenstruktur-Liste erlaubt sind. Soll eine Spielerliste `players` nur Objekte vom Typ `Player` aufnehmen, so sieht die Deklaration so aus:

```
List<Player> players = new ArrayList<>(); // bzw. ArrayList<Player>
```

Mit dieser Schreibweise lässt die Liste nur den Typ `Player` beim Hinzufügen und Anfragen zu, nicht aber andere Typen, wie etwa Zeichenketten. Das ist eine schöne Sicherheit für den Programmierer.

Geschachtelte Generics

Die Schreibweise `List<String>` deklariert eine Liste, die Strings enthält. Um eine verkettete Liste aufzubauen, deren Elemente wiederum Listen mit Strings sind, lassen sich die Deklarationen auch zusammenführen:[5]

```
List<List<String>> las = new LinkedList<>();
```

5 Das erinnert mich immer unangenehm an C: ein Feld von Pointern, die auf Strukturen zeigen, die Pointer enthalten.

4.1.9 Die Schnittstelle Iterable und das erweiterte for

Das erweiterte `for` erwartet rechts vom Doppelpunkt den Typ `java.lang.Iterable`, um durch eine Sammlung laufen zu können. Praktisch ist, dass alle `java.util.Collection`-Klassen die Schnittstelle `Iterable` implementieren, denn damit kann das erweiterte `for` leicht über diverse Sammlungen laufen.

Beispiel

Füge Zahlen in eine sortierte Menge ein, und gib sie aus:

```
Collection<Integer> numbers = new TreeSet<>();
numbers.add( 10 ); numbers.add( 2 ); numbers.add( 5 );
for ( Integer number : numbers )  // Alternativ: for ( int number : numbers )
  System.out.println( number );   // 2 5 10
```

Von der Datenstruktur übernimmt das erweiterte `for` den konkreten generischen Typ, hier `Integer`.

Ist die Sammlung nicht typisiert, wird die lokale Variable vom erweiterten `for` nicht den Typ bekommen können, sondern nur `Object`. Dann muss eine explizite Typumwandlung im Inneren der Schleife vorgenommen werden.

Hinweis

Ist die Datenstruktur `null`, so führt das zu einer `NullPointerException`:

```
Collection<String> list = null;
for ( String s : list )      // ☠ NullPointerException zur Laufzeit
  ;
```

Es wäre interessant, wenn Java dann die Schleife überspringen würde, aber der Grund für die Ausnahme ist, dass die Realisierung vom erweiterten `for` versucht, eine Methode vom `Iterable` aufzurufen, was natürlich bei `null` schiefgeht. Bei Arrays gilt übrigens das Gleiche, auch wenn hier keine Methode aufgerufen wird.

4.2 Listen

Eine Liste steht für eine Sequenz von Daten, bei der die Elemente eine feste Reihenfolge besitzen. Die Schnittstelle `java.util.List` schreibt Verhalten vor, die alle konkreten Listen implementieren müssen. Interessante Realisierungen der `List`-Schnittstelle sind:

- `java.util.ArrayList`: Liste auf der Basis eines Arrays
- `java.util.LinkedList`: Liste durch verkettete Elemente

- `java.util.concurrent.CopyOnWriteArrayList`: schnelle Liste, optimal für häufige nebenläufige Lesezugriffe
- `java.util.Vector`: synchronisierte Liste seit Java 1.0, die der `ArrayList` wich. Die Klasse ist zwar nicht deprecated, sollte aber nicht mehr verwendet werden.

Die Methoden zum Zugriff über die gemeinsame Schnittstelle `List` sind immer die gleichen. So ermöglicht jede Liste einen Punktzugriff über `get(index)`, und jede Liste kann alle gespeicherten Elemente sequenziell über einen Iterator geben. Doch die Realisierungen einer Liste unterscheiden sich in Eigenschaften wie der Performance, dem Speicherplatzbedarf oder der Möglichkeit der sicheren Nebenläufigkeit.

Da in allen Datenstrukturen jedes Exemplar einer von `Object` abgeleiteten Klasse Platz findet, sind die Listen grundsätzlich nicht auf bestimmte Datentypen fixiert, doch Generics spezifizieren diese Typen genauer.

4.2.1 Erstes Listen-Beispiel

Listen haben die wichtige Eigenschaft, dass sie sich die Reihenfolge der eingefügten Elemente merken und dass Elemente auch doppelt vorkommen können. Wir wollen diese Listenfähigkeit für ein kleines Gedächtnisspiel nutzen. Der Anwender gibt Städte für eine Route vor, die sich das Programm in einer Liste merkt. Nach der Eingabe eines neuen Ziels auf der Route soll der Anwender alle Städte in der richtigen Reihenfolge wiedergeben. Hat er das geschafft, kommt eine neue Stadt hinzu. Im Prinzip ist das Spiel unendlich, doch da sich kein Mensch unendlich viele Städte in der Reihenfolge merken kann, wird es zu einer Falscheingabe kommen, was das Programm beendet.

Listing 4.3 src/main/java/com/tutego/insel/util/list/HowDoesYourRouteLooksLike.java

```
package com.tutego.insel.util.list;

import java.text.*;
import java.util.*;

public class HowDoesYourRouteLooksLike {
  @SuppressWarnings( "resource" )
  public static void main( String[] args ) {
    List<String> cities = new ArrayList<>();

    while ( true ) {
      System.out.println( "Welche neue Stadt kommt hinzu?" );
      String newCity = new Scanner( System.in ).nextLine();
      cities.add( newCity );
```



```
      System.out.printf( "Wie sieht die gesamte Route aus? (Tipp: %d %s)%n",
                         cities.size(), cities.size() == 1 ? "Stadt" : "Städte" );

      for ( String city : cities ) {
        String guess = new Scanner( System.in ).nextLine();
        if ( ! city.equalsIgnoreCase( guess ) ) {
          System.out.printf( "%s ist nicht richtig, %s wäre korrekt. Schade!%n",
                             guess, city );
          return;
        }
      }
      System.out.println( "Prima, alle Städte in der richtigen Reihenfolge!" );
    }
  }
}
```

4.2.2 Auswahlkriterium ArrayList oder LinkedList

Eine `ArrayList` (das Gleiche gilt für `Vector`) speichert Elemente in einem internen Array. `LinkedList` dagegen speichert die Elemente in einer verketteten Liste und realisiert die Verkettung mit einem eigenen Hilfsobjekt für jedes Listenelement. Es ergeben sich Einsatzgebiete, die einmal für `LinkedList` und einmal für `ArrayList` sprechen:

- Da `ArrayList` intern ein Array benutzt, ist der Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell. Eine `LinkedList` muss aufwändiger durchsucht werden, und dies kostet Zeit.
- Die verkettete Liste ist aber deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden; hier muss einfach nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden. Bei einer `ArrayList` bedeutet dies viel Arbeit, es sei denn, das Element kann am Ende gelöscht oder – bei ausreichender Puffergröße – eingefügt werden. Soll ein Element nicht am Ende eingefügt oder gelöscht werden, müssen alle nachfolgenden Listenelemente verschoben werden.
- Bei einer `ArrayList` kann die Größe des internen Arrays zu klein werden. Dann bleibt der Laufzeitumgebung nichts anderes übrig, als ein neues, größeres Array-Objekt anzulegen und alle Elemente zu kopieren.

4.2.3 Die Schnittstelle List

Die Schnittstelle `List` schreibt das allgemeine Verhalten für Listen vor. Die allermeisten Methoden kennen wir schon vom `Collection`-Interface, und zwar deshalb, weil `List` die Schnittstelle `Collection` erweitert. Hinzugekommen sind Methoden, die einen Bezug zur Position eines Elements haben – Mengen, die auch `Collection` implementieren, kennen keinen Index.

Hinzufügen und Setzen von Elementen

Die `add(…)`-Methode fügt neue Elemente an die Liste an, wobei eine Position die Einfügestellen bestimmen kann. Die Methode `addAll(…)` fügt fremde Elemente einer anderen Sammlung in die Liste ein. `set(…)` setzt ein Element an eine bestimmte Stelle, überschreibt das ursprüngliche Element und verschiebt es auch nicht wie `add(…)`. Die Methode `size()` nennt die Anzahl der Elemente in der Datenstruktur:

Listing 4.4 src/main/java/com/tutego/insel/util/list/ListDemo.java. main()

```
List<String> list1 = new ArrayList<>();
list1.add( "Eva" );
list1.add( 0, "Charisma" );
list1.add( "Pallas" );

List<String> list2 = Arrays.asList( "Tina", "Wilhelmine" );
list1.addAll( 3, list2 );
list1.add( "XXX" );
list1.set( 5, "Eva" );

System.out.println( list1 );        // [Charisma, Eva, Pallas, Tina, Wilhelmine, Eva]
System.out.println( list1.size() ); // 6
```

Positionsanfragen und Suchen

Ob die Sammlung leer ist, bestimmt `isEmtpy()`. Ein Element an einer speziellen Stelle erfragen kann `get(int)`. Ob Elemente Teil der Sammlung sind, beantworten `contains(…)` und `containsAll(…)`. Wie bei Strings liefern `indexOf(…)` und `lastIndexOf(…)` die Fundpositionen:

```
boolean b = list1.contains( "Tina" );
System.out.println( b );            // true

b = list1.containsAll( Arrays.asList( "Tina", "Eva" ) );
System.out.println( b );            // true

Object o = list1.get( 1 );
System.out.println( o );            // Eva

int i = list1.indexOf( "Eva" );
System.out.println( i );            // 1

i = list1.lastIndexOf( "Eva" );
System.out.println( i );            // 5

System.out.println( list1.isEmpty() ); // false
```


Listen zu Arrays und neue Listen bilden

Von den Listen können Arrays abgeleitet werden und sich Schnittmengen bilden lassen:

```
String[] array = list1.toArray( new String[list1.size()] );
System.out.println( array[3] );     // "Tina"

List<String> list3 = new LinkedList<>( list1 );
System.out.println( list3 );        // [Charisma, Eva, Pallas, Tina,
                                    // Wilhelmine, Eva]
list3.retainAll( Arrays.asList( "Tina", "Eva" ) );
System.out.println( list3 );        // [Eva, Tina, Eva]
```

Löschen von Elementen

Außerdem gibt es Methoden zum Löschen von Elementen. Hier bietet die Liste eine überladene `remove(…)`-Methode, `removeIf(…)` und `removeAll(…)`. Den kürzesten Weg, alles aus der Liste zu löschen, bietet `clear()`:

```
System.out.println( list1 );     // [Charisma, Eva, Pallas, Tina, Wilhelmine, Eva]
list1.remove( 1 );
System.out.println( list1 );     // [Charisma, Pallas, Tina, Wilhelmine, Eva]

list1.remove( "Wilhelmine" );
list1.remove( "Wilhelmine" );
System.out.println( list1 );     // [Charisma, Pallas, Tina, Eva]

list1.removeAll( Arrays.asList( "Pallas", "Eva" ) );
System.out.println( list1 );     // [Charisma, Tina]

list1.clear();
System.out.println( list1 );     // []
```

Hinweis

Die Methode `remove(int)` löscht ein Element an der gegebenen Stelle, `remove(Object)` sucht einmal nach einem `equals`-gleichen Objekt und löscht es dann, würde aber nicht weitersuchen nach anderen Vorkommen. `removeIf(…)` und `removeAll(…)` laufen immer komplett über die ganze Datenstruktur und schauen, ob ein Element dem Kriterium genügt, um es zu löschen, was mehrmals vorkommen kann.

Beispiel:

Lösche alle `null`-Referenzen und Weißraum-Strings aus der Liste:

```
List<String> list = new ArrayList<>();
Collections.addAll( list, "1", "", " ", "zwei", null, "Polizei" );
list.removeIf( e -> Objects.isNull( e ) || e.trim().isEmpty() );
System.out.println( list ); // [1, zwei, Polizei]
```

Zusammenfassung

Die Methoden der Schnittstelle `List` (inklusive der aus der erweiterten Schnittstelle `Collection`) sind:

```
interface java.util.List<E>
extends Collection<E>
```

- `boolean add(E o)`
  Fügt das Element am Ende der Liste an. Eine optionale Operation.
- `void add(int index, E element)`
  Fügt ein Objekt an der angegebenen Stelle in die Liste ein. Eine optionale Operation.
- `boolean addAll(int index, Collection<? extends E> c)`
  Fügt alle Elemente der Collection an der angegebenen Stelle in die Liste ein. Eine optionale Operation.
- `void clear()`
  Löscht alle Elemente aus der Liste. Eine optionale Operation.
- `boolean contains(Object o)`
  Liefert `true`, wenn das Element `o` in der Liste ist. Den Vergleich übernimmt `equals(…)`, und es ist kein Referenz-Vergleich.
- `boolean containsAll(Collection<?> c)`
  Liefert `true`, wenn alle Elemente der Sammlung `c` in der aktuellen Liste sind.
- `E get(int index)`
  Wird das Element an dieser angegebenen Stelle der Liste liefern.
- `int indexOf(Object o)`
  Liefert die Position des ersten Vorkommens für `o` oder –1, wenn kein Listenelement mit `o` inhaltlich – also per `equals(…)` und nicht per Referenz – übereinstimmt. Leider gibt es keine Methode, die ab einer bestimmten Stelle weitersucht, so wie sie die Klasse `String` bietet. Dafür lässt sich jedoch eine Teilliste einsetzen, die `subList(…)` bildet – eine Methode, die später in der Aufzählung folgt.
- `boolean isEmpty()`
  Liefert `true`, wenn die Liste leer ist.
- `Iterator<E> iterator()`
  Liefert den Iterator. Die Methode ruft aber `listIterator()` auf und gibt ein `ListIterator`-Objekt zurück.
- `int lastIndexOf(Object o)`
  Sucht von hinten in der Liste nach dem ersten Vorkommen von `o` und liefert –1, wenn kein Listenelement inhaltlich mit `o` übereinstimmt.

- `ListIterator<E> listIterator()`
  Liefert einen Listen-Iterator für die ganze Liste. Ein Listen-Iterator bietet gegenüber dem allgemeinen Iterator für Container zusätzliche Operationen.
- `ListIterator<E> listIterator(int index)`
  Liefert einen Listen-Iterator, der die Liste ab der Position `index` durchläuft.
- `E remove(int index)`
  Entfernt das Element an der Position `index` aus der Liste.
- `boolean remove(Object o)`
  Entfernt das erste Objekt in der Liste, das `equals(…)`-gleich mit `o` ist. Liefert `true`, wenn ein Element entfernt wurde. Eine optionale Operation.
- `boolean removeAll(Collection<?> c)`
  Löscht in der eigenen Liste die Elemente aus `c`. Eine optionale Operation.
- `default boolean removeIf(Predicate<? super E> filter)`
  Entfernt alle Elemente aus der Liste, bei denen das Prädikat erfüllt ist.
- `boolean retainAll(Collection<?> c)`
  Optional. Entfernt alle Objekte aus der Liste, die nicht in der Collection `c` vorkommen.
- `default void replaceAll(UnaryOperator<E> operator)`
  Ruft auf jedem Element den Operator auf und schreibt das Ergebnis zurück.
- `E set(int index, E element)`
  Ersetzt das Element an der Stelle `index` durch `element`. Eine optionale Operation.
- `List<E> subList(int fromIndex, int toIndex)`
  Liefert den Ausschnitt dieser Liste von Position `fromIndex` (einschließlich) bis `toIndex` (nicht mit dabei). Die zurückgelieferte Liste stellt eine Ansicht eines Ausschnitts der Originalliste dar. Änderungen an der Teilliste wirken sich auf die ganze Liste aus und umgekehrt (soweit sie den passenden Ausschnitt betreffen).
- `default void sort(Comparator<? super E> c)`
  Sortiert die Liste, entspricht `Collections.sort(dieseListe, c)`.
- `boolean equals(Object o)`
  Vergleicht die Liste mit einer anderen Liste. Zwei Listenobjekte sind gleich, wenn ihre Elemente paarweise gleich sind.
- `int hashCode()`
  Liefert den Hashwert der Liste.

Was `List` der `Collection` hinzufügt, sind also die indexbasierten Methoden `add(int index, E element)`, `addAll(int index, Collection<? extends E> c)`, `get(int index)`, `indexOf(Object o)`, `lastIndexOf(Object o)`, `listIterator()`, `listIterator(int index)`, `remove(int index)`, `set(int`

index, E element) und subList(int fromIndex, int toIndex). Zudem die beiden Default-Methoden replaceAll(…) und sort(…).

Hinweis

Die remove(…)-Methode ist überschrieben und löscht a) einmal ein Element, das mit equals(…) gesucht wird, und b) mit remove(int) ein Element an der gegebenen Position. Irritierend ist Folgendes:

```
List<Integer> list = new ArrayList<>( Arrays.asList( 9, 8, 1, 7 ) );
Integer index = 1;
list.remove( index );
System.out.println( list );  // [9, 8, 7]
```

Es wird remove(Object) aufgerufen, weil Object dem Argumenttyp Integer am ähnlichsten ist. Somit verschwindet das Element Integer.valueOf(1) aus der Liste. Unboxing findet nicht statt. Das dürfte zu erwarten sein, und daher ist die Namensgebung index im Beispiel bewusst irreführend.

Kopieren und Ausschneiden

Die Listen-Klassen implementieren clone() und erzeugen eine flache Kopie.

Um einen Bereich zu löschen, nutzen wir subList(from, to).clear(). Die subList-Technik deckt gleich noch einige andere Operationen ab, für die es keine speziellen Range-Varianten gibt, zum Beispiel indexOf(…), also die Suche in einem Teil der Liste.

Beispiel

Baue eine Liste auf, kürze sie, und gib die Elemente rückwärts aus:

```
List<String> list = new ArrayList<>(
    Arrays.asList( "0 1 2 3 4 5 6 7 8 9".split( " " ) ) );
list.subList( 2, list.size() - 2 ).clear();
System.out.println( list );                  // [0, 1, 8, 9]
for ( ListIterator<String> it = list.listIterator( list.size() );
      it.hasPrevious(); )
  System.out.print( it.previous() + " " );    // 9 8 1 0
```

subList(…) erzeugt wie viele andere Methoden der Collection-Datenstrukturen eine Ansicht auf die Liste, was bedeutet, dass Änderungen an dieser Teilliste zu Änderungen des Originals führen. Das gilt auch für clear(), das dazu genutzt werden kann, einen Teilbereich der Originalliste zu löschen.

Hinweis

Die zum Löschen naheliegende Methode removeRange(int, int) kann nicht (direkt) eingesetzt werden, da sie protected[6] ist. Das lässt sich zum Beispiel wie folgt beheben:

```
List<gewünschterTyp> list = new ArrayList<gewünschterTyp>() {
  @Override public void removeRange( int fromIndex, int toIndex ) {
    super.removeRange( fromIndex, toIndex );
  }
};
```


4.2.4 ArrayList

Jedes Exemplar der Klasse ArrayList vertritt ein Array mit variabler Länge. Der Zugriff auf die Elemente erfolgt effizient über Indizes, was ArrayList über die Implementierung der Markierungsschnittstelle RandomAccess andeutet.

Eine ArrayList erzeugen

Um ein ArrayList-Objekt zu erzeugen, existieren drei Konstruktoren:

```
class java.util.ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

- ArrayList()
  Eine leere Liste mit einer Anfangskapazität von zehn Elementen wird angelegt. Werden mehr als zehn Elemente eingefügt, muss sich die Liste vergrößern.
- ArrayList(int initialCapacity)
  Eine Liste mit der internen Größe von initialCapacity-vielen Elementen wird angelegt.
- ArrayList(Collection<? extends E> c)
  Kopiert alle Elemente der Collection c in das neue ArrayList-Objekt.

Die interne Arbeitsweise von ArrayList und Vector *

Die Klassen ArrayList und Vector verwalten zwei Größen: zum einen die Anzahl der gespeicherten Elemente nach außen, zum anderen die interne Größe des Arrays. Ist die Kapazität des Arrays größer als die Anzahl der Elemente, so können noch Elemente aufgenommen werden, ohne dass die Liste etwas unternehmen muss. Die Anzahl der Elemente in der Liste,

6 In AbstractList ist removeRange(int, int) gültig mit einem ListIterator implementiert, also nicht abstrakt. Die API-Dokumentation begründet das damit, dass removeRange(…) nicht zur offiziellen Schnittstelle von Listen gehört, sondern für die Autoren neuer Listenimplementierungen gedacht ist.

die Größe, liefert die Methode `size()`; die Kapazität des darunterliegenden Arrays liefert `capacity()`.

Die Liste vergrößert sich automatisch, falls mehr Elemente aufgenommen werden, als ursprünglich am Platz vorgesehen waren. Diese Operation heißt *Resizing*. Dabei spielt die Größe `initialCapacity` für effizientes Arbeiten eine wichtige Rolle. Sie sollte passend gewählt sein. Betrachten wir daher zunächst die Funktionsweise der Liste, falls das interne Array zu klein ist.

Wenn das Array zehn Elemente fasst, nun aber ein elftes eingefügt werden soll, so muss das Laufzeitsystem einen neuen Speicherbereich reservieren und jedes Element des alten Arrays in das neue kopieren. Das kostet Zeit. Schon aus diesem Grund sollte der Konstruktor `ArrayList(int initialCapacity)`/`Vector(int initialCapacity)` gewählt werden, weil dieser eine Initialgröße festsetzt. Das Wissen über unsere Daten hilft dann der Datenstruktur. Falls kein Wert voreingestellt wurde, so werden zehn Elemente angenommen. In vielen Fällen ist dieser Wert zu klein.

Nun haben wir zwar darüber gesprochen, dass ein neues Array angelegt wird und die Elemente kopiert werden, wir haben aber nichts über die Größe des neuen Arrays gesagt. Hier gibt es Strategien wie die »Verdopplungsmethode« beim `Vector`. Wird er vergrößert, so ist das neue Array doppelt so groß wie das alte. Dies ist eine Vorgehensweise, die für kleine und schnell wachsende Arrays eine clevere Lösung darstellt, großen Arrays jedoch schnell zum Verhängnis werden kann. Für den Fall, dass wir die Vergrößerung selbst bestimmen wollen, nutzen wir den Konstruktor `Vector(int initialCapacity, int capacityIncrement)`, der die Verdopplung ausschaltet und eine fixe Vergrößerung befiehlt. Die `ArrayList` verdoppelt nicht, sie nimmt die neue Größe mal 1,5. Bei ihr gibt es nicht das `capacityIncrement` im Konstruktor.

Die Größe eines Arrays *

Die interne Größe des Arrays kann mit `ensureCapacity(int)` geändert werden. Ein Aufruf von `ensureCapacity(int minimumCapacity)` bewirkt, dass die Liste insgesamt mindestens `minimumCapacity` Elemente aufnehmen kann, ohne dass ein Resizing nötig wird. Ist die aktuelle Kapazität der Liste kleiner als `minimumCapacity`, so wird mehr Speicher angefordert. Der Vektor verkleinert die aktuelle Kapazität nicht, falls sie schon höher als `minimumCapacity` ist. Um aber auch diese Größe zu ändern und somit ein nicht mehr wachsendes Vektor-Array so groß wie nötig zu machen, gibt es, ähnlich wie beim String mit Weißraum, die Methode `trimToSize()`. Sie reduziert die Kapazität des Vektors auf die Anzahl der Elemente, die gerade in der Liste sind. Mit `size()` lässt sich die Anzahl der Elemente in der Liste erfragen.

Bei der Klasse `Vector` lässt sich mit `setSize(int newSize)` auch die Größe der Liste verändern. Ist die neue Größe kleiner als die alte, werden die Elemente am Ende des Vektors abgeschnitten. Ist `newSize` größer als die alte Größe, werden die neu angelegten Elemente mit `null` initialisiert.[7] Vorsicht ist bei `newSize=0` geboten, denn `setSize(0)` bewirkt das Gleiche wie `removeAllElements()`.

7 Zudem können `null`-Referenzen ganz normal als Elemente eines Vektors auftreten, bei den anderen Datenstrukturen gibt es Einschränkungen.

4.2.5 LinkedList

Die Klasse `LinkedList` realisiert die Schnittstelle `List` als verkettete Liste und bildet die Elemente nicht auf ein Array ab. Die Implementierung realisiert die `LinkedList` als doppelt verkettete Liste, in der jedes Element – die Ränder lassen wir außen vor – einen Vorgänger und Nachfolger hat. (Einfach verkettete Listen haben nur einen Nachfolger, was die Navigation in beide Richtungen schwierig macht.)

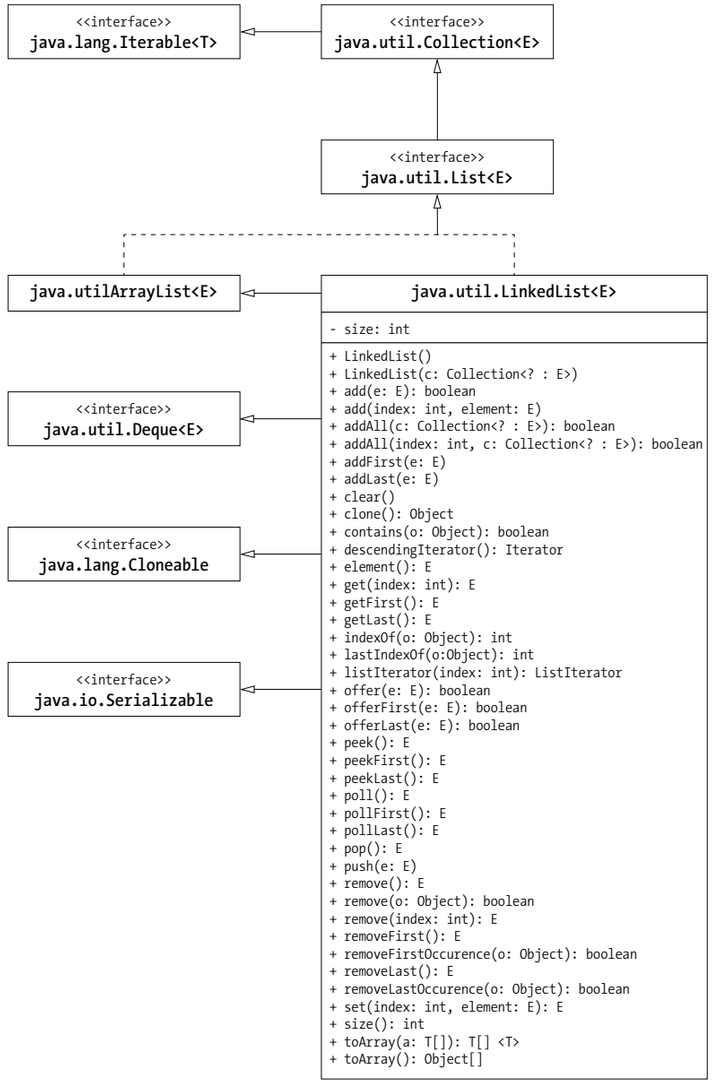

Abbildung 4.2 Klassendiagramm von LinkedList mit Vererbungsbeziehungen

Eine `LinkedList` hat neben den gegebenen Operationen aus der Schnittstelle `List` weitere Hilfsmethoden: Dabei handelt es sich um die Methoden `addFirst(…)`, `addLast(…)`, `getFirst()`, `getLast()`, `removeFirst()` und `removeLast()`. Die implementierten Schnittstellen `Queue` und `Deque` sind nicht ganz unschuldig an diesen Methoden.

```
class java.util.LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

- `LinkedList()`
  Erzeugt eine neue leere Liste.
- `LinkedList(Collection<? extends E> c)`
  Kopiert alle Elemente der Collection `c` in die neue verkettete Liste.

4.2.6 Der Array-Adapter Arrays.asList(...)

Arrays von Objektreferenzen und dynamische Datenstrukturen passen nicht so richtig zusammen, obwohl sie schon häufiger zusammen benötigt werden. Die Java-Bibliothek bietet mit der statischen Methode `Arrays.asList(…)` an, ein existierendes Array als `java.util.List` zu behandeln. Der Parametertyp ist ein Vararg – das ja intern auf ein Array abgebildet wird –, sodass sich `asList(…)` auf zwei Arten verwenden lässt:

- `Arrays.asList(array)`: Die Variable `array` ist eine Referenz auf ein Array, und das Ergebnis ist eine Liste, die die gleichen Elemente wie das Array enthält.
- `Arrays.asList(e1, e2, e3)`: Die Elemente `e1`, `e2`, `e3` sind Elemente der Liste.

Das Entwurfsmuster, das die Java-Bibliothek bei der statischen Methode anwendet, nennt sich *Adapter*. Er passt die Schnittstelle eines Typs an eine andere Schnittstelle eines anderen Typs an.

Beispiel

Ermittle die Anzahl der »:-)«-Smileys im String.

```
String s = "Oten :-) Bilat :-) Iyot";
int i = Collections.frequency( Arrays.asList(s.split("\\s")), ":-)" );
System.out.println( i );    // 2
```

In `String` gibt es keine Methode zum Zählen von Teil-Strings.

Beispiel

Gib das größte Element eines Arrays aus:

```
Integer[] ints = { 3, 9, -1, 0 };
System.out.println( Collections.max( Arrays.asList( ints ) ) );
```


Zum Ermitteln des Maximums bietet die Utility-Klasse `Arrays` keine Methode, daher bietet sich die `max(…)`-Methode von `Collections` an. Auch etwa zum Ersetzen von Array-Elementen bietet `Arrays` nichts, aber `Collections`. Sortieren und füllen kann `Arrays` aber schon, hier muss `asList()` nicht einspringen.

```
class java.util.Arrays
```

- `public static <T> List<T> asList(T... a)`
  Ermöglicht es, über die Schnittstelle `List` Zugriff auf ein Array zu erhalten. Die variablen Argumente sind sehr praktisch.

Hinweis

Wegen der Generics ist der Parametertyp von `asList(…)` ein Objekt-Array, aber niemals ein primitives Array. In unserem Beispiel von eben würde so etwas wie

```
int[] ints = { 3, 9, -1, 0 };
Arrays.asList( ints );
```

zwar compilieren, aber die Rückgabe von `Arrays.asList(ints)` ist vom Typ `List<int[]>`, was bedeutet, die gesamte Liste besteht aus genau einem Element, und dieses Element ist das primitive Array. Zum Glück führt `Collections.max(Arrays.asList(ints))` zu einem Compilerfehler, denn von einer `List<int[]>`, also einer Liste von Arrays, kann `max(Collection<? extends T>)` kein Maximum ermitteln. Anders wäre das bei `Arrays.asList(3, 9, -1, 0)`, denn hier konvertiert der Compiler die Varargs-Argumente über Autoboxing schon direkt in Wrapper-Objekte, und es kommt eine Liste von `Integer`-Objekten heraus.

Internes

Die Rückgabe von `asList(…)` ist kein konkreter Klassentyp wie `ArrayList` oder `LinkedList`, sondern irgendetwas Unbekanntes, was `asList(…)` als `List` herausgibt. Diese Liste ist nur eine andere Ansicht des Arrays.

Jetzt gibt es allerdings einen Interpretationsspielraum, was genau mit der Rückgabe möglich ist. Zudem ist es nicht ganz uninteressant, zu wissen, ob die Liste einen schnellen Punktzugriff zulässt (`RandomAccess` implementiert) bzw. ob optionale Operationen wie Veränderungen oder sogar totale Reorganisationen denkbar sind. Ein Blick auf die Implementierung verrät mehr. Das Ergebnis ist ein Adapter, der Listen-Methoden wie `get(index)` oder `set(index, element)` direkt auf das Array umleitet. Da Array-Längen final sind, führen Modifikationsmethoden wie `remove(…)` oder `add(…)` zu einer `UnsupportedOperationException`.

Hinweis

Sind Veränderungen an der `asList(…)`-Rückgabe erwünscht, so muss das Ergebnis in eine neue Datenstruktur kopiert werden, etwa so:

```
List<String> list = new ArrayList<>( Arrays.asList( "A", "B" ) );
list.add( "C" );
```

4.2.7 ListIterator *

Die Schnittstelle `ListIterator` ist eine Erweiterung von `Iterator`. Diese Schnittstelle fügt noch Methoden hinzu, damit an der aktuellen Stelle auch Elemente eingefügt werden können. Mit einem `ListIterator` lässt sich rückwärts laufen und auf das vorhergehende Element zugreifen:

Listing 4.5 src/main/java/com/tutego/insel/util/list/ListIteratorDemo.java, main()

```
List<String> list = new ArrayList<>();
Collections.addAll( list, "b", "c", "d" );

ListIterator<String> it = list.listIterator();

it.add( "a" );                    // Vorne anfügen
System.out.println( list );       // [a, b, c, d]

it.next();                        // Position vor
it.remove();                      // Element löschen
System.out.println( list );       // [a, c, d]

it.next();                        // Position vor
it.set( "C" );                    // Element ersetzen
System.out.println( list );       // [a, C, d]

it = list.listIterator( 1 );      // Neuen Iterator mit Startpos. 1
it.add( "B" );                    // B hinzufügen
System.out.println( list );       // [a, B, C, d]

it = list.listIterator( list.size() );

it.previous();                    // Eine Stelle nach vorne
it.remove();                      // Letztes Element löschen
System.out.println( list );       // [a, B, C]
```


Tipp

Ein `ListIterator` kann die Elemente auch rückwärts verarbeiten:

```
List<String> list = new ArrayList<String>();
Collections.addAll( list, "1", "2", "3", "4" );
for ( ListIterator<String> it = list.listIterator( list.size() );
      it.hasPrevious(); )
  System.out.print( it.previous() + " " ); // 4 3 2 1
```



```
interface ListIterator<E>
extends Iterator<E>
```

- `boolean hasPrevious()`
- `boolean hasNext()`
  Liefert `true`, wenn es ein vorhergehendes/nachfolgendes Element gibt.
- `E previous()`
- `E next()`
  Liefert das vorangehende/nächste Element der Liste oder `NoSuchElementException`, wenn es das Element nicht gibt.
- `int previousIndex()`
- `int nextIndex()`
  Liefert den Index des vorhergehenden/nachfolgenden Elements. Geht `previousIndex()` vor die Liste, so liefert die Methode die Rückgabe –1. Geht `nextIndex()` hinter die Liste, liefert die Methode die Länge der gesamten Liste.
- `void remove()`
  Optional. Entfernt das letzte von `next()` oder `previous()` zurückgegebene Element.
- `void add(E o)`
  Optional. Fügt ein neues Objekt in die Liste ein.
- `void set(E o)`
  Optional. Ersetzt das Element, das `next()` oder `previous()` als Letztes zurückgegeben haben.

4.2.8 toArray(…) von Collection verstehen – die Gefahr einer Falle erkennen

Die `toArray()`-Methode aus der Schnittstelle `Collection` gibt laut Definition ein Array von Objekten zurück. Es ist wichtig, zu verstehen, welchen Typ die Einträge und das Array selbst haben. Eine Implementierung der `Collection`-Schnittstelle ist `ArrayList`.

Beispiel

Diese Anwendung von `toArray()` kopiert Punkte in ein Array:

```
ArrayList<Point> list = new ArrayList<>();
list.add( new Point(13, 43) );
list.add( new Point(9, 4) );
Object[] points = list.toArray();
```

Wir erhalten nun ein Array mit Referenzen auf `Point`-Objekte, können jedoch zum Beispiel nicht einfach `points[1].x` schreiben, um auf das Attribut des `Point`-Exemplars zuzugreifen, denn das Array `points` hat den deklarierten Elementtyp `Object`. Es fehlt die explizite Typumwandlung, und erst `((Point)points[1]).x` ist korrekt. Doch spontan kommen wir sicherlich auf die Idee, einfach den Typ des Arrays auf `Point` zu ändern. In dem Array befinden sich ja nur Referenzen auf `Point`-Exemplare:

```
Point[] points = list.toArray();            // ☠ Compilerfehler
```

Jetzt wird der Compiler einen Fehler melden, weil der Rückgabewert von `toArray()` ein `Object[]` ist. Spontan reparieren wir dies, indem wir eine Typumwandlung auf ein `Point`-Array an die rechte Seite setzen:

```
Point[] points = (Point[]) list.toArray();   // ☠ Problem zur Laufzeit
```

Jetzt haben wir zur Übersetzungszeit kein Problem mehr, aber zur Laufzeit wird es immer knallen, auch wenn sich im Array tatsächlich nur `Point`-Objekte befinden.

Diesen Programmierfehler müssen wir verstehen. Was wir falsch gemacht haben, ist einfach: Wir haben den Typ des Arrays mit den Typen der Array-Elemente durcheinandergebracht. Einem Array von Objektreferenzen können wir alles zuweisen:

```
Object[] os = new Object[ 3 ];
os[0] = new Point();
os[1] = "Trecker fahr'n";
os[2] = LocalDate.now();
```

Wir merken, dass der Typ des Arrays `Object[]` ist, und die Array-Elemente sind ebenfalls vom Typ `Object`. Hinter dem Schlüsselwort `new`, das das Array-Objekt erzeugt, steht der gemeinsame Obertyp für zulässige Array-Elemente. Bei `Object[]`-Arrays dürfen die Elemente Referenzen für beliebige Objekte sein. Klar ist, dass ein Array nur Objektreferenzen aufnehmen kann, die mit dem Typ für das Array selbst kompatibel sind, sich also auf Exemplare der angegebenen Klasse beziehen oder auf Exemplare von Unterklassen dieser Klasse:

```
/* 1 */  Object[] os = new Point[ 3 ];
/* 2 */  os[0] = new Point();
```



```
/* 3 */  os[1] = LocalDate.now();      // ☠ ArrayStoreException
/* 4 */  os[2] = "Trecker fahr'n";     // ☠ ArrayStoreException
```

Zeilen 3 und 4 sind vom Compiler erlaubt, führen aber zur Laufzeit zu einer `ArrayStoreException`.

Kommen wir wieder zur Methode `toArray()` zurück. Weil die auszulesende Datenstruktur alles Mögliche enthalten kann, muss der Typ der Elemente also `Object` sein. Wir haben gerade festgestellt, dass der Elementtyp des Array-Objekts, das die Methode `toArray()` als Ergebnis liefert, mindestens so umfassend sein muss. Da es keinen allgemeineren (umfassenderen) Typ als `Object` gibt, ist auch der Typ des Arrays `Object[]`. Dies muss so sein, auch wenn die Elemente einer Datenstruktur im Einzelfall einen spezielleren Typ haben. Einer allgemeingültigen Implementierung von `toArray()` bleibt gar nichts anderes übrig, als das Array vom Typ `Object[]` und die Elemente vom Typ `Object` zu erzeugen.

Wenn sich auch die Elemente wieder in einen spezielleren Typ konvertieren lassen, ist das bei dem Array-Objekt selbst jedoch nicht der Fall. Ein Array-Objekt mit Elementen vom Typ `X` ist nicht automatisch auch selbst vom Typ `X[]`, sondern von einem Typ `Y[]`, wobei `Y` eine (echte) Oberklasse von `X` ist.

Die Lösung für das Problem

Bevor wir nun eine Schleife mit einer Typumwandlung für jedes einzelne Array-Element schreiben oder eine Typumwandlung bei jedem Zugriff auf die Elemente vornehmen, sollten wir einen Blick auf die zweite `toArray(T[])`-Methode werfen. Sie akzeptiert als Parameter ein vorgefertigtes Array für das Ergebnis. Mit dieser Methode lässt sich erreichen, dass das Ergebnis-Array von einem spezielleren Typ als `Object[]` ist.

Beispiel

Wir fordern von der `toArray()`-Methode ein Array vom Typ `Point`:

```
List<Point> list = new ArrayList<>();
list.add( new Point(13,43) );
list.add( new Point(9,4) );
Point[] points = (Point[]) list.toArray( new Point[0] );
```

Die Listenelemente bekommen wir in ein Array kopiert, und der Typ des Arrays ist `Point[]` – passend zu den aktuell vorhandenen Listenelementen. Der Parameter zeigt dabei den Wunschtyp an, der hier das `Point`-Array ist.

Performance-Tipp

Am besten ist es, bei `toArray(T[])` ein Array anzugeben, das so groß ist wie das Ergebnis-Array, also so groß wie die Liste. Dann füllt nämlich `toArray(T[])` genau dieses Array und gibt es zurück, anstatt ein neues Array aufzubauen:

```
ArrayList<Point> list = new ArrayList<>();
list.add( new Point(13,43) );
list.add( new Point(9,4) );
Point[] points = list.toArray( new Point[list.size()] );
```

Arrays mit Reflection anlegen *

Spannend ist die Frage, wie so etwas funktionieren kann. Dazu verwendet die Methode `toArray(T[])` die Technik *Reflection*, um dynamisch ein Array vom gleichen Typ wie das übergebene Array zu erzeugen.

Wollten wir ein Array `b` vom Typ des Arrays `a` mit Platz für `len` Elemente anlegen, so schreiben wir:

```
Object[] b=(Object[])Array.newInstance(a.getClass().getComponentType(), len);
```

Mit `a.getClass().getComponentType()` erhalten wir ein `Class`-Objekt für den Elementtyp des Arrays. Zum Beispiel liefert das `Class`-Objekt `Point.class` für die Klasse `Point`. `a.getClass()` allein ein `Class`-Objekt für das Array `a`, etwa ein Objekt, das den Typ `Point[]` repräsentiert. `Array.newInstance()`, eine statische Methode von `java.lang.reflect.Array`, konstruiert ein neues Array mit dem Elementtyp aus dem `Class`-Objekt und der angegebenen Länge. Nichts anderes macht auch ein `new X[len]`, nur dass hier der Elementtyp zur Übersetzungszeit festgelegt werden muss. Da der Rückgabewert von `newInstance()` ein allgemeines `Object` ist, muss letztendlich noch die Konvertierung in ein passendes Array stattfinden.

Ist das übergebene Array so groß, dass es alle Elemente der Sammlung aufnehmen kann, kopiert `toArray(T[])` die Elemente aus der Collection in das Array. Im Übrigen entspricht `toArray(new Object[0])` dem Aufruf von `toArray()`.

4.2.9 Primitive Elemente in Datenstrukturen verwalten

Jede Datenstruktur der Collection-API akzeptiert, auch wenn sie generisch verwendet wird, nur Objekte. Primitive Datentypen nehmen die Sammlungen nicht auf, was zur Konsequenz hat, dass Wrapper-Objekte nötig sind (über das Boxing fügt Java scheinbar primitive Elemente ein, doch in Wahrheit sind es Wrapper-Objekte). Auch wenn es also

```
List<Double> list = new ArrayList<>();
list.add( 1.1 );
list.add( 2.2 );
```

heißt, sind es zwei neue `Double`-Objekte, die aufgebaut werden und in die Liste wandern. Anders und klarer geschrieben, sehen wir hier, was wirklich passiert:


```
List<Double> list = new ArrayList<>();
list.add( Double.valueOf(1.1) );
list.add( new Double(2.2) );
```

Dem Aufruf von `Double.valueOf(…)` ist die `new`-Nutzung nicht abzulesen, doch die Methode ist implementiert als: `Double valueOf(double d){ return new Double(d); }`.

Spezialbibliotheken

Für performante Anwendungen und große Mengen von primitiven Elementen ist es sinnvoll, eine Klasse für den speziellen Datentyp einzusetzen. Anstatt so etwas selbst zu programmieren, kann der Entwickler auf drei Implementierungen zurückgreifen:

- *fastutil* (*http://fastutil.di.unimi.it/*): Erweiterung um Datenstrukturen für (sehr viele) primitive Elemente und hochperformante Ein-/Ausgabe-Klassen
- *GNU Trove: High performance collections for Java* (*https://bitbucket.org/trove4j/trove*): Stabil, und die Entwicklung ist relativ aktiv.
- *Eclipse Collections* (*https://www.eclipse.org/collections/*): Neben primitiven Sammlungen reiche API. Ehemals *GS Collections*.

4.3 Mengen (Sets)

Eine Menge ist eine (erst einmal) ungeordnete Sammlung von Elementen. Jedes Element darf nur einmal vorkommen. Für Mengen sieht die Java-Bibliothek die Schnittstelle `java.util.Set` vor. Beliebte implementierende Klassen sind:

- `HashSet`: schnelle Mengenimplementierung durch Hashing-Verfahren (dahinter steckt die `HashMap`)
- `TreeSet`: Mengen werden durch balancierte Binärbäume realisiert, die eine Sortierung ermöglichen.
- `LinkedHashSet`: schnelle Mengenimplementierung unter Beibehaltung der Einfügereihenfolge
- `EnumSet`: eine spezielle Menge ausschließlich für Aufzählungen
- `CopyOnWriteArraySet`: schnelle Datenstruktur für viele lesende Operationen

4.3.1 Ein erstes Mengen-Beispiel

Das folgende Programm analysiert einen Text und erkennt Städte, die vorher in eine Datenstruktur eingetragen wurden. Alle Städte, die im Text vorkommen, werden gesammelt und später ausgegeben.

Listing 4.6 src/main/java/com/tutego/insel/util/set/WhereHaveYouBeen.java

```
package com.tutego.insel.util.set;

import java.text.BreakIterator;
import java.util.*;

public class WhereHaveYouBeen {
  public static void main( String[] args ) {
    // Menge mit Städten aufbauen

    Set<String> allCities = new HashSet<>();
    allCities.add( "Sonsbeck" );
    allCities.add( "Düsseldorf" );
    allCities.add( "Manila" );
    allCities.add( "Seol" );
    allCities.add( "Siquijor" );

    // Menge für besuchte Städte aufbauen

    Set<String> visitedCities = new TreeSet<>();

    // Satz parsen und in Wörter zerlegen. Alle gefundenen Städte
    // in neue Datenstruktur aufnehmen
    String sentence =
"Von Sonsbeck fahre ich nach Düsseldorf und fliege nach Manila.";
    BreakIterator iter = BreakIterator.getWordInstance();
    iter.setText( sentence );

    for ( int first = iter.first(), last = iter.next();
          last != BreakIterator.DONE;
          first = last, last = iter.next() ) {
      String word = sentence.substring( first, last );
      if ( allCities.contains( word ) )
        visitedCities.add( word );
    }

    // Kleine Statistik

    System.out.println( "Anzahl besuchter Städte: " + visitedCities.size() );
    System.out.println( "Anzahl nicht besuchter Städte: " +
                        (allCities.size() - visitedCities.size()) );
    System.out.println( "Besuchte Städte: " + String.join( ", ", visitedCities ) );
```



```
    Set<String> unvisitedCities = new TreeSet<>( allCities );
    unvisitedCities.removeAll( visitedCities );
    System.out.println( "Unbesuchte Städte: " + String.join( ", ", unvisitedCities ) );
  }
}
```


Insgesamt kommen drei Mengen im Programm vor:

- `allCities` speichert alle möglichen Städte. Die Wahl fällt auf den Typ `HashSet`, da die Menge nicht sortiert sein muss, Nebenläufigkeit kein Thema ist und `HashSet` eine gute Zugriffszeit bietet.
- Ein `TreeSet visitedCities` merkt sich die besuchten Städte. Auch dieses `Set` ist schnell, hat aber den Vorteil, dass es die Elemente sortiert hält. Das ist später hübsch in der Ausgabe.
- Um alle nicht besuchten Städte herauszufinden, berechnet das Programm die Differenzmenge zwischen allen Städten und besuchten Städten. Es gibt in der Schnittstelle `Set` keine Methode, die das direkt macht, genau genommen gibt es keine Operation in `Set`, die den Rückgabetyp `Set` oder `Collection` hat. Also können wir nur mit einer Methode wie `removeAll(…)` arbeiten, die aus der Menge aller Städte die besuchten entfernt, um zu denen zu kommen, die noch nicht besucht wurden. Das »Problem« der `removeAll(…)`-Methode ist aber ihre zerstörerische Art – die Elemente werden aus der Menge gelöscht. Da die Originalmenge jedoch nicht verändert werden soll, kopieren wir alle Städte in einen Zwischenspeicher (`unvisitedCities`) und löschen aus diesem Zwischenspeicher, was die Originalmenge unangetastet lässt.

4.3.2 Methoden der Schnittstelle Set

Eine Mengenklasse deklariert neben Operationen für die Anfrage und das Einfügen von Elementen auch Methoden für Schnitt und Vereinigung von Mengen.

```
interface java.util.Set<E>
extends Collection<E>
```

- `boolean add(E o)`
  Setzt `o` in die Menge, falls es dort noch nicht vorliegt. Liefert `true` bei erfolgreichem Einfügen.
- `boolean addAll(Collection<? extends E> c)`
  Fügt alle Elemente von `c` in das `Set` ein und liefert `true` bei erfolgreichem Einfügen. Ist `c` ein anderes `Set`, so steht `addAll(…)` für die Mengenvereinigung.
- `void clear()`
  Löscht das `Set`.

- `boolean contains(Object o)`
  Ist das Element `o` in der Menge?
- `boolean containsAll(Collection<?> c)`
  Ist `c` eine Teilmenge von `Set`?
- `boolean isEmpty()`
  Ist das `Set` leer?
- `Iterator<E> iterator()`
  Gibt einen Iterator für das `Set` zurück.
- `boolean remove(Object o)`
  Löscht `o` aus dem `Set`, liefert `true` bei erfolgreichem Löschen.
- `boolean removeAll(Collection<?> c)`
  Löscht alle Elemente der `Collection` aus dem `Set` und liefert `true` bei erfolgreichem Löschen.
- `boolean retainAll(Collection<?> c)`
  Bildet die Schnittmenge mit `c`.
- `int size()`
  Gibt die Anzahl der Elemente in der Menge zurück.
- `Object[] toArray()`
  Erzeugt zunächst ein neues Array, in dem alle Elemente der Menge Platz finden, und kopiert anschließend die Elemente in das Array.
- `<T> T[] toArray(T[] a)`
  Ist das übergebene Array groß genug, dann werden alle Elemente der Menge in das Array kopiert. Ist das Array zu klein, wird ein neues Array vom Typ `T` angelegt, und alle Elemente werden vom `Set` in das Array kopiert und zurückgegeben.

In der Schnittstelle `Set` werden die aus `Object` stammenden Methoden `equals(…)` und `hashCode()` mit ihrer Funktionalität bei Mengen in der API-Dokumentation präzisiert.

[+]

Hinweis

In einem `Set` gespeicherte Elemente müssen immutable bleiben. Einerseits sind sie nach einer Änderung vielleicht nicht wiederzufinden, und andererseits können Elemente auf diese Weise doppelt in der Menge vorkommen, was der Philosophie der Schnittstelle widerspricht.

Ein Element erneut hinzunehmen

Ist ein Element in der Menge noch nicht vorhanden, fügt `add(…)` es ein und liefert als Rückgabe `true`. Ist es schon vorhanden, macht `add(…)` nichts und liefert `false` (das ist bei einer `Map` anders, denn dort überschreibt `put(…)` den Schlüssel). Ob ein hinzuzufügendes Element mit einem existierenden in der Menge übereinstimmt, bestimmt die `equals(…)`-Methode, also die Gleichheit und nicht die Identität:

Listing 4.7 src/main/java/com/tutego/insel/util/set/HashSetDoubleAdd.java, main()

```
Set<Point> set = new HashSet<>();
Point p1 = new Point(), p2 = new Point();
System.out.println( set.add(p1) );      // true
System.out.println( set.add(p1) );      // false
System.out.println( set.add(p2) );      // false
System.out.println( set.contains(p1) ); // true
System.out.println( set.contains(p2) ); // true
```


4.3.3 HashSet

Ein `java.util.HashSet` verwaltet die Elemente in einer schnellen hashbasierten Datenstruktur. Dadurch sind die Elemente schnell einsortiert und schnell zu finden. Falls eine Sortierung vom `HashSet` nötig ist, müssen die Elemente nachträglich umkopiert und dann sortiert werden.

```
class java.util.HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

- `HashSet()`
  Erzeugt ein leeres `HashSet`-Objekt.
- `HashSet(Collection<? extends E> c)`
  Erzeugt aus der Sammlung `c` ein neues unsortiertes `HashSet`.
- `HashSet(int initialCapacity)`
- `HashSet(int initialCapacity, float loadFactor)`
  Die beiden Konstruktoren sind zur Optimierung gedacht und werden bei der `HashMap` im Abschnitt »Der Füllfaktor und die Konstruktoren« in Abschnitt 4.6.9 genauer erklärt – `HashSet` basiert intern auf der `HashMap`.

4.3.4 TreeSet – die sortierte Menge

Die Klasse `java.util.TreeSet` implementiert ebenfalls wie `HashSet` die `Set`-Schnittstelle, verfolgt aber eine andere Implementierungsstrategie. Ein `TreeSet` verwaltet die Elemente immer sortiert (intern werden die Elemente in einem balancierten Binärbaum gehalten). Speichert `TreeSet` ein neues Element, so fügt `TreeSet` das Element automatisch sortiert in die Datenstruktur ein. Das kostet zwar etwas mehr Zeit als ein `HashSet`, doch ist diese Sortierung dauerhaft. Daher ist es auch nicht zeitaufwändig, alle Elemente geordnet auszugeben. Die Suche nach einem einzigen Element ist aber etwas langsamer als im `HashSet`. Der Begriff »langsamer« muss jedoch relativiert werden: Die Suche ist logarithmisch und daher nicht wirklich »langsam«. Beim Einfügen und Löschen muss bei bestimmten Konstellationen eine

Reorganisation des Baums in Kauf genommen werden, was die Einfüge-/Löschzeit verschlechtert. Doch auch beim Re-Hashing gibt es diese Kosten, die sich dort jedoch durch die passende Startgröße vermeiden lassen.

```
class java.util.TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

- `TreeSet()`
  Erzeugt ein neues, leeres `TreeSet`.
- `TreeSet(Collection<? extends E> c)`
  Erzeugt ein neues `TreeSet` aus der gegebenen `Collection`.
- `TreeSet(Comparator<? super E> c)`
  Erzeugt ein leeres `TreeSet` mit einem gegebenen `Comparator`, der für die Sortierung der internen Datenstruktur die Vergleiche übernimmt.
- `TreeSet(SortedSet<E> s)`
  Erzeugt ein neues `TreeSet`, und übernimmt alle Elemente von `s` und auch die Sortierung von `s`. (Einen Konstruktor mit `NavigableSet` gibt es nicht.)

Beispiel

Teste, ob eine Liste von Datumswerten aufsteigend sortiert ist:

```
List<Instant> dates = Arrays.asList( Instant.ofEpochMilli( 2L ),
                                     Instant.ofEpochMilli( 3L ) );
boolean isSorted = new ArrayList<>( new TreeSet<>( dates ) ).equals( dates );
```

Nimmt der Konstruktor von `TreeSet` eine andere Sammlung entgegen, so entsteht eine sortierte Sammlung aller Elemente. Diese Sammlung kann wiederum in einen anderen Konstruktor gegeben werden, der `Collection`-Objekte annimmt, wie zum Beispiel eine `ArrayList`. Unser Beispiel vergleicht zwei `List`-Exemplare mit `equals(…)`, wobei Listen eine Ordnung haben. Stimmt die Ordnung nach dem Sortieren mit der vor der Sortierung überein, war die Liste schon sortiert.

Bedeutung der Sortierung

Durch die interne sortierte Speicherung gibt es zwei ganz wichtige Bedingungen:

- Die Elemente müssen sich vergleichen lassen. Kommen zum Beispiel `Player`-Objekte in das `TreeSet`, aber implementiert `Player` nicht die Schnittstelle `Comparable`, löst `TreeSet` eine Ausnahme aus, da `TreeSet` nicht weiß, in welcher Reihenfolge die Spieler stehen.
- Die Elemente müssen vom gleichen Typ sein. Wie sollte sich ein Kirchen-Objekt mit einem Staubsauger-Objekt vergleichen lassen?

Beispiel

Sortiere Strings in eine Menge ein, wobei die Groß-/Kleinschreibung und vorangestellter bzw. nachfolgender Weißraum keine Rolle spielen. Anders gesagt: Wörter sollen auch dann als gleich angesehen werden, wenn sie sich in der Groß-/Kleinschreibweise unterscheiden oder etwa Weißraum am Anfang und Ende besitzen:

```
Comparator<String> comparator = (s1, s2) -> String.CASE_INSENSITIVE_
ORDER.compare( s1.trim(), s2.trim() );
Set<String> set = new TreeSet<>( comparator );
Collections.addAll( set, "xxx ", " XXX", "tang", " xXx", " QUEEF " );
System.out.println( set ); // [ QUEEF , tang, xxx ]
```

Die Methode equals(…) und die Vergleichsmethoden

Die Methode `equals(…)` spielt für Datenstrukturen eine große Rolle. Beim `TreeSet` ist das anders, denn es nutzt zur Einordnung einen externen `Comparator` bzw. die `compareTo(…)`-Eigenschaft, wenn die Elemente `Comparable` sind. Gibt die Vergleichsmethode 0 zurück, so sind die Elemente gleich, und gleiche Elemente sind in der Menge nicht erlaubt – `equals(…)` wird dabei nicht gefragt!

Nehmen wir als Beispiel den `Comparator` aus dem vorangegangenen Beispiel für `String`-Objekte, der unabhängig von der Groß-/Kleinschreibung und Weißraum vergleicht. Dann sind laut `equals(…)` die Strings `"xxx "` und `" XXX"` sicherlich nicht gleich, der `Comparator` würde aber Gleichheit anzeigen. Dies führt dazu, dass tatsächlich nur eines der beiden Objekte in das `TreeSet` kommt und eine Anfrage nach einem `Comparator`-gleichen Objekt daher das Element liefert:

```
Comparator<String> comparator = (s1, s2) -> String.CASE_INSENSITIVE_
ORDER.compare( s1.trim(), s2.trim() );
Set<String> set = new TreeSet<>( comparator );
```

4.3.5 Die Schnittstellen NavigableSet und SortedSet

`TreeSet` implementiert die Schnittstelle `NavigableSet` und bietet darüber Methoden, die insbesondere zu einem gegebenen Element das nächsthöhere/-kleinere liefern. Somit sind auf Mengen nicht nur die üblichen Abfragen über Mengenzugehörigkeit denkbar, sondern auch Abfragen wie »Gib mir das Element, das größer oder gleich einem gegebenen Element ist«.

Folgendes Beispiel reiht in ein `TreeSet` drei `Calendar`-Objekte ein – die Klasse `Calendar` implementiert `Comparable<Calendar>`. Die Methoden `lower(…)`, `ceiling(…)`, `floor(…)` und `higher(…)` wählen aus der Menge das angefragte Objekt aus:

Listing 4.8 src/main/java/com/tutego/insel/util/set/SortedSetDemo.java

```
NavigableSet<Calendar> set = new TreeSet<>();
set.add( new GregorianCalendar(2007, Calendar.MARCH, 10) );
set.add( new GregorianCalendar(2007, Calendar.MARCH, 12) );
set.add( new GregorianCalendar(2007, Calendar.APRIL, 12) );

Calendar cal1 = set.lower( new GregorianCalendar(2007, Calendar.MARCH, 12) );
System.out.printf( "%tF%n", cal1 );   // 2007-03-10

Calendar cal2 = set.ceiling( new GregorianCalendar(2007, Calendar.MARCH, 12) );
System.out.printf( "%tF%n", cal2 );   // 2007-03-12

Calendar cal3 = set.floor( new GregorianCalendar(2007, Calendar.MARCH, 12) );
System.out.printf( "%tF%n", cal3 );   // 2007-03-12

Calendar cal4 = set.higher( new GregorianCalendar(2007, Calendar.MARCH, 12) );
System.out.printf( "%tF%n", cal4 );   // 2007-04-12
```

Eine Methode wie `tailSet(…)` ist insbesondere bei Datumsobjekten sehr praktisch, da sie alle Zeitpunkte liefern kann, die nach einem Startdatum liegen.

`TreeSet` implementiert die Schnittstelle `NavigableSet`, die ihrerseits `SortedSet` erweitert. Insgesamt bietet `NavigableSet` 15 Operationen, wobei sie aus `SortedSet` die Methoden `headSet(…)`, `tailSet(…)` und `subSet(…)` um die überladene Version der Methoden ergänzt, die die Grenzen exklusiv oder inklusiv erlauben.

```
interface java.util.NavigableSet<E>
extends SortedSet<E>
```

- `NavigableSet<E> headSet(E toElement)`
- `NavigableSet<E> tailSet(E fromElement)`
  Liefert eine Teilmenge von Elementen, die echt kleiner/größer als `toElement`/`fromElement` sind.
- `NavigableSet<E> headSet(E toElement, boolean inclusive)`
- `NavigableSet<E> tailSet(E fromElement, boolean inclusive)`
  Bestimmt gegenüber den oberen Methoden zusätzlich, ob das Ausgangselement zur Ergebnismenge gehören darf.
- `NavigableSet<E> subSet(E fromElement, E toElement)`
  Liefert eine Teilmenge im gewünschten Bereich.
- `E pollFirst()`
- `E pollLast()`
  Holt und entfernt das erste/letzte Element. Die Rückgabe ist `null`, wenn das `Set` leer ist.

- `E higher(E e)`
- `E lower(E e)`
  Liefert das folgende/vorangehende Element im `Set`, das echt größer/kleiner als `E` ist, oder `null`, falls ein solches Element nicht existiert.
- `E ceiling(E e)`
- `E floor(E e)`
  Liefert das folgende/vorangehende Element im `Set`, das größer/kleiner oder gleich `E` ist, oder `null`, falls ein solches Element nicht existiert.
- `Iterator<E> descendingIterator()`
  Liefert die Elemente in umgekehrter Reihenfolge.

Aus der Schnittstelle `SortedSet` erbt `NavigableSet` im Grunde nur drei Operationen, denn `subSet(…)`, `headSet(…)` und `tailSet(…)` werden mit kovariantem Rückgabetyp in `NavigableSet` re-definiert.

```
interface java.util.SortedSet<E>
extends Set<E>
```

- `E first()`
  Liefert das erste Element in der Liste.
- `E last()`
  Liefert das größte Element.
- `Comparator<? super E> comparator()`
  Liefert den mit der Menge verbundenen `Comparator`. Die Rückgabe kann `null` sein, wenn sich die Objekte mit `Comparable` selbst vergleichen können.
- `SortedSet<E> subSet(E fromElement, E toElement)`
- `SortedSet<E> headSet(E toElement)`
- `SortedSet<E> tailSet(E fromElement)`

Anders als `HashSet` liefert der Iterator beim `TreeSet` die Elemente aufsteigend sortiert. Davon profitieren auch die beiden `toArray(…)`-Methoden – implementiert in `AbstractCollection` –, da sie den Iterator nutzen, um ein sortiertes Array zurückzugeben.

Beispiel

Eine Variable `contacts` ist vom Typ `Map<Long,String>` und assoziiert IDs vom Typ `long` mit Strings. Ein neuer Kontakt soll eine ID bekommen, die um 1 höher ist als die höchste ID des Assoziativspeichers:

```
contact.setId( new TreeSet<Long>( contacts.keySet() ).last() + 1L );
```

4.3.6 LinkedHashSet

Ein `LinkedHashSet` vereint die Reihenfolgentreue einer Liste und die hohe Performance für Mengenoperationen vom `HashSet`. Dabei bietet die Klasse keine Listen-Methoden wie `first()` oder `get(int index)`, sondern ist eine Implementierung ausschließlich der `Set`-Schnittstelle, in der der Iterator die Elemente in der Einfügereihenfolge liefert:

Listing 4.9 src/main/java/com/tutego/insel/util/set/LinkedHashSetDemo.java, main()

```
Set<Integer> set = new LinkedHashSet<>(
  Arrays.asList( 9, 8, 7, 6, 9, 8 )
);

for ( Integer i : set )
  System.out.print( i + " " );      // 9 8 7 6

System.out.printf( "%n%s", set );   // [9, 8, 7, 6]
```

Da ein `Set` jedes Element nur einmal beinhalten kann, bekommen wir als Ergebnis jedes Element nur einmal, aber gleichzeitig geht die Reihenfolge des Einfügens nicht verloren. Der Iterator liefert die Elemente genau in der Einfügereihenfolge.

Beispiel

Dass ein `LinkedHashSet` eine Menge ist, die Elemente nur einmal enthält, sich aber beim Einfügen wie eine Liste verhält, ist nützlich, um doppelte Elemente aus einer Liste zu löschen:

```
public static <T> List<T> removeDuplicate( List<T> list ) {
  return new ArrayList<>( new LinkedHashSet<>( list ) );
}
```

Das Ergebnis ist eine neue Liste, und `list` selbst wird nicht modifiziert. Es ergibt zum Beispiel `removeDuplicate( Arrays.asList( 1,2,1,3,1,2,4 ) )` die Liste [1, 2, 3, 4].

LinkedHashSet und Iterator

Mit einem Iterator lässt sich jedes Element von `LinkedHashSet` nach der Reihenfolge des Einfügens auflisten. Der Iterator von `LinkedHashSet` unterstützt auch die `remove()`-Methode. Sie kann eingesetzt werden, um die ältesten Einträge zu löschen und nur noch die neuesten zwei Elemente beizubehalten:

```
LinkedHashSet<Integer> set = new LinkedHashSet<>();
set.addAll( Arrays.asList( 3, 2, 1, 6, 5, 4 ) );
System.out.println( set );  // [3, 2, 1, 6, 5, 4]
for ( Iterator<Integer> iter = set.iterator(); iter.hasNext(); ) {
  iter.next();
  if ( set.size() > 2 )
    iter.remove();
}
System.out.println( set );  // [5, 4]
```


4.4 Queues (Schlangen) und Deques

In der Klassenbibliothek von Java gibt es die Schnittstelle `java.util.Queue` für Datenstrukturen, die nach dem FIFO-Prinzip (für engl. *first in, first out*) arbeiten. Die verwandte Datenstruktur ist der `Stack`, der nach dem Prinzip LIFO (für engl. *last in, first out*) arbeitet. Eine Deque bietet Methoden für die FIFO-Verarbeitung an beiden Enden über eine Schnittstelle `java.util.Deque`, die direkt `java.util.Queue` erweitert.

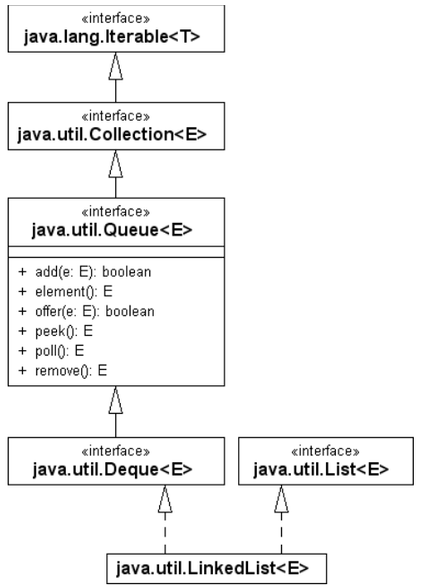

Abbildung 4.3 Queue- und Deque-Schnittstellen

4.4.1 Queue-Klassen

Die Schnittstelle `Queue` erweitert `Collection` und ist somit auch vom Typ `Iterable`. Zu den Klassen, die `Queue` implementieren, gehört unter anderem `LinkedList`:

Listing 4.10 src/main/java/com/tutego/insel/util/queue/QueueDemo.java, main()

```
Queue<String> queue = new LinkedList<>();
queue.offer( "Fischers" );
```

```
queue.offer( "Fritze" );
queue.offer( "fischt" );
queue.offer( "frische" );
queue.offer( "Fische" );

queue.poll();

queue.offer( "Nein, es war Paul!" );

while ( ! queue.isEmpty() )
  System.out.println( queue.poll() );
```

Die Operationen sind:

```
interface java.util.Queue<E>
extends Collection<E>
```

- boolean empty()
- E element()
- boolean offer(E o)
- E peek()
- E poll()
- E remove()

Auf den ersten Blick sieht es so aus, als ob es für das Erfragen zwei Methoden gibt: `element()` und `peek()`. Doch der Unterschied besteht darin, dass `element()` eine `NoSuchElementException` auslöst, wenn die Queue kein Element mehr liefern kann, `peek()` jedoch `null` bei leerer Queue liefert. Da `null` als Element erlaubt ist, kann `peek()` das nicht detektieren; die Rückgabe könnte für das `null`-Element oder als Anzeige für eine leere Queue stehen. Daher ist `peek()` nur sinnvoll, wenn keine `null`-Elemente vorkommen, was aber in der Regel erfüllt ist.

Gefüllt werden kann die Queue mit der von `Collection` geerbten Methode `add(…)` oder durch die Methode `offer(…)`. Der Unterschied: Im Fehlerfall löst `add(…)` eine Exception aus, während `offer(…)` durch die Rückgabe `false` anzeigt, dass das Element nicht hinzugefügt wurde. Tabelle 4.2 macht den Zusammenhang deutlich:

	Mit Ausnahme	Ohne Ausnahme
Einfügen	`add(…)`	`offer(…)`
Erfragen	`element()`	`peek()`
Löschen	`remove()`	`poll()`

Tabelle 4.2 Operationen von Queue mit Ausnahmen und ohne

4.4.2 Deque-Klassen

Eine *Deque* (gesprochen »Deck«) bietet Queue-Operationen an beiden Enden an. Die Operationen schreibt eine Schnittstelle `Deque` vor.

Deque-API

Da es jeweils die Queue-Operationen an beiden Enden gibt, erscheint die Schnittstelle erst einmal groß, was sie aber im Prinzip nicht ist.

	Erstes Element (Kopf)		Letztes Element (Schwanz)	
	Ausnahme im Fehlerfall	Besondere Rückgabe	Ausnahme im Fehlerfall	Besondere Rückgabe
Einfügen	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Löschen	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Entnehmen	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Tabelle 4.3 Operationen der Datenstruktur Deque

»Besondere Rückgabe« in der Tabelle bedeutet, dass etwa `getFirst()`/`getLast()` eine Ausnahme auslösen, wenn die `Deque` leer ist, aber `peekFirst()`/`peekLast()` die Rückgabe `null` liefern.

Beispiel

Die Methode `removeLast()` eignet sich gut dafür, die Datenstruktur auf eine gewisse Anzahl an Elementen zu beschränken. Nehmen wir an, eine Liste soll maximal zehn Elemente aufnehmen, dann können wir beim Einfügen eines neuen Elements schreiben:

```
if ( searches.size() > 10 )
  searches.removeLast();
```

Beispiel

Die Schnittstelle `Deque` deklariert neben der bekannten Methode `iterator()` auch die Methode `descendingIterator()`, die die Datenstruktur »von der anderen Seite« abläuft. Nehmen wir an, wir wollen für einen Domainnamen einen umgedrehten Domainnamen erzeugen, so wie er für Paketnamen verwendet wird:

```
String domain = "email.tutego.de";
new ArrayDeque<String>( Arrays.asList( domain.split( "(?!^)\\b" ) ) )
  .descendingIterator()
  .forEachRemaining( System.out::print );  // de.tutego.email
```

Deque-Implementierungen

Von der Schnittstelle gibt es drei Implementierungen:

- `ArrayDeque`: Wie der Namensbestandteil »Array« schon andeutet, sitzt hinter dieser Realisierung ein Array, das die Deque beschränken kann.
- `LinkedList`: Einer `LinkedList` ist diese Beschränkung fremd; sie kann beliebig wachsen.
- `LinkedBlockingDeque`: Realisiert `BlockingDeque` als blockierende `Deque`, was weder `ArrayDeque` noch `LinkedList` machen.

4.4.3 Blockierende Queues und Prioritätswarteschlangen

Die Schnittstelle `java.util.concurrent.BlockingQueue` erweitert die Schnittstelle `java.util.Queue`. Implementierende Klassen blockieren, falls eine Operation wie `take()` aufgrund fehlender Daten nicht durchgeführt werden konnte. Die Konsumenten/Produzenten sind mit diesen Klassen ausgesprochen einfach zu implementieren.

Spannende `Queue`-Klassen (und speziellere `BlockingQueue`-Klassen) sind:

- `ConcurrentLinkedQueue`: threadsichere Queue, durch verkettete Listen implementiert
- `DelayQueue`: Queue, der die Elemente erst nach einer gewissen Zeit entnommen werden können
- `ArrayBlockingQueue`: Queue mit einer maximalen Kapazität, abgebildet auf ein Array
- `LinkedBlockingQueue`: unbeschränkte oder beschränkte Queue (also mit maximaler Kapazität), abgebildet durch eine verkettete Liste
- `PriorityQueue`: Hält in einem Heap-Speicher Elemente sortiert und liefert bei Anfragen das jeweils größte Element. Wie beim `TreeSet` müssen die Elemente entweder `Comparable` implementieren, oder es muss ein `Comparator` angegeben werden. Unbeschränkt.
- `PriorityBlockingQueue`: wie `PriorityQueue`, nur blockierend
- `SynchronousQueue`: Eine blockierende Queue zum Austausch von genau einem Element. Wenn ein Thread ein Element in die Queue setzt, muss ein anderer Thread auf dieses Element warten, andernfalls blockiert die Datenstruktur den Ableger. Erwartet ein Thread ein Element, ohne dass ein anderer Thread etwas in die Queue gesetzt hat, blockiert sie ebenfalls den Holer. Durch diese Funktionsweise benötigt die `SynchronousQueue` keine Kapazität, denn Elemente werden, falls platziert, direkt konsumiert und müssen nicht zwischengelagert werden.

Die `PriorityXXX`-Klassen implementieren im Gegensatz zu den übrigen kein FIFO-Verhalten.

4.4.4 PriorityQueue

Eine Prioritätswarteschlange ist eine Datenstruktur, die vergleichbar wie ein `SortedSet` die Elemente sortiert. Wenn etwa Nachrichten in ein System kommen, sind diese in der Regel

nicht alle gleich wichtig, sondern unterscheiden sich in ihrer Priorität. Elemente höherer Priorität sollen zuerst verarbeitet werden. Das Fundament bildet eine Queue mit einem modifizierten FIFO-Prinzip, das deswegen nicht nur pur FIFO ist (»Wer zuerst kommt, mahlt zuerst«), weil Elemente, die schon am Anfang stehen, von später kommenden Elementen mit höherer Priorität verdrängt werden können.

`PriorityQueue` ist eine Implementierung einer solchen Prioritätswarteschlange. Damit die Priorisierung möglich ist, müssen die Elemente eine natürliche Sortierung besitzen (`String` oder Wrapper-Objekte haben diese zum Beispiel), oder ein `Comparator` muss angegeben werden. `PriorityQueue` verbietet das `null`-Element, und Elemente dürfen durchaus doppelt vorkommen (wie bei einer Liste); daher kann auch `SortedSet` nicht die Aufgabe einer Prioritätswarteschlange übernehmen, da eine Menge Elemente nur einmal enthalten kann.

Bleibt die Frage, ob Elemente hoher Priorität vorne oder hinten stehen, denn das beeinflusst die Implementierung. Eine Queue hat einen Kopf, und dort steht das kleinste Element. Das ist entgegen der Intuition, denn wir sprechen immer von »hoher Priorität«, nur bei der Datenstruktur ist es so, dass sich die hohe Priorität in »kleiner in der Ordnung« übersetzt. Sehr anschaulich zeigt es das nächste Beispiel, in dem wir einige `Integer`-Objekte in die `PriorityQueue` einsortieren; da `Integer` die Schnittstelle `Comparable` implementiert, ist dadurch eine natürliche Ordnung gegeben.

Listing 4.11 src/main/java/com/tutego/insel/util/queue/PriorityQueueDemo.java, main()

```
PriorityQueue<Integer> q = new PriorityQueue<>();
q.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8, 1 ) );
System.out.println( q );            // [1, 2, 1, 9, 3, 8, 3]
System.out.println( q.remove() ) ;  // 1
System.out.println( q );            // [1, 2, 3, 9, 3, 8]
System.out.println( q.remove() ) ;  // 1
System.out.println( q );            // [2, 3, 3, 9, 8]
System.out.println( q.remove() ) ;  // 2
System.out.println( q );            // [3, 8, 3, 9]
System.out.println( q.remove() ) ;  // 3
System.out.println( q.remove() ) ;  // 3
System.out.println( q.remove() ) ;  // 8
System.out.println( q );            // [9]
System.out.println( q.remove() ) ;  // 9
System.out.println( q );            // []
```

Beim Entnehmen mit `remove()` liefert die `PriorityQueue` immer das kleinste Element und entfernt es aus der Sammlung. Wichtig ist zu verstehen, dass der Iterator über die Elemente (anschaulich durch `toString()`) keine sortierte Sammlung zeigt – daher darf eine `Priority-`

`Queue` nicht als sortiertes `Set` gesehen werden, sondern eben als `java.util.Queue`, bei der nur Anfrage-Operationen am Kopf erlaubt sind. Dass dabei zum Beispiel die 9 vor der 8 steht, ist ein Internum, das zwar irgendwie interessant, aber für den Entwickler irrelevant ist, da nur Kopfoperationen wie `element()`, `remove()`, `peek()`, `poll()`, `remove()` erlaubt sind. Während das kleinste Element entnommen und entfernt wird, sortiert sich die interne Datenstruktur um, wobei dies auch die Reihenfolge der 8 und 9 in unserem Beispiel verändert, aber hier dringt nur die interne Arbeitsweise nach außen, die uns nicht interessiert.

Comparator für wichtige Strings

Um mit der Klasse `PriorityQueue` und den Vergleichen warm zu werden, wollen wir mit einem `Comparator` beginnen, der allen Zeichenketten mit den Wörtern »wichtig«, »important« und »sofort« eine höhere Priorität einräumt als anderen Zeichenketten. In die Rückgabe eines `Comparator` übersetzt, bedeutet das, dass die höher priorisierten Zeichenketten »kleiner« als die Zeichenketten ohne Signalwörter sind. Ein `s1 = "Wichtig! Essen ist fertig"` ist damit kleiner als `s2 = "Bier ist kalt!"`, und ein `compare(s1, s2)` würde –1 liefern.

Listing 4.12 src/main/java/com/tutego/insel/util/queue/ImportanceComparator.java, main()

```
enum ImportanceComparator implements Comparator<String> {

  INSTANCE;

  Predicate<String> IMPORTANT_PATTERN =
          Pattern.compile( "(?i)(wichtig|important|sofort)" ).asPredicate();

  @Override public int compare( String s1, String s2 ) {
    boolean isS1Important = IMPORTANT_PATTERN.test( s1 ),
            isS2Important = IMPORTANT_PATTERN.test( s2 );

    // Wenn beide Strings wichtig oder beide Strings unwichtig sind,
    // dann ist die Rückgabe 0.
    if ( isS1Important == isS2Important )
      return 0;    // wichtigkeit(s1) == wichtigkeit(s2)

    // Andernfalls ist einer der Strings wichtig und der andere unwichtig
    if ( isS1Important )
      return -1;   // wichtigkeit(s1) > wichtigkeit(s2) -> s1 < s2
    // else if ( isS2Important )
    return +1;     // wichtigkeit(s2) > wichtigkeit(s1) -> s1 > s2
  }
}
```


Die `Pattern`-Klasse aus dem `regex`-Paket hilft uns bei der Frage, ob ein Teil-String im String vorkommt, und zwar unabhängig von der Groß-/Kleinschreibung.

Die Funktionalität vom `Comparator` testet ein kleines Beispielprogramm:

Listing 4.13 src/main/java/com/tutego/insel/util/queue/ImportanceComparatorDemo.java, main()

```
ImportanceComparator comp = ImportanceComparator.INSTANCE;

String s1 = "Schönes Wetter heute.";
String s2 = "Schickes Kleid!";

// Beides nicht wichtig, daher ist das Ergebnis 0
System.out.println( comp.compare( s1, s2 ) );      // 0

String s3 = "Sofort nach den Blumen schauen!";

// Zweiter String wichtiger als der erste String
System.out.println( comp.compare( s1, s3 ) );      // 1

// Erster String wichtiger als der zweite String
System.out.println( comp.compare( s3, s1 ) );      // -1

String s4 = "Wichtig! An Lakritz denken!";

// Beide Strings gleich wichtig
System.out.println( comp.compare( s3, s4 ) );      // 0

List<String> list = new ArrayList<>();
Collections.addAll( list, s1, s2, s3, s4, s1 );
System.out.println( list );
Collections.sort( list, comp );
System.out.println( list );
```

Das Beispielprogramm endet mit einer Liste, die nach der Sortierung mit dem `Comparator` folgende Ausgabe liefert:

```
[Sofort nach den Blumen schauen!, Wichtig! An Lakritz denken!, Schönes Wetter heute., ↩
Schickes Kleid!, Schönes Wetter heute.]
```

Die wichtigen Zeichenketten stehen vorne in der Liste. Dort stehen sie genau richtig, damit die Prioritätswarteschlange sie direkt abholen kann.

Wichtige Meldungen in der Prioritätswarteschlange

Für die Prioritätswarteschlange ist das Element mit der höchsten Priorität das wichtigste und steht vorne, am Kopf.

Listing 4.14 src/main/java/com/tutego/insel/util/queue/ImportancePriorityQueue.java, main()

```
ImportanceComparator comp = ImportanceComparator.INSTANCE;
PriorityQueue<String> queue = new PriorityQueue<>( comp );

queue.add( "Schönes Wetter heute." );
System.out.println( queue );
// [Schönes Wetter heute.]
queue.add( "Sofort nach den Blumen schauen!" );
System.out.println( queue );
// [Sofort nach den Blumen schauen!, Schönes Wetter heute.]
queue.add( "Schickes Kleid!" );
System.out.println( queue );
// [Sofort nach den Blumen schauen!, Schönes Wetter heute., Schickes Kleid!]
queue.remove();
System.out.println( queue );
// [Schickes Kleid!, Schönes Wetter heute.]
queue.add( "Wichtig! An Lakritz denken!" );
System.out.println( queue );
// [Wichtig! An Lakritz denken!, Schönes Wetter heute., Schickes Kleid!]
```

Im Konstruktor von `PriorityQueue` geben wir den `Comparator` an, der für die Vergleiche herangezogen wird und die Elemente an den Kopf setzt.

Bemerkung

Bei unserer Implementierung kann es passieren, dass Elemente in der Queue verhungern, wenn sich wichtigere Elemente immer vorschieben.

```
class java.util.PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

- `PriorityQueue()`
  Erzeugt eine neue `PriorityQueue` mit der Anfangsgröße von elf Elementen. Die Elemente werden nach ihrer natürlichen Ordnung sortiert.
- `PriorityQueue(Collection<? extends E> c)`
  Erzeugt eine neue `PriorityQueue`, die mit den Elementen aus `c` vorkonfiguriert wird. Die Elemente werden nach ihrer natürlichen Ordnung sortiert.

- `PriorityQueue(int initialCapacity)`
  Erzeugt eine neue `PriorityQueue` mit der Anfangsgröße `initialCapacity`. Die Elemente werden nach ihrer natürlichen Ordnung sortiert. Die Initialkapazität ist eine Größenabschätzung, die für die Performanz bei vielen Elementen wichtig ist.
- `PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`
  Erzeugt eine neue `PriorityQueue` mit der Anfangsgröße `initialCapacity`. Die Elemente werden mit einem `Comparator` verglichen.
- `PriorityQueue(Comparator<? super E> comparator)`
  Erzeugt eine neue `PriorityQueue` mit der Anfangsgröße von elf Elementen. Die Elemente werden mit einem `Comparator` verglichen.
- `PriorityQueue(PriorityQueue<? extends E> c)`
- `PriorityQueue(SortedSet<? extends E> c)`
  Erzeugt eine neue `PriorityQueue`, die mit den Elementen aus `c` vorkonfiguriert wird. Die Sortiereigenschaft, entweder natürlich oder mit einem `Comparator`, wird aus der übergebenen `PriorityQueue` bzw. dem `SortedSet` übernommen.

Die Klasse `PriorityQueue` übernimmt die Operationen aus der Schnittstelle `Queue` und fügt nur eine neue Methode hinzu:

- `Comparator<? super E> comparator()`
  Liefert den `Comparator` der `PriorityQueue` oder `null`, falls die natürliche Ordnung verwendet wird.

Wichtig zu wissen ist, dass die Elemente entweder eine natürliche Ordnung haben müssen oder sich ein `Comparator` um die Vergleiche kümmern muss – der `Comparator` kann später nicht mehr verändert werden! Daher gibt es auch nur eine `comparator()`-Methode zum Holen, aber das Zuweisen eines `Comparator` geschieht immer nur einmal im Konstruktor.

4.5 Stack (Kellerspeicher, Stapel)

Ein Stapelspeicher, auch *Keller* genannt, ist eine LIFO-(Last-in-First-out-)Datenstruktur. Die lässt sich mit einem Stapel Teller veranschaulichen. Beim Hinzufügen von Elementen wächst die Datenstruktur dynamisch, wobei das zuletzt abgelegte Element als erstes auch wieder entnommen wird – ein wahlfreier Zugriff ist nicht vorgesehen.

In Java gibt es drei unterschiedliche Herangehensweisen, falls ein LIFO-Verhalten erwünscht ist:

- `Stack`: Die Klasse `Stack` repräsentiert einen Stapelspeicher seit Java 1.0. Die Methode `push(…)` setzt Elemente auf den Stack, `pop()` holt sie herunter. `Stack` ist nicht mehr angebracht.

- `Deque` ist ein Typ, dessen implementierende Klassen sich je nach Methoden FIFO (für engl. *first in, first out*) oder LIFO verhalten. Das bestimmen die Methoden. Soll sich eine `Deque`-Datenstruktur als LIFO-Stack verhalten, sind die Methoden `addFirst(…)` (entspricht `push(…)` bei `Stack`) und `removeFirst()` (entspricht `pop()` bei `Stack`) zu nutzen. Eine `LinkedList` implementiert `Deque`, kann also als Stack verwendet werden.
- `Collections.asLifoQueue(Deque)`: `Deque` zeigt das LIFO-Verhalten nur bei den entsprechenden Methoden `addFirst(…)` und `removeFirst(…)`, doch ist der Typ anfällig für Programmierfehler, denn schnell ist statt `addFirst(…)` nur `add(…)` geschrieben, und schon landet das Element an der falschen Stelle. Interessant ist daher die Methode `Collections.asLifoQueue(Deque)`. Sie liefert eine `Queue` zurück, also eine Datenstruktur mit weniger Methoden als eine `Deque`, wobei das traditionelle FIFO-Prinzip einer Queue umgedreht wurde in ein LIFO-Verhalten. Mit anderen Worten: Die Queue einer `Collections.asLifoQueue(Deque)` zeigt das typische Verhalten eines Stacks. Und weil die Schnittstelle `Queue` nur wenige Methoden hat, minimiert sich die Fehlerquelle, aus Versehen die falsche Methode aufgerufen zu haben.

Beispiel

Füge in den Stack `view` Strings ein, und lies sie wieder aus:

```
Queue<String> stack = Collections.asLifoQueue( new LinkedList<>() );
stack.add( "59" );
stack.add( "59ing" );
stack.addAll( Arrays.asList( "fifty-nine", "fifty-nining" ) );
while ( ! stack.isEmpty() )
  System.out.println( stack.remove() ); // fifty-nining fifty-nine 59ing 59
```

4.5.1 Die Methoden von java.util.Stack

Die Klasse erweitert `Vector` (wir diskutieren diese prickelnde Designentscheidung weiter unten), womit die Klasse zusätzliche Funktionalität besitzt, beispielsweise die Fähigkeit zur Aufzählung und zum wahlfreien Zugriff auf Kellerelemente. `Stack` besitzt nur wenige zusätzliche Methoden.

```
class java.util.Stack<E>
extends Vector<E>
```

- `Stack()`
  Der Konstruktor erzeugt einen neuen Stack.
- `boolean empty()`
  Testet, ob Elemente auf dem Stapel vorhanden sind.

- `E push(E item)`
  Das Element `item` wird auf den Stapel gebracht.
- `E pop()`
  Holt das letzte Element vom Stapel. `EmptyStackException` signalisiert einen leeren Stapel.
- `E peek()`
  Das oberste Element wird nur vom Stapel gelesen, aber nicht wie bei `pop()` entfernt. Bei leerem Stapel wird eine `EmptyStackException` ausgelöst.
- `int search(Object o)`
  Sucht im Stapel nach dem obersten Eintrag, der mit dem Objekt `o` übereinstimmt. Gibt die Distanz von der Spitze zurück oder –1, falls das Objekt nicht im Stapel ist. 1 bedeutet, dass der gesuchte Eintrag ganz oben auf dem Stapelspeicher liegt, 2 bezeichnet die zweitoberste Position usw. Die Zählweise ist ungewöhnlich, da sie nicht nullbasiert ist wie alle anderen Methoden, die mit Positionen arbeiten. Doch hier handelt es sich ausdrücklich um die Distanz und nicht um die Position!

Hinweis

Exceptions von Stack: Im Gegensatz zu `Vector` kann `Stack` die Exception `EmptyStackException` erzeugen, um einen leeren Stapel zu signalisieren. Durch einen Rückgabewert `null` ist ein Fehlschlag nicht angezeigt, da `null` ein gültiger Rückgabewert sein kann.

Ein Stack ist ein Vector – aha!

Eine genaue Betrachtung der Klasse `Stack` zeigt den unsinnigen und falschen Einsatz der Vererbung, und es stellt sich die Frage, warum sich der Autor Jonathan Payne für jene Variante entschieden hat. `Stack` erbt alle Methoden von `Vector` (einer `List`) und damit viele Methoden, die im krassen Gegensatz zu den charakteristischen Eigenschaften eines Stapels stehen. Dazu zählen unter anderem die Methoden `elementAt(…)`, `indexOf(…)`, `insertElementAt(…)`, `removeElementAt(…)`, `setElementAt(…)`. Wenn eine Unterklasse nicht bedingungslos alle Eigenschaften der Oberklasse unterstützt, ist die Vererbung falsch angewendet, es geht »kaufen vor erben«.

4.6 Assoziative Speicher

Ein assoziativer Speicher verbindet einen Schlüssel mit einem Wert. Java bietet für Datenstrukturen dieser Art die allgemeine Schnittstelle `Map` mit wichtigen Operationen wie `put(key, value)` zum Aufbau einer Assoziation und `get(key)` zum Erfragen eines assoziierten Werts.

4.6.1 Die Klassen HashMap und TreeMap

Die Java-Bibliothek implementiert assoziativen Speicher mit einigen Klassen, wobei wir unser Augenmerk zunächst auf zwei wichtige Klassen richten wollen:

- Eine schnelle Implementierung ist die *Hash-Tabelle* (engl. *hashtable*), die in Java durch `java.util.HashMap` implementiert ist. Vor Java 1.2 wurde `java.util.Hashtable` verwendet. Die Schlüsselobjekte müssen »hashbar« sein, also `equals(…)` und `hashCode()` konkret implementieren. Eine besondere Schnittstelle für die Elemente ist nicht nötig.
- Daneben existiert die Klasse `java.util.TreeMap`, die etwas langsamer im Zugriff ist, doch dafür alle Schlüsselobjekte immer sortiert hält. Sie sortiert die Elemente in einen internen Binärbaum ein. Die Schlüssel müssen sich in eine Ordnung bringen lassen, wozu etwas Vorbereitung nötig ist.

Beispiel

Ein Assoziativspeicher, dem wir Werte hinzufügen:

```
Map<String,String> aldiSupplier = new HashMap<>();
aldiSupplier.put( "Carbo, spanischer Sekt", "Freixenet" );
aldiSupplier.put( "ibu Stapelchips", "Bahlsen Chipsletten" );
aldiSupplier.put( "Ko-kra Katzenfutter", "felix Katzenfutter" );
aldiSupplier.put( "Küchenpapier", "Zewa" );
aldiSupplier.put( "Nuss-Nougat-Creme", "Zentis" );
aldiSupplier.put( "Pommes Frites", "McCain" );
```

Die zweite `HashMap` soll Strings mit Zahlen assoziieren:

```
Map<String,Number> num = new HashMap<>();
num.put( "zwei", 2 );    // Boxing durch Integer.valueOf(2)
num.put( "drei", 3.0 );  // Boxing durch Double.valueOf(3.0)
```

Während also bei den Assoziativspeichern nach dem Hashing-Verfahren eine `hashCode()`- und `equals(…)`-Methode bei den Schlüssel-Objekten essenziell ist, ist das bei den baumorientierten Verfahren nicht nötig – hier muss nur eine Ordnung zwischen den Elementen entweder mit `Comparable` oder `Comparator` her.

Ein Assoziativspeicher arbeitet nur in eine Richtung schnell. Wenn etwa im Fall eines Telefonbuchs ein Name mit einer Nummer assoziiert wurde, kann die Datenstruktur die Frage nach einer Telefonnummer schnell beantworten. In die andere Richtung dauert es wesentlich länger, weil hier keine Verknüpfung besteht. Sie ist immer nur einseitig. Auf wechselseitige Beziehungen sind die Klassen nicht vorbereitet.

Abbildung 4.4 UML-Klassendiagramm der Schnittstelle Map

Die Klasse HashMap

Die Klasse `HashMap` eignet sich ideal dazu, viele Elemente unsortiert zu speichern und sie über die Schlüssel schnell wieder verfügbar zu machen. Das interne Hashing-Verfahren ist schnell, eine Sortierung der Schlüssel nach einem gegebenen Kriterium aber nicht möglich.

```
class java.util.HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

- `HashMap()`
  Erzeugt eine neue Hash-Tabelle.
- `HashMap(Map<? extends K,? extends V> m)`
  Erzeugt eine neue Hash-Tabelle aus einer anderen `Map`.

Die Klasse TreeMap und die Schnittstelle SortedMap/NavigableMap

Eine `TreeMap` implementiert die Schnittstelle `NavigableMap`, die wiederum von der Schnittstelle `SortedMap` erbt, wobei diese wiederum `Map` erweitert. Eine `NavigableMap` sortiert die Elemente eines Assoziativspeichers nach Schlüsseln und bietet Zugriff auf das kleinste oder größte Element.

- Einige Methoden aus `SortedMap`: `firstKey()`, `lastKey()`. `subMap(fromKey, toKey)` und `tailMap(fromKey, toKey)` bilden Teilansichten des Assoziativspeichers.
- Einige Methoden aus `NavigableMap`: `pollFirstEntry()`, `pollLastEntry()`, `descendingMap()`

Damit die Schlüssel in einer `TreeMap` sortiert werden können, gilt das Gleiche wie beim `TreeSet`: Die Elemente müssen eine natürliche Ordnung besitzen, oder ein externer `Comparator` muss die Ordnung festlegen.

```
class java.util.TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

- `TreeMap()`
  Erzeugt eine neue `TreeMap`, die eine natürliche Ordnung von ihren Elementen erwartet.
- `TreeMap(Comparator<? super K> comparator)`
  Erzeugt eine neue `TreeMap` mit einem `Comparator`, sodass die Elemente keine natürliche Ordnung besitzen müssen.
- `TreeMap(Map<? extends K,? extends V> m)`
  Erzeugt eine `TreeMap` mit einsortierten Elementen aus `m`, die eine natürliche Ordnung besitzen müssen.
- `TreeMap(SortedMap<K,? extends V> m)`
  Erzeugt eine `TreeMap` mit einsortierten Elementen aus `m` und übernimmt von `m` auch die Ordnung.

Um die Sortierung zu ermöglichen, ist der Zugriff etwas langsamer als über `HashMap`, aber mit dem Hashing-Verfahren lassen sich Elemente nicht sortieren.

4.6.2 Einfügen und Abfragen des Assoziativspeichers

Wir haben gesagt, dass die Elemente des Assoziativspeichers Paare aus Schlüssel und zugehörigem Wert sind. Das Wiederfinden der Werte ist effizient nur über Schlüssel möglich.

Daten einfügen

Zum Hinzufügen von Schlüssel-Wert-Paaren dient die Methode `put(key, value)`. Das erste Argument ist der Schlüssel und das zweite Argument der mit dem Schlüssel zu assoziierende Wert.

Hinweis für Nullen

Bei manchen Implementierungen der `Map`-Schnittstelle kann der Schlüssel bzw. Wert `null` sein – hier lohnt ein Blick auf die Javadoc der einzelnen Klassen. Bei `HashMap` ist `null` als Schlüssel und Wert ausdrücklich erlaubt, bei `ConcurrentHashMap` dürfen weder Schlüssel noch Wert `null` sein.


```
interface java.util.Map<K,V>
```

- `V put(K key, V value)`
  Speichert den Schlüssel und den Wert im Assoziativspeicher. Falls sich zu diesem Schlüssel schon ein Eintrag im Assoziativspeicher befand, wird der alte Wert überschrieben und der vorherige Wert zum Schlüssel zurückgegeben (das ist anders als beim `Set`, wo die Operation dann nichts tut). Ist der Schlüssel neu, liefert `put(…)` den Rückgabewert `null`. Das heißt auch, dass mit `put(key, value) == null` nicht klar ist, ob `put(…)` einen Wert überschreibt und der alte Wert `null` war oder ob noch kein Schlüssel-Wert-Paar in dem Assoziativspeicher lag.
- `default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
  Speichert ein neues Schlüssel-Wert-Paar, wobei der assoziierte Wert von einer Funktion zum Zeitpunkt des Einfügens berechnet wird. War mit dem alten Wert etwas assoziiert, wird der alte Wert zurückgegeben. Ein Sonderfall ist es, wenn die Funktion `null` liefert, dann passiert nichts, bzw. ein altes existierendes Schlüssel-Wert-Paar wird gelöscht.
- `void putAll(Map<? extends K,? extends V> m)`
  Fügt alle Schlüssel-Wert-Paare aus `m` in die aktuelle `Map` ein. Auch diese Methode überschreibt unter Umständen vorhandene Schlüssel.

Über alle Werte laufen

Die Default-Methode `forEach(java.util.function.BiConsumer)` läuft über alle Schlüssel-Wert-Paare und ruft einen `BiConsumer` auf – das ist eine funktionale Schnittstelle mit einer Methode, die zwei Parameter bekommt, nämlich Schüssel und Wert. Der Konsument kann die Daten dann auswerten.

Beispiel

Die Ausgabe wird sein: »zwei=2« und »drei=3.0«.

```
Map<String,Number> num = new HashMap<>();
num.put( "zwei", 2 );
num.put( "drei", 3.0 );
BiConsumer<String,Number> action =
    (key, value) -> System.out.println( key + "=" + value );
num.forEach( action );
```

Intern greift die Methode auf `entrySet()` zurück, wir werden die Methode in Abschnitt 4.6.8 vorstellen.

Assoziierten Wert erfragen

Um wieder ein Element auszulesen, deklariert `Map` die Operation `get(key)`. Das Argument identifiziert das zu findende Objekt über den Schlüssel, indem dasjenige Objekt aus der Datenstruktur herausgesucht wird, das im Sinne von `equals(…)` mit dem Abfrageobjekt gleich ist. Wenn das Objekt nicht vorhanden ist, ist die Rückgabe `null`. Allerdings kann auch `null` der mit einem Schlüssel assoziierte Wert sein, da `null` als Wert durchaus erlaubt ist.

Beispiel

Erfrage den Assoziativspeicher nach »zwei«. Das Ergebnis wird ein `Number`-Objekt sein:

```
Map<String,Number> num = new HashMap<>();
Number number = num.get( "zwei" );
if ( number != null )
  System.out.println( number.intValue() );
```

Mit Generics kann eine Typumwandlung entfallen, wenn – wie in unserem Beispiel – `Number`-Objekte mit dem String assoziiert waren. Wurde der Typ nicht angegeben, ist eine Typumwandlung nötig.

```
interface java.util.Map<K,V>
```

- `V get(Object key)`
  Liefert das mit dem entsprechenden Schlüssel verbundene Objekt. Falls kein passendes Objekt vorhanden ist, liefert die Methode `null`.
- `default V getOrDefault(Object key, V defaultValue)`
  Existiert zum Schlüssel `key` ein assoziierter Wert, so wird dieser zurückgegeben, andernfalls der vorgegebene `defaultValue`.

Existiert der Schlüssel, existiert der Wert?

Mit `get(…)` kann nicht wirklich sicher das Vorhandensein eines Schlüssels getestet werden, da mit einem Schlüssel `null` assoziiert sein kann – das gilt zum Beispiel für `HashMap`, in der `null` als Schlüssel und Wert erlaubt sind. Eine sichere Alternative bietet die Methode `containsKey(…)`, die `true` zurückgibt, wenn ein Schlüssel in der Tabelle vorkommt.

Im Gegensatz zu `get(…)` und `containsKey(…)`, die das Auffinden eines Werts bei gegebenem Schlüssel erlauben, lässt sich auch nur nach den Werten ohne Schlüssel suchen. Dies ist allerdings wesentlich langsamer, da alle Werte der Reihe nach durchsucht werden müssen. Die Klasse bietet hierzu `containsValue(…)` an.


```
interface java.util.Map<K,V>
```

- `boolean containsKey(Object key)`
  Liefert `true`, falls der Schlüssel im Assoziativspeicher vorkommt. Den Vergleich auf Gleichheit führt `HashMap` mit `equals(…)` durch. Demnach sollte das zu vergleichende Objekt diese Methode aus `Object` passend überschreiben. `hashCode()` und `equals(…)` müssen miteinander konsistent sein. Aus der Gleichheit zweier Objekte unter `equals(…)` muss auch jeweils die Gleichheit von `hashCode()` folgen.
- `boolean containsValue(Object value)`
  Liefert `true`, falls der Assoziativspeicher einen oder mehrere Werte enthält, die mit dem Objekt inhaltlich (also per `equals(…)`) übereinstimmen.

Einträge ersetzen

Die Methode `getOrDefault(Object key, V defaultValue)` ist insofern interessant, als sie zwei Dinge tut: testen und in Abhängigkeit vom Ausgang des Tests eine Operation durchführen. Zum Austauschen bietet der `Map` drei `replaceXXX(…)`-Methoden, die nach dem gleichen Prinzip funktionieren; es muss ein Element erst existieren, bevor es ersetzt werden kann.

```
interface java.util.Map<K,V>
```

- `default V replace(K key, V value)`
  Wenn es zu `key` schon einen assoziierten Wert gab, wird dieser überschrieben. Wenn mit `key` noch nichts assoziiert war, passiert nichts. Das ist der Unterschied zu `put(…)`, das in jedem Fall ein Paar assoziiert.
- `default boolean replace(K key, V oldValue, V newValue)`
  Assoziiert `key` mit `newValue` genau dann, wenn `key` bisher mit `oldValue` assoziiert war.
- `default void replaceAll(BiFunction<? super K,? super V,? extends V> function)`
  Läuft über alle Schlüssel-Wert-Paare der Map, ruft die gegebene Funktion mit den Parametern Schlüssel und Wert auf und überschreibt den alten assoziierten Wert mit dem Ergebnis der Funktion.

Implementierungsdetail

Die Implementierung von `replaceAll(…)` ist über die Default-Methode gegeben und sieht (abgesehen von der internen Ausnahmebehandlung) so aus:

```
for ( Map.Entry<K, V> entry : map.entrySet() )
  entry.setValue( function.apply( entry.getKey(), entry.getValue() ) );
```

Einträge löschen oder die ganze Map leeren

Zum Löschen eines Elements gibt es zwei Formen von `remove(…)`, und zum Löschen der gesamten `Map` gibt es die Methode `clear()`:

```
interface java.util.Map<K,V>
```

- `V remove(Object key)`
  Löscht den Schlüssel und seinen zugehörigen Wert. Wenn der Schlüssel nicht im Assoziativspeicher ist, so bewirkt die Methode nichts. Im letzten Atemzug wird noch der Wert zum Schlüssel zurückgegeben.
- `default boolean remove(Object key, Object value)`
  Löscht den Schlüssel mit dem Wert nur dann, wenn `key` auch mit `value` assoziiert ist und nicht mit etwas anderem. Die Rückgabe ist `true`, wenn der Wert gelöscht wurde.
- `void clear()`
  Löscht den Assoziativspeicher so, dass er keine Werte mehr enthält.

Map-Operationen in Abhängigkeit von (nicht-)existierenden Werten

Die Map-API hat einige clevere Methoden, die mehrere Operationen zusammenfassen, wobei die Funktionsweise folgendem Bauplan entspricht: Ist ein assoziierter Wert zu einem Schlüssel (nicht) vorhanden, dann tue dies, sonst das.

```
interface java.util.Map<K,V>
```

- `default V putIfAbsent(K key, V value)`
  Testet zuerst, ob zu dem gegebenen Schlüssel `key` ein assoziierter Wert existiert, wenn ja, gibt es keine Änderung an der Datenstruktur, nur der alte assoziierte Wert wird zurückgegeben. Existiert kein assoziierter Wert, speichert die Datenstruktur den `value` zum Schlüssel. Die Rückgabe von `putIfAbsent(…)` ist `null`, falls es vorher keinen alten assoziierten Wert gab, andernfalls die Referenz vom alten Objekt (das auch `null` sein kann, wenn die `Map` auch `null`-Werte erlaubt), das jetzt durch den neuen Wert überschrieben wurde. Falls `null` als Wert in der `Map` erlaubt ist – wie etwa in `HashMap` –, so gilt eine Besonderheit: Ist ein existierender Schlüssel mit `null` assoziiert, dann würde `putIfAbsent(…)` den Wert `null` mit etwas anderem überschreiben.
- `default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`
  Vergleichbar mit `putIfAbsent(…)`, nur nutzt diese Methode eine Berechnungsmethode statt eines festen Werts. Ein wichtiger Punkt ist, dass, wenn die Berechnungsfunktion `null` zurückgibt, nichts an dem Assoziativspeicher verändert wird und der alte Wert bleibt. Der Rückgabewert ist immer entweder der letzte assoziierte Wert oder der neue Eintrag, es sei denn, die `mappingFunction` liefert `null`. Die Methode lässt sich damit perfekt in einer Methodenkaskadierung der Art `map.computeIfAbsent(…).`*`methodeVonV`*`(…)` verwenden.

- `default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
  Überschreibt den assoziierten Wert mit einem von der Funktion berechneten neuen Wert, wenn das Schlüssel-Wert-Paar existiert. Dann liefert die Methode den neuen Wert zurück. Zwei Sonderfälle sind zu unterscheiden: Falls es zu dem Schlüssel `key` keinen Wert gibt, macht `computeIfPresent(…)` nichts, und die Rückgabe ist `null`. Gibt es einen assoziierten Wert, doch liefert die auf den Wert angewendete Funktion `null`, wird das Schlüssel-Wert-Paar gelöscht, und die Rückgabe ist ebenfalls `null`.

Beispiel

Java bietet von Haus aus keine Datenstruktur, die wie ein Assoziativspeicher arbeitet, aber einen Schlüssel mit einer Sammlung von Werten assoziieren kann. Doch so eine Klasse ist schnell geschrieben:[8]

Listing 4.15 src/main/java/com/tutego/insel/util/map/MultimapDemo.java, Multimap

```
class Multimap<K, V> {
  private final Map<K,Collection<V>> map = new HashMap<>();

  public Collection<V> get( K key ) {
    return map.getOrDefault( key, Collections.<V> emptyList() );
  }

  public void put( K key, V value ) {
    map.computeIfAbsent( key, k -> new ArrayList<>() )
       .add( value );
  }
}
```

Ein kleines Beispiel:

```
Multimap<Integer,String> mm = new Multimap<>();
System.out.println( mm.get( 1 ) ); // []
mm.put( 1, "eins" );
System.out.println( mm.get( 1 ) ); // [eins]
mm.put( 1, "one" );
System.out.println( mm.get( 1 ) ); // [eins, one]
```

```
interface java.util.Map<K,V>
```

- `default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`

8 Und so etwas ist Teil jeder Java-API-Erweiterung, wir kommen darauf noch zu sprechen.

Setzt einen neuen Eintrag in die Map oder verschmilzt einen existierenden Eintrag mit der angegebenen Funktion. Von der Semantik her ist das die komplexeste Methode. Der erste Fall ist einfach, denn wenn es zum Schlüssel keinen Wert gibt, dann wird das Schlüssel-Wert-Paar in die `Map` gesetzt, und `merge(…)` liefert als Rückgabe den `value`. Gibt es schon einen assoziierten Wert, wird die Funktion mit dem alten Wert und `value` aufgerufen (eine `BiFunction` hat zwei Parameter) und der alte assoziierte Wert mit diesem neuen Wert überschrieben, woraufhin die Rückgabe von `merge(…)`diesen neuen Wert liefert. Jetzt gibt es noch zwei Sonderfälle, und die hängen damit zusammen, wenn das Argument `value` gleich `null` ist oder die Funktion `null` liefert. In beiden Fällen wird das Schlüssel-Wert-Paar gelöscht, und die Rückgabe von `merge(…)` ist `null`.

Beispiel

Zu einer ID soll ein Punkt assoziiert werden. Neue, zu dieser ID hinzugefügte Punkte sollen die Koordinate des ursprünglichen Punktes verschieben.

```
Map<Integer,Point> map = new HashMap<>();
BiFunction BiFunction<Point, Point, Point> remappingFunc =
    (oldVal, val) -> { val.translate( oldVal.x, oldVal.y ); return val; };
map.merge( 1, new Point( 12, 3 ), remappingFunc );
System.out.println( map.get( 1 ) );  // java.awt.Point[x=12,y=3]
map.merge( 1, new Point( -2, 2 ), remappingFunc );
System.out.println( map.get( 1 ) );  // java.awt.Point[x=10,y=5]
map.merge( 1, new Point( 0, 5 ), remappingFunc );
System.out.println( map.get( 1 ) );  // java.awt.Point[x=10,y=10]
```

Tipp

Ganz nützlich ist für Ganzzahlen auch `Integer::sum`, etwa wie in `map.merge(key, 1, Integer::sum)`. Ist die `map` leer, bekommt der `key` den assoziierten Wert 1. Andernfalls addiert die `sum(…)`-Methode zum existierenden Wert 1 hinzu.

Größe und Leertest

Mit `size()` lässt sich die Anzahl der Werte im Assoziativspeicher erfragen. `isEmpty()` entspricht einem `size() == 0`, gibt also `true` zurück, falls der Assoziativspeicher keine Elemente enthält.

String-Repräsentation, Gleichheitstest, Hashwert und Klon eines Assoziativspeichers *

Viele Methoden von `Object` überschreiben die Assoziativspeicher-Klassen. `toString()` auf Assoziativspeichern liefert eine Zeichenkette, die den Inhalt der Sammlung aufzeigt. Die String-Repräsentation liefert jeden enthaltenen Schlüssel, gefolgt von einem Gleichheitszeichen und dem zugehörigen Wert. Entwickler sollten nie diese Zeichenkennung parsen bzw. irgendwelche Annahmen über die Formatierung machen.

Beispiel

Ein Assoziativspeicher soll die Zahlen 1, 2, 3 jeweils mit ihrem Quadrat assoziieren. Zum Aufbau benutzen wir eine fortgeschrittene Technik:

```
Map<Integer,Integer> map = Stream.iterate( 1, Math::incrementExact )
                                 .limit( 3 )
                                 .collect( Collectors.toMap( id -> id, id -> id*id) );
System.out.println( map );   // {1=1, 2=4, 3=9}
```

Aus `Object` überschreiben die Standardimplementierungen die Methoden `equals(…)` und `hashCode()`.

Die Klassen `HashMap` (und Unterklasse `LinkedHashMap`), `IdentityHashMap`, `TreeMap`, `ConcurrentSkipListMap` und `EnumMap` deklarieren eine öffentliche `clone()`-Methode, die eine Kopie eines Assoziativspeichers erzeugt. Die Kopie bezieht sich allerdings nur auf den Assoziativspeicher selbst; die Schlüssel- und Wert-Objekte teilen sich Original und Klon. Diese Form der Kopie nennt sich auch *flache Kopie* (engl. *shallow copy*). Eine Veränderung an den enthaltenen Schlüssel-Wert-Objekten betrifft also immer beide Datenstrukturen, und eine unsachgemäße Modifikation kann zu Unregelmäßigkeiten im Original führen. Eine `ConcurrentHashMap` oder `WeakHashMap` unterstützt kein `clone()`, und eigentlich ist `clone()` überhaupt nicht nötig, denn die Konstruktoren der Datenstrukturen können immer eine andere Datenstruktur als Vorlage nehmen, etwa `clone = new ConcurrentHashMap(existingMap)`.

```
interface java.util.Map<K,V>
```

- `boolean equals(Object o)`
  Damit die Gleichheit von zwei Assoziativspeichern gezeigt werden kann, vergleicht `equals(…)` alle Elemente von beiden Sammlungen.
- `int hashCode()`
  Liefert den Hashwert des Assoziativspeichers. Das ist wichtig, wenn die Sammlung selbst als Schlüssel benutzt wird – was jedoch als problematisch gelten kann, wenn der (grundsätzlich mutable) Assoziativspeicher später noch verändert werden soll.[9]

```
class java.util.HashMap<K,V> …
class java.util.TreeMap<K,V> …
```

9 Fast schon philosophisch wird's, wenn eine Hash-Tabelle als Schlüssel oder Wert in sich selbst eingefügt werden soll. Das kann sie zwar noch erkennen, aber bei `Map<Object,String> h = new HashMap<>(); h.put(h, "a"); h.put(h, "b");` gibt es einen `StackOverflowError`, und damit ist die Philosophie am Ende. Mit Generics fallen Fehler wie diese schneller auf. Interessanterweise fängt ein `HashSet` diesen Sonderfall ab, und `Set<Object> s = new HashSet<>(); s.add(s);` führt zu keinem Fehler.

- `Object clone()`
  Fertigt eine Kopie an, ohne jedoch die Werte selbst zu klonen.

4.6.3 Über die Bedeutung von equals(...) und hashCode() bei Elementen

Wenn wir Assoziativspeicher wie eine `HashMap` nutzen, dann sollte uns bewusst sein, dass Vergleiche nach dem Hashwert und der Gleichheit durchgeführt werden, nicht aber nach der Identität. Die folgenden Zeilen zeigen ein Beispiel:

Listing 4.16 src/main/java/com/tutego/insel/util/map/HashMapAndEquals.java(), main()

```
Map<Point,String> map = new HashMap<>();
Point p1 = new Point( 10, 20 );
map.put( p1, "Point p1" );
```

Die `HashMap` assoziiert den Punkt `p1` mit einer Zeichenkette. Was ist nun, wenn wir ein zweites Punkt-Objekt mit den gleichen Koordinaten bilden und die `Map` nach diesem Objekt fragen?

```
Point p2 = new Point( 10, 20 );
System.out.println( map.get( p2 ) );                    // ???
```

Die Antwort ist die Zeichenfolge »Point p1«. Das liegt daran, dass zunächst der Hashwert von `p1` und `p2` gleich ist. Des Weiteren liefert auch `equals(…)` ein `true`, sodass dies als ein Fund zu werten ist (das liefert noch einmal einen wichtigen Hinweis darauf, dass immer beide Methoden, `equals(…)` und `hashCode()`, in Unterklassen zu überschreiben sind).

Mit etwas Überlegung folgt dieser Punkt fast zwangsläufig, denn bei einer Abfrage ist ja das zu erfragende Objekt nicht bekannt. Daher kann der Vergleich nur auf Gleichheit, nicht aber auf Identität stattfinden.

4.6.4 Eigene Objekte hashen

Für Objekte, die als Schlüssel in einen Hash-Assoziativspeicher gesetzt werden, gibt es keine Schnittstelle zu implementieren, lediglich die Aufforderung, dass `equals(…)` und `hashCode()` in geeigneter Weise (also der Bedeutung oder Semantik des Objekts entsprechend) untereinander konsistent implementiert sein sollen. Wenn `equals(…)` von zwei Objekten gleichen Typs zum Beispiel `true` ergibt, muss der Hashwert auch gleich sein, und wenn zwei Hashwerte ungleich sind, darf `equals(…)` nicht wahr ergeben. Viele Standardklassen, wie `String` oder `Point`, erfüllen diese Anforderung, andere, wie `StringBuilder`, implementieren kein eigenes `hashCode()` und `equals(…)`. Ein weiteres Problem ist, dass bei Vergleichen mit Ordnung ein externer Comparator eingesetzt werden kann, aber die Logik für einen Test auf Gleichheit und die Berechnung eines Hashwerts lassen sich nicht in ein externes Objekt setzen. Für Schlüsselobjekte in einer `NavigableMap`/`SortedMap` ist `hashCode()` nicht erforderlich, da es hier auf die Ordnung ausdrücklich ankommt.

Wir wollen eine Klasse entwerfen, die `hashCode()` und `equals(…)` so implementiert, dass Strings unabhängig von ihrer Groß-/Kleinschreibung einsortiert und gefunden werden:

Listing 4.17 src/main/java/com/tutego/insel/util/map/EqualsIgnoreCaseString.java

```
package com.tutego.insel.util.map;

public class EqualsIgnoreCaseString {

  private final String string;

  public EqualsIgnoreCaseString( String string ) {
    this.string = string.toLowerCase();
  }

  @Override public int hashCode() {
    return string.hashCode();
  }

  @Override public boolean equals( Object obj ) {
    if ( this == obj )
      return true;
    if ( obj == null )
      return false;
    if ( getClass() != obj.getClass() )
      return false;
    return string.equals( ((EqualsIgnoreCaseString) obj).string );
  }
}
```

Ein kleiner Test mit den Rückgaben im Kommentar:

Listing 4.18 src/main/java/com/tutego/insel/util/map/EqualsIgnoreCaseStringDemo.java, main()

```
Map<EqualsIgnoreCaseString, String> map = new HashMap<>();
map.put( new EqualsIgnoreCaseString("tutego"), "tutego" ); // null
map.put( new EqualsIgnoreCaseString("Tutego"), "Tutego" ); // tutego
map.put( new EqualsIgnoreCaseString("TUTI!"), "TUTI!" );   // null

map.get( new EqualsIgnoreCaseString("tutego") );           // Tutego
map.get( new EqualsIgnoreCaseString("TUTEGO") );           // Tutego
map.get( new EqualsIgnoreCaseString("tUtI!") );            // TUTI!
map.get( new EqualsIgnoreCaseString("tröt") );             // null
```

4.6.5 LinkedHashMap und LRU-Implementierungen

Da die Reihenfolge der eingefügten Elemente bei einem Assoziativspeicher verloren geht, gibt es mit `LinkedHashMap` eine Mischung, also einen schnellen Assoziativspeicher mit gleichzeitiger Speicherung der Reihenfolge der Objekte. Die Bauart des Klassennamens `LinkedHashMap` macht schon deutlich, dass es eine `Map` ist, und die Reihenfolge der Objekte liefert einen `Iterator`; es gibt keine listenähnliche Schnittstelle mit `get(int)`. `LinkedHashMap` ist für Assoziativspeicher das, was `LinkedHashSet` für `HashSet` ist.

Im Gegensatz zur normalen `HashMap` ruft `LinkedHashMap` immer genau dann die besondere Methode `boolean removeEldestEntry(Map.Entry<K,V> eldest)` auf, wenn intern ein Element der Sammlung hinzugenommen wird. Die Standardimplementierung dieser Methode liefert immer `false`, was bedeutet, dass das älteste Element nicht gelöscht werden soll, wenn ein neues hinzukommt. Doch bietet das JDK die Methode aus Absicht `protected` an, denn sie kann von uns überschrieben werden, um eine Datenstruktur aufzubauen, die eine maximale Anzahl an Elementen hat.

So sieht das aus:

Listing 4.19 src/main/java/com/tutego/tutego/insel/util/map/LRUMap.java

```
package com.tutego.insel.util.map;

import java.util.*;

public class LRUMap<K,V> extends LinkedHashMap<K,V> {
  private final int capacity;

  public LRUMap( int capacity ) {
    super( capacity, 0.75f, true );
    this.capacity = capacity;
  }

  @Override
  protected boolean removeEldestEntry( Map.Entry<K,V> eldest ) {
    return size() > capacity;
  }
}
```

`LinkedHashSet` bietet eine vergleichbare Methode `removeEldestEntry(…)` nicht. Wer diese benötigt, muss eine eigene Mengenklasse auf der Basis von `LinkedHashMap` realisieren.

[+]

Tipp

Wer einen Cache benötigt, der findet zum Beispiel in dem quelloffenen *Caffeine* (*https://github.com/ben-manes/caffeine*) eine deutlich leistungsfähigere Lösung.

4.6.6 IdentityHashMap

Es gibt eine besondere Datenstruktur mit dem Namen `IdentityHashMap`, die statt der internen `equals(…)`-Vergleiche einen Identitätsvergleich mit `==` durchführt. Die Implementierung ist selten im Einsatz, kann aber im Bereich der Performance-Optimierung eine interessante Rolle übernehmen und auch das Problem lösen, wenn in der `Map` denn absichtlich Objekte enthalten sein sollen, die `equals`-gleich, aber nicht identisch sind. Es lässt sich auch so sehen: `IdentityHashMap` ist attraktiv, wenn als Schlüssel Objekte zum Einsatz kommen, bei denen Gleichheit und Identität dasselbe bedeuten.

[+]

Hinweis

An `Integer`-Objekten in einer `IdentityHashMap` zeigt sich genau der Unterschied zur klassischen Map wie einer `HashMap`. Nehmen wir

```
Map<Integer,String> map = new IdentityHashMap<>();
map.put( 1, "1" );
map.put( 1000, "1000" );
System.out.printf( "1=%s, 1000=%s", map.get( 1 ), map.get( 1000 ) );
```

Die Ausgabe ist »1=1, 1000=null«. Wegen des Autoboxings führt der Compiler ein `Integer.value(…)` ein. Und wegen des internen `Integer`-Caches liegt das `Integer`-Objekt für 1 genau einmal vor – wir erinnern uns: Die `Integer`-Klasse cacht standardmäßig `Integer`-Objekte für Ganzzahlen im Wertebereich eines `byte`. Für die Zahl 1.000 führt `Integer.value(1000)` jedoch zu zwei nicht identischen `Integer`-Objekten. Abfragen mit `equals`-gleichen `Integer`-Objekten führen in der `HashMap` natürlich zu einem Treffer, laufen jedoch bei einer `IdentityHashMap` ins Leere, wenn die Objekte zwar gleich, aber nicht identisch sind.

4.6.7 Das Problem veränderter Elemente

Ein Hashwert ergibt sich aus den Attributen eines Objekts. Um ein Objekt in einem Assoziativspeicher zu finden, wird dann nach dem Hashwert gesucht; dumm, wenn sich dieser in der Zwischenzeit geändert hat:

Listing 4.20 src/main/java/com/tutego/insel/util/map/MapImmutable.java, main()

```
Map<Point,String> map = new HashMap<>();
Point q = new Point( 1, 1 );
map.put( q, "Punkt q" );
q.x = 12345;
System.out.println( map.get( q ) ); // ???
q.x = 1;
System.out.println( map.get( q ) ); // ???
```

Nach der Zuweisung an `x` wird `hashCode()` einen anderen Wert als vorher liefern. Wenn nun `get(…)` nach dem Objekt sucht, berechnet es den Hashwert und sucht in den internen Datenstrukturen. Ändert sich der Hashwert jedoch unterdessen, kann das Element nicht mehr gefunden werden und liegt als Leiche in der Map. Daher kann nur davor gewarnt werden, Attribute von Objekten, die durch Assoziativspeicher verwaltet werden, nachträglich zu ändern. Das Prinzip Hashing benutzt gerade diese Eigenschaft, um Objekte durch unveränderte Zustände wiederzufinden.

4.6.8 Aufzählungen und Ansichten des Assoziativspeichers

Eine `Map` kann beim erweiterten `for` nicht rechts vom Doppelpunkt stehen, da sie kein `Iterable` implementiert – nicht direkt und, da eine `Map` keine `Collection` ist, auch nicht indirekt. Auch fehlt der `Map` irgendeine direkte Methode `iterator()`.

Eine `Map` kann auf drei Arten Sammlungen zurückgeben, über die sich iterieren lässt:

- `keySet()` liefert eine Menge der Schlüssel.
- `values()` liefert eine `Collection` der Werte.
- `entrySet()` liefert eine Menge mit speziellen `Map.Entry`-Objekten. Die `Map.Entry`-Objekte speichern gleichzeitig den Schlüssel sowie den Wert.

Für die Sammlungen gibt es erst einmal keine definierte Reihenfolge beim Auflaufen, es sei denn, die `Map` ist eine `NavigableMap`, wo ein `Comparator` die Ordnung vorgibt oder die Elemente `Comparable` sind.

Beispiel

Laufe die Schlüssel einer `HashMap` mit einem Iterator (über das erweiterte `for`) ab:

```
Map<String,String> h = new HashMap<>();
h.put( "C.Ullenboom", "c.ullenboom@no-spam.com" );
h.put( "Webmaster", "c.ullenboom@spammer.com" );
h.put( "Weihnachtsmann", "wunsch@weihnachtsmann.com" );
h.put( "Christkind", "wunsch@weihnachtsmann.com" );
for ( String elem : h.keySet() )
  System.out.println( elem );
```

Liefert:

```
Weihnachtsmann
C.Ullenboom
Christkind
Webmaster
```



```
interface java.util.Map<K,V>
```

- `Set<K> keySet()`
  Liefert eine Menge mit den Schlüsseln.
- `Set<Map.Entry<K,V>> entrySet()`
  Liefert eine Menge von `Map.Entry`-Objekten, die Zugriff auf die Schlüssel und Werte bieten.
- `Collection<V> values()`
  Liefert eine Sammlung der Werte.

Da eine `Map` immer typisiert ist, gilt das natürlich auch für die Sichten auf die Daten.

keySet() und values()

`keySet()` liefert eine Sammlung als Menge, da die Schlüssel eines Assoziativspeichers immer eindeutig sind.

Beispiel

Es ist zu erfragen, ob sich in zwei Assoziativspeichern `map1` und `map2` die gleichen Schlüssel befinden – unabhängig vom Wert:

```
boolean sameKeys = map1.keySet().equals( map2.keySet() );
```

Für eine Sammlung aller Werte gibt es *keine* `Set`-liefernde Methode `valueSet()`, weil ein Wert mehr als einmal vorkommen kann, wie auch unser Beispiel zeigt: `"Weihnachtsmann"` ist assoziiert mit `"wunsch@weihnachtsmann.com"` und `"Christkind"` ist ebenfalls assoziiert mit `"wunsch@weihnachtsmann.com"`. Daher heißt die Methode für alle assoziierten Werte einfach `values()`; die Methode liefert die spezielle `Collection` mit den Werten. Ein Aufruf von `iterator()` auf dieser `Collection` bietet dann eine Aufzählung dieser Werte. Über den Iterator oder die `Collection` können Elemente aus der `Map` gelöscht, aber keine neuen eingefügt werden.

Beispiel

Laufe die Werte einer `HashMap` mit einem Iterator ab:

```
for ( String elem : h.values() )
  System.out.println( elem );
```

Das liefert:

```
wunsch@weihnachtsmann.com
c.ullenboom@no-spam.com
wunsch@weihnachtsmann.com
c.ullenboom@spammer.com
```

Map.Entry

Während `keySet()` nur die eindeutigen Schlüssel in einer Menge liefert und die assoziierten Elemente in einem zweiten Schritt geholt werden müssten, gibt `entrySet()` ein `Set` von Elementen, typisiert mit `Map.Entry`, zurück. `Entry` ist eine innere Schnittstelle von `Map`, die eine API zum Zugriff auf Schlüssel-Wert-Paare deklariert. Die wichtigen Operationen dieser Schnittstelle sind `getKey()`, `getValue()` und `setValue()`, wobei die letzte Methode optional ist, aber etwa von `HashMap` angeboten wird. Neben diesen Methoden überschreibt `Entry` auch `hashCode()` und `equals(…)`.

Beispiel

Laufe die Elemente `HashMap` als Menge von `Map.Entry`-Objekten ab:

```
for ( Map.Entry<String,String> e : h.entrySet() )
  System.out.println( e.getKey() + "=" + e.getValue() );
```

`Map.Entry` ist eher ein Internum, und die Objekte dienen nicht der langfristigen Speicherung. Ein `entrySet()` ist eine Momentaufnahme, und das Ergebnis sollte nicht referenziert werden, denn ändert sich der darunterliegende Assoziativspeicher, ändern sich auch die `Entry`-Objekte, und das `Set<Map.Entry>` als Ganzes ist vielleicht nicht mehr gültig. `Entry`-Objekte sind nur gültig im Moment der Iteration, was den Nutzen einschränkt. Daher ist die Rückgabe von `entrySet()` mit `Set<Map.Entry<K,V>>` auch relativ unspezifisch, und es ist unklar, um was für eine Art von Set es sich genau handelt; ob `HashSet` oder vielleicht `NavigableSet`, spielt keine Rolle.

Auch wenn die `Map.Entry`-Objekte nicht für die Speicherung gedacht sind, können Sie in einem Strom von Daten verarbeitet und in einer zustandsbehafteten Operation sortiert werden. Der Bonus der `Entry`-Objekte im Strom ist einfach, dass es von Vorteil ist, Schüssel und Wert in einem Objekt gekapselt zu sehen. Aber was ist, wenn jetzt der Strom sortiert werden soll, etwa nach dem Schlüssel oder dem Wert? Hier kommen spezielle Methoden ins Spiel, die den nötigen `Comparator` liefern.

Beispiel *

Erfrage eine Menge von `Entry`-Objekten, und sortiere sie nach dem assoziierten Wert:

```
map.entrySet()
   .stream()
   .sorted( Map.Entry.<String,String>comparingByValue() )
   .forEach( System.out::println );
```

```
static interface Map.Entry<K,V>
```

- `static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey()`
- `static <K extends Comparable<? super V>,V> Comparator<Map.Entry<K,V>> comparingByValue()`
- `static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp)`
- `static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp)`

Verändernde Ansichten

Allen Methoden ist gemeinsam, dass sie nur eine andere Sicht auf die Originalmenge darstellen. Wir müssen uns dessen bewusst sein, dass Löschoperationen die ursprüngliche Menge verändern. Mit anderen Worten: Die von `keySet()`, `values()` oder `entrySet()` zurückgegebenen Sammlungen sind verschiedene Ansichten des Originals, und Veränderungen wirken sich unmittelbar auf das Original aus:

Listing 4.21 src/main/java/com/tutego/insel/util/map/MapView.java, main()

```
Map<Integer,String> m = new HashMap<>();
m.put( 1, "Eins" );
m.put( 2, "ZZZZZWWWWEEEEEIIII" );
m.put( 3, "drei" );
System.out.println( m );     // {1=Eins, 2=ZZZZZWWWWEEEEEIIII, 3=drei}

m.keySet().remove( 2 );
System.out.println( m );     // {1=Eins, 3=drei}

m.values().remove( "Eins" );
System.out.println( m );     // {3=drei}

m.entrySet().clear();
System.out.println( m );     // {}
```

4.6.9 Die Arbeitsweise einer Hash-Tabelle *

Für Assoziativspeicher gibt es unterschiedliche Implementierungen; eine basiert auf sortierten Bäumen, eine andere nutzt eine besondere mathematische Funktion, sodass wir bei diesen Assoziativspeichern von *Hash-Tabellen* sprechen. Insgesamt setzt Java bei mehreren Datenstrukturen auf das Hashing-Verfahren; die Klassen sind durch das Wort »Hash« zu identifizieren:

- `Hash`Map
- `Hash`table
- `Hash`Set
- Linked`Hash`Map
- Linked`Hash`Set
- Weak`Hash`Map
- Concurrent`Hash`Map

Die Arbeitsweise ist wie folgt: Aus dem Schlüssel wird nach einer mathematischen Funktion – der so genannten *Hash-Funktion* – ein *Hashwert* (auch *Hashcode*) berechnet. Dieser dient dann als Index für ein internes Array. Dieses Array hat am Anfang eine feste Größe. Wenn später eine Anfrage nach dem Schlüssel gestellt wird, muss einfach diese Berechnung erfolgen, und wir können dann an dieser Stelle nachsehen. Wir können uns eine einfache Hash-Funktion für folgendes Problem denken: Beliebige Zeichenketten sollen in der Hash-Tabelle abgelegt werden. Die Hash-Funktion summiert einfach alle ASCII-Werte der Buchstaben und nimmt sie modulo 77. Dann können in einem Array mit 77 Elementen 77 verschiedene Wörter aufgenommen werden. Leider hat diese Technik einen entscheidenden Nachteil: Wenn zwei unterschiedliche Wörter denselben Hashwert besitzen, kommt es zu einer Kollision. Darauf muss die Datenstruktur vorbereitet sein. Hier gibt es verschiedene Lösungsansätze. Die unter Java implementierte Lösung referenziert eine Datenstruktur hinter jedem Array-Element (einen so genannten *Bucket*); diese Implementierungsvariante heißt *Hashing mit Verkettung*. Falls eine Kollision auftritt, wird dieses Element in die Datenstruktur gesetzt. Oftmals ist es eine simple lineare Liste (ein Array), in der aktuellen Java SE ist es eine Kombination aus Baum und Liste.

Der Füllfaktor und die Konstruktoren

Um ein Maß für den Füllgrad zu bekommen, wird ein *Füllfaktor* (Füllgrad; engl. *load factor*) eingeführt. Dieser liegt zwischen 0 % und 100 %. Ist er 0 %, so bedeutet dies, dass die Hash-Tabelle leer ist; ist er 100 %, so enthält die Hash-Tabelle genauso viele Einträge, wie das interne Array Elemente umfasst. Die Verteilung der Einträge auf die Array-Elemente wird dabei in der Regel ungleichmäßig sein. Einige Array-Elemente enthalten bereits (kurze) Listen mit kollidierenden Einträgen, während andere Array-Elemente noch unbenutzt sind. Der Füllfaktor einer Hash-Tabelle sollte für einen schnellen Zugriff nicht höher als 75 % sein. Das heißt, ein Viertel der Array-Elemente wird grundsätzlich nicht belegt.

Der Füllfaktor gibt also an, wie »voll« die Hash-Tabelle ist. Es lässt sich nun einstellen, dass die Hash-Tabelle sich automatisch vergrößert, damit der Zugriff wieder schneller wird. Dazu ordnen wir dem Füllgrad einen Prozentwert als Fließkommazahl zwischen 0,0 und 1,0 zu. Ein Wert von 0,75 entspricht also dem oben angesprochenen idealen Füllgrad von 75 %. Es gibt einen Konstruktor für `HashMap`/`Hashtable`-Exemplare, der die Angabe eines Füllgrads erlaubt. Ist dieser überschritten, wird die Hash-Tabelle neu berechnet. Dies nennt sich *Re-Hashing*. Dazu wird eine neue Hash-Tabelle angelegt, deren Array größer als das alte ist. Jeder Wert aus der alten Hash-Tabelle wird dabei gemäß der Hash-Funktion an die passende Stelle in das größere Array eingefügt. Ist dies für alle Elemente geschehen, wird die alte Hash-Tabelle gelöscht. Dieses Kopieren und Neuberechnen dauert zwar einige Zeit, doch direkt danach lassen sich die Anfragen an die Datenstruktur wieder schnell beantworten. Wenn die Hash-Tabelle zu oft vergrößert und neu organisiert werden muss, ist dies natürlich ein gewaltiger Geschwindigkeitsnachteil. Doch durch die Vergrößerung wird der Zugriff wieder schneller. Das Re-Hashing kann nicht ausdrücklich erzwungen werden.


```
class java.util.HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

- `HashMap()`
  Die Hash-Tabelle enthält initial eine Kapazität von 16 freien Plätzen und einen Füllfaktor von 75 %, also 0,75.
- `HashMap(int initialCapacity)`
  Erzeugt eine Hash-Tabelle mit einer vorgegebenen Kapazität und dem Füllfaktor 0,75.
- `HashMap(int initialCapacity, float loadFactor)`
  Erzeugt eine Hash-Tabelle mit einer vorgegebenen Kapazität und dem angegebenen Füllfaktor.

Die anfängliche Größe des internen Arrays lässt sich in zwei Konstruktoren angeben; ein unsinniger Wert löst eine `IllegalArgumentException` aus. Zusätzlich kann der Füllfaktor angegeben werden; ist dieser falsch, wird diese Exception ebenfalls ausgelöst. `initialCapacity` muss größer als die geplante Nutzlast der Hash-Tabelle gewählt werden. Das heißt, bei geplanten 1.000 Einträgen ist `initialCapacity` etwa 1.000 × (1/0,75) = 1.333. Ist ein Füllfaktor nicht explizit angegeben, wird die Hash-Tabelle dann vergrößert und neu organisiert, wenn die Anzahl der Einträge in der Hash-Tabelle größer gleich 0,75 × *Größe des Arrays* ist.

Auch die Konstruktoren von `HashSet` ermöglichen die Angabe des Füllfaktors und der Initialgröße.

```
class java.util.HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

- `HashSet()`
  Erzeugt ein neues `HashSet`-Objekt mit 16 freien Plätzen und einem Füllfaktor von 0,75.
- `HashSet(int initialCapacity)`
  Erzeugt ein neues `HashSet` mit einer gegebenen Anzahl freier Plätze und dem Füllfaktor von 0,75.
- `HashSet(int initialCapacity, float loadFactor)`
  Erzeugt ein neues, leeres `HashSet` mit einer Startkapazität und einem gegebenen Füllfaktor.

Die Startgröße ist für die Performance wichtig. Ist die Größe zu klein gewählt, muss die Datenstruktur bei neu hinzugefügten Elementen vergrößert werden – hier unterscheidet sich die Klasse `HashSet` nicht von der Klasse `HashMap`, da `HashSet` intern auf `HashMap` basiert.

Kollisionen und Hash-Funktionen

Die Wahl der richtigen Hash-Funktion ist wichtig für die Performance. Denn eine »dumme« Hash-Funktion, die beispielsweise alle Schlüssel nur auf einen konstanten Wert abbildet, erreicht keine Verteilung, sondern lediglich eine lange Liste von Schlüssel-Wert-Paaren; diese Anhäufung an einer Stelle nennt sich *Clustering*. Doch auch bei der besten Verteilung über *n* Buckets ist nach dem Einfügen des Elements *n* + 1 irgendwo eine Liste mit mindestens zwei Elementen aufgebaut, daher vergrößert die Bibliothek standardmäßig das Array. Ist aber die Hash-Funktion so schlecht, dass alles auf das gleiche Bucket abgebildet wird, hilft dieses Re-Hashing auch nicht.

Je länger die Datenstruktur der miteinander kollidierenden Einträge wird, desto langsamer wird der Zugriff der auf Hashing basierenden Datenstruktur insgesamt. Java greift beim Hashing auf die `hashCode()`-Methode zurück, und es liegt in unserer Hand, sie gut zu implementieren. Die Klasse `String` hat eine relativ einfache (dafür schnelle) Implementierung von `hashCode()`, die in der Vergangenheit zu Denial-of-Service-Attacken führte, gerade weil es zu Kollisionen kam.

4.6.10 Die Properties-Klasse

Die Klasse `java.util.Properties` ist eine Sonderform der Assoziativspeicher, bei der Schlüssel-Wert-Paare immer vom Typ `String` sind. Ein Assoziativspeicher, der Strings mit Strings verbindet, ist auch `HashMap<String,String>`, doch die Klasse `Properties` kann noch mehr: Sie kann Properties hierarchisch verketten und Einträge in einer Datei speichern und wieder auslesen. Auf diese Weise lassen sich fest verdrahtete Zeichenketten aus dem Programmtext externalisieren, sodass sich die Werte auch ohne Neuübersetzung bequem verändern lassen.

Im Folgenden schauen wir uns an, wie ein Properties-Objekt aufgebaut, gefüllt und erfragt wird. Die Speicherung und das Laden mithilfe von `Properties` gehören thematisch zu den Datenformaten und sind Bestandteil von Kapitel 9, »Dateiformate«.

Properties setzen und lesen

Die Methode `setProperty(String, String)` fügt dem `Properties`-Objekt ein Schlüssel-Wert-Paar hinzu. Um später wieder an den Wert zu kommen, wird `getProperty(String)` mit dem Schlüssel aufgerufen und liefert dann – wenn beide Zeichenketten vorher verbunden wurden – den Wert:

```
Properties props = new Properties();
props.setProperty( "User", "Eddie Grundies" );
props.setProperty( "Version", "0.02" );
System.out.println( props.getProperty("User") );      // Eddie Grundies
System.out.println( props.getProperty("Passwort") );  // null
```


Von der Klassendeklaration her erweitert `Properties` die Klasse `Hashtable<Object, Object>`, und da `Hashtable` die Schnittstelle `Map` implementiert, ist `Properties` auch vom Typ `Map`. Die `Map`-Methoden werden jedoch nicht eingesetzt.

Properties verketten

`Properties`-Objekte lassen sich hierarchisch verbinden, sodass im Fall einer erfolglosen Suche nach einem Schlüssel das `Properties`-Objekt die Anfrage an ein übergeordnetes `Properties`-Objekt weiterleitet; das Eltern-`Properties`-Objekt wird einfach im Konstruktor übergeben:

Listing 4.22 src/main/java/com/tutego/insel/util/map/PropertiesDemo.java. main()

```
Properties defaultProperties = new Properties(),
           userProperties    = new Properties( defaultProperties );
```

Die Zeilen erzeugen zwei `Properties`-Objekte. Obwohl am Anfang beide leer sind, werden doch die in `defaultProperties` hinzugefügten Einträge auch in `userProperties` sichtbar sein. Im Folgenden ist abzulesen, wie `userProperties` einen Eintrag überschreibt:

```
defaultProperties.setProperty( "User", "Bill Grundies" );
defaultProperties.setProperty( "Password", "(nicht gesetzt)" );
userProperties.setProperty( "Password", "SagIchNet" );
```

Zuerst durchsucht ein `Property`-Exemplar die eigene Datenstruktur. Liefert diese `Property` keinen Eintrag oder keinen Wert vom Typ `String`, so wird das im Konstruktoraufruf angegebene `Property`-Objekt durchsucht. Auf diese Weise lassen sich mehrstufige Hierarchien von `Property`-Verzeichnissen konstruieren.

```
class java.util.Properties
extends Hashtable<Object,Object>
```

- `Properties()`
  Erzeugt ein leeres `Properties`-Objekt ohne Schlüssel und Werte.
- `Properties(Properties defaults)`
  Erzeugt ein leeres `Properties`-Objekt, das bei Anfragen auch auf die Einträge in dem übergebenen `Properties`-Objekt zurückgreift.
- `String getProperty(String key)`
  Sucht in den `Properties` nach der Zeichenkette `key` als Schlüssel und liefert den zugehörigen Wert. Durchsucht auch übergeordnete `Properties`-Objekte.
- `String getProperty(String key, String default)`
  Sucht in den `Properties` nach der Zeichenkette `key` als Schlüssel und liefert den zugehörigen Wert. Ist der Schlüssel nicht vorhanden, wird der String `default` zurückgegeben.

- `Object setProperty(String key, String value)`
  Trägt Schlüssel und Wert im `Properties`-Exemplar ein. Existiert der Schlüssel schon, wird er überschrieben. Mitunter verdeckt der Schlüssel den Wert der `Property` in der übergeordneten `Property`.
- `void Enumeration<?> propertyNames()`
  Liefert eine `Enumeration` aller Schlüssel in der `Properties`-Liste inklusive der Standardwerte aus übergeordneten Properties.

Hierarchische Eigenschaften

Leider kann eine Eigenschaftendatei nicht segmentiert werden, wie etwa alte Windows-INI-Dateien dies machen. Die Alternative besteht darin, hierarchisch benannte Eigenschaften zu erzeugen, indem eine Zeichenkette vor jeden Schlüssel gesetzt wird. Um zum Beispiel einen Schlüssel `User` einmal unter `Private` und einmal unter `Public` zu halten, lassen sich die Eigenschaften `Private.User` und `Public.User` einsetzen. Doch leider tauchen sie nach dem Speichern durcheinandergewürfelt in der Datei auf, weil der Assoziativspeicher keine Sortierung besitzt (`Properties` basiert nicht auf `TreeMap`).

Eigenschaften auf der Konsole ausgeben *

Die Methoden `list(PrintStream)` bzw. `list(PrintWriter)` wandern durch die Daten eines `Properties`-Exemplars und schreiben sie in den Datenstrom. Eine Ausgabe auf dem Bildschirm erhalten wir mit `list(System.out)`. Schlüssel und Werte trennt ein Gleichheitszeichen. Die Ausgabe über `list(…)` ist gekürzt, denn ist ein Wert länger als 40 Zeichen, wird er abgekürzt; daher eignet sich die Methode nicht dafür, die Belegungen gültig in eine Datei zu schreiben. Ein `list(System.out)` auf die `defaultProperties` bzw. `userProperties` von unserem vorangegangenen Beispiel ergibt folgende Ausgabe:

```
-- listing properties --
Password=(nicht gesetzt)
User=Bill Grundies
-- listing properties --
Password=SagIchNet
User=Bill Grundies
```

Den Paaren geht eine Kopfzeile der Art `-- listing properties --` voran. Es ist wichtig zu verstehen, dass durch die Art der Speicherung (ein Assoziativspeicher auf Basis des Hashings) die Ausgabe unsortiert erfolgt.

```
class java.util.Properties
extends Hashtable<Object,Object>
```


- `void list(PrintStream out)`
  Listet die Properties aus dem `PrintStream` auf.
- `void list(PrintWriter out)`
  Listet die Properties aus dem `PrintWriter` auf.

Klassenbeziehungen – Properties und Hashtable *

Die `Properties`-Klasse ist eine Erweiterung von `Hashtable`, weil die Speicherung der Einstellungsdaten in dieser Datenstruktur erfolgt. Ein `Properties`-Objekt erweitert den Assoziativspeicher um die Möglichkeit, die Schlüssel-Wert-Paare in einem festgelegten Format aus einer Textdatei bzw. einem Datenstrom zu laden und wieder zu speichern. Doch gerade weil `Hashtable` erweitert wird, sind auch alle Methoden der Klasse `Hashtable` auf ein `Properties`-Objekt anwendbar. Das ergibt nicht für alle Methoden Sinn und ist auch nicht in jedem Fall problemlos. Dass `Properties` eine Unterklasse von `Hashtable` ist, ist ähnlich fragwürdig wie die Vererbungsbeziehung von `Stack` als Unterklasse von `Vector`. So ist ein Augenmerk auf die `put(…)`-Methode zu richten. Sie gibt es in der Klasse `Properties` nicht, denn `put(…)` wird von `Hashtable` geerbt. Wir sollten sie auch nicht verwenden, da es über sie möglich ist, Objekte einzufügen, die nicht vom Typ `String` sind. Das gleiche Argument könnte für `get(…)` gelten, doch sprechen zwei Dinge dagegen: zum einen, dass wir beim `get(…)` aus einem `Hashtable`-Objekt immer ein `Object`-Objekt bekommen und daher meistens eine Typumwandlung benötigen; und zum anderen durchsucht diese Methode lediglich den Inhalt des angesprochenen `Properties`-Exemplars. `getProperties()` arbeitet da etwas anders. Nicht nur ist der Rückgabewert automatisch ein String, sondern `getProperties()` durchsucht auch übergeordnete `Properties`-Objekte mit, die zum Beispiel Standardwerte speichern.

4.7 Immutable Datenstrukturen

Nehmen wir an, ein Spieler speichert Gegenstände in einer `ArrayList`. Wollen wir diese Gegenstände in irgendeiner Form nach außen geben, bringt ein einfacher Getter das Problem mit sich, dass der Nutzer eine direkte Referenz auf das interne Attribut bekommt und Unsinn anrichten kann. Wenn die direkte Rückgabe etwa lautet:

```
private List<String> items = new ArrayList<>();
public List<String> getItems() {
  return items;
}
```

dann kann der Nutzer `getItems().clear()` aufrufen, und alle Daten sind futsch, und ein `((List)getItems()).add(int.class)` führt in der Folge sicherlich zu tollen Überraschungen.

Eine Nur-Lese-Schnittstelle von `List` oder `Collection` gibt es nicht, und selbst ein `Iterator` hilft nicht viel, denn er hat ein potenziell löschendes `remove()`.

4.7.1 Nichtänderbare Datenstrukturen, immutable oder nur Lesen?

Wir können das Problem mit unterschiedlichen Lösungen angehen. Doch als Erstes müssen wir klarstellen, was wir genau wollen, denn das resultiert in unterschiedlichen Ansätzen, die verhindern, dass der Client an die interne Datenstruktur kommt.

- **Kopie der Datenstruktur**: Eine Kopie mit dem Copy-Konstruktor herzustellen, ist eine einfache Lösung; wir schreiben `getItems() { return new ArrayList<String>(items); }`. Das Problem bei der Lösung ist jedoch, dass sie leicht zu Performance-Problemen führt, wenn die Datenmenge groß ist. Aber nach der Kopie sind die interne Datenstruktur und die herausgegebene Sammlung komplett unabhängig, und Änderungen machen keine Probleme. Allerdings muss der Client natürlich auch wissen, dass er auf einer Kopie arbeitet und Änderungen eben nicht durchgeschrieben werden. Dafür gibt es die API-Dokumentation.
- **Nur-Lese-Sichten**: Wenn es möglich ist, Modifikationsoperationen zu verbieten und nur ungefährliche Leseoperationen durchzulassen bzw. eine Nur-Lese-Sicht anzubieten, könnte mehr oder weniger direkt die Datenstruktur nach außen gereicht werden. Java bietet dafür eine gewisse Unterstützung – gleich mehr dazu. Allerdings muss beachtet werden, dass auch diese Sicht nicht unproblematisch ist, denn Änderungen aus dem Inneren einer Komponente spiegeln sich auch in der Sicht wider. »Unmodifiable« bedeutet also nur eine »nicht veränderbare« Datenstruktur für den Client. Wenn zum Beispiel der Client mit der Sicht auf die Datenstruktur arbeitet und sich die Länge holt, dann die Komponente intern Daten löscht und so die Länge ändert, hat der Client eine alte Länge, die schnell zur Ausnahme führen kann. Wenn sich also die interne Datenstruktur ändert, bekommt der Client das immer mit, und es kann Probleme mit der Nebenläufigkeit geben.
- **Immutable Datenstruktur**: In Java sind alle typischen Datenstrukturen mutable, das heißt, Elemente können neu gesetzt oder auch in die Datenstruktur eingefügt und gelöscht werden. Immutable Datenstrukturen gibt es erst in Java 9 und sie könnten problemlos nach außen gegeben werden. Die angenehme Begleiterscheinung ist, dass die Kopie und das Original nicht auseinanderlaufen können und Nebenläufigkeit kein Problem darstellt. Jedoch könnte selbst die Komponente keine Veränderungen mehr vornehmen, da das Objekt sozusagen abgeschlossen ist; nur komplett neue Objekte als Kopie mit veränderten Elementen lassen sich dann wieder aufbauen.
- **Stream-Rückgabe**: Eine gute Rückgabe für Sammlungen ist ein `Stream`, denn über ihn lassen sich die Daten nicht verändern. Der Empfänger kann den Strom ablaufen oder – wenn gewünscht – die Daten einsammeln; Abschnitt 4.11 gibt Details.

Im API-Design immutable Listen in der Rückgabe bevorzugen

Nun lässt das API-Design mehrere Varianten bei Rückgaben von Sammlungen zu:

1. Der Empfänger bekommt eine immutable Sammlung, die er also nicht ändern kann.
2. Der Empfänger bekommt eine Kopie der Daten und kann die empfangene Sammlung nach Herzenslust ändern.

3. Der Empfänger bekommt direkten Zugriff auf interne Zustände und kann diese somit modifizieren.

Alle Varianten haben ihre Vor- und Nachteile, aber üblicherweise wählen Entwickler die erste Variante. Der Grund ist, dass, wie im dritten Fall, Aufrufern kein Einblick in die Interna gegeben werden soll und dass, wie im zweiten Fall, der Aufrufer vielleicht gar keine Änderung vornehmen möchte, sodass die Kopie überflüssig ist – wenn ein Aufrufer eine veränderbare Kopie für sich möchte, erzeugt er einfach eine, etwa mit `new ArrayList(resultList)`.

4.7.2 Null Object Pattern und leere Sammlungen/Iteratoren zurückgeben

In Java gilt es als guter Stil, auf `null` in der Rückgabe wenn möglich zu verzichten. Das Problem ist, dass der Aufrufer dann eine Fallunterscheidung auf `null` bzw. ungleich `null` vornehmen muss, ob die Operation durchführbar war. Insbesondere bei Methoden, die Datenstrukturen liefern, kann leicht auf die `null`-Rückgabe verzichtet werden, denn sie geben einfach eine leere Sammlung zurück. Das nennt sich *Null Object Pattern*, denn statt `null` wird ein Objekt ohne Inhalt, eben ein Null-Objekt, zurückgegeben.

Ein Beispiel soll dieses Vorgehen zeigen. Eine eigene statische Methode `words(String)` soll eine Zeichenkette nach Wörtern zerlegen und diese Wörter in einer `List` zurückgeben:

```
public static List<String> words( String sentence ) {
  if ( sentence == null || sentence.trim().isEmpty() )
    return new ArrayList<>( 0 );

  return Arrays.asList( sentence.split( "(\\s|\\p{Punct})+" ) );
}
```

Ist ein übergebenes Argument `null` oder nur Weißraum im String, so soll eine leere Liste zurückgegeben werden. Andernfalls zerlegen wir die Zeichenkette mit `split(…)`, wobei als Trennausdruck der Einfachheit halber entweder ein Zeichensetzungszeichen alleine oder ein Zeichensetzungszeichen, gefolgt von Leerraum, möglich ist.

Der Vorteil, dass das Null-Objekt, also die leere Liste, eine Fallunterscheidung auf `null` unnötig macht, ist praktisch, da zum Beispiel einfach die Methode `words(Sting)` im erweiterten `for` eingesetzt werden kann: Ist der übergebene »String« bei `words(String)` nun `null`, so kümmert das die erweiterte `for`-Schleife nicht, denn über eine leere Liste muss das erweiterte `for` nicht iterieren.

Szenarien, in denen aufgrund von Bedingungen leere Datenstrukturen zurückgegeben werden, gibt es viele. Nun haben alle diese leeren Sammlungen auch eine Sache gemeinsam: Sie sind alle gleich leer und können sozusagen »gemeinsam« verwendet werden.

Collections.emptyXXX()

Java bietet in `Collections` diverse vorgefertigte leere immutable Datenstrukturen. Dabei gibt es zwei Möglichkeiten. Die erste ist der Rückgriff auf drei statische, finale Variablen:

- `Collections.EMPTY_SET` ist ein leeres immutable `Set`.
- `Collections.EMPTY_LIST` ist eine leere immutable `List`.
- `Collections.EMPTY_MAP` ist eine leere immutable `Map`.

Die Variablen werden wir aber nicht nutzen wollen, denn sie sind alle nicht mit einem generischen Typ deklariert, also im Raw-Typ angeboten. Es ist besser, auf Methoden zurückzugreifen, die Type-Inference nutzen:

```
class java.util.Collections
```

- `static <T> List<T> emptyList()`
- `static <T> Set<T> emptySet()`
- `static <K,V> Map<K,V> emptyMap()`
  Liefert eine leere unveränderbare Datenstruktur.
- `static <E> SortedSet<E> emptySortedSet()`
- `static <E> NavigableSet<E> emptyNavigableSet()`
  Liefert eine leere unveränderbare Menge.
- `static final <K,V> SortedMap<K,V> emptySortedMap()`
- `static final <K,V> NavigableMap<K,V> emptyNavigableMap()`
  Liefert einen leeren unveränderbaren Assoziativspeicher.

Unser Beispiel mit der Methode `word(String)` kann daher mit einer passenden `Collections`-Methode optimiert werden. Und da konsequenterweise auch im nicht leeren Fall eine immutable Datenstruktur zurückgegeben werden sollte, sieht die Lösung wie folgt aus:

```
public static List<String> words( String sentence ) {
  if ( sentence == null || sentence.trim().isEmpty() )
    return Collections.emptyList();

  return Arrays.asList( sentence.split( "(\\s|\\p{Punct})+" ) );
}
```

Die Performance ist nun ausgezeichnet, und der Druck auf die automatische Speicherbereinigung ist genommen, denn ist der String leer oder `null`, muss nun keine neue leere `ArrayList` mehr aufgebaut werden.

Des Weiteren kommen drei statische Methoden hinzu, die leere Iteratoren geben, also Iteratoren, die keine Elemente liefern.


```
class java.util.Collections
```

- `static <T> Iterator<T> emptyIterator()`
- `static <T> ListIterator<T> emptyListIterator()`
- `static <T> Enumeration<T> emptyEnumeration()`

Ein `Iterable<E> emptyIterable()` ist nicht nötig, da ja `Set` und `List` die Schnittstelle `Iterable` implementieren und somit `emptySet()` und `emptyList()` sozusagen `emptyIterable()` sind.

Beispiel

Bei `for ( Object o : iterable )` muss die Variable `iterable` vom Typ `Iterable` und zugleich ungleich `null` sein – bei `null` folgt eine `NullPointerException`. Um diese ungeprüfte Ausnahme zu vermeiden, lässt sich `for ( Object o : unnull(iterable) )` nutzen, und `unnull(Iterable)` ist eine eigene Methode, die bei einem `null`-Argument ein leeres `Iterable` liefert:

```
public static <E> Iterable<E> unnull( Iterable<E> iterable ) {
  return iterable != null ? iterable : Collections.<E>emptySet();
}
```

4.7.3 Immutable Datenstrukturen mit einem Element: Singletons

Singletons sind Objekte, die genau ein Exemplar realisieren. Die Klasse `Collections` bietet drei statische Methoden, die ein gegebenes Element als einziges Element einer immutable Menge, Liste oder eines Assoziativspeichers verpacken:

```
class java.util.Collections
```

- `static <T> Set<T> singleton(T o)`
- `static <T> List<T> singletonList(T o)`
- `static <K,V> Map<K,V> singletonMap(K key, V value)`

Auf den ersten Blick erscheinen die Methoden ziemlich unnütz. Sie sind jedoch immer dann nützlich, wenn eine Methode eine Sammlung erwartet – auch wenn diese Sammlungen nur aus einem Element bestehen – und mit einzelnen Elementen nicht viel anfangen können.

Beispiel

Konvertiere die Elemente eines Arrays in eine Collection.

```
public static <R> Collection<R> of( R... elements ) {
  if ( elements.length == 0 )
    return Collections.emptyList();
```

```
  if ( elements.length == 1 )
    return Collections.singletonList( elements[0] );
  return Collections.unmodifiableList( Arrays.asList(elements.clone()) );
}
```

Löschen von Elementen in einer Sammlung

Die `Collection`-Klassen bieten bisher keine Lösung zum Löschen aller Vorkommen eines Elements. Zwar gibt es die `Collection`-Methode `remove(Object)`, doch löscht diese nur das erste Vorkommen. Uns hilft beim Löschen aller Elemente die Methode `removeAll(Collection)`, doch erwartet sie als Argument eine `Collection`; und hier können wir `singleton(…)`nutzen.

Beispiel

Lösche aus der `Collection ids` alle `null`-Einträge:

```
ids.removeAll( Collections.singleton( null ) );
```

4.7.4 Collections.unmodifiableXXX(…)

Diverse `Collections.unmodifiableXXX(…)`-Methoden legen eine Hülle um eine Datenstruktur und lassen nur die Lesemethoden zum Container durch, blockieren aber Modifizierungsbefehle wie `add(…)` durch eine `UnsupportedOperationException`. Diese Herangehensweise ist gut und performant und trägt keine internen Details nach außen.

Listing 4.23 src/main/java/com/tutego/insel/util//PlayerWithUnmodifiableItems.java

```
public class PlayerWithUnmodifiableItems {

  private List<String> items = new ArrayList<>();

  public List<String> getItems() {
    return Collections.unmodifiableList( items );
  }

  public void addItem( String item ) {
    items.add( item );
  }

  public static void main( String[] args ) {
    PlayerWithUnmodifiableItems p = new PlayerWithUnmodifiableItems();
    p.addItem( "Lasso" );
    System.out.println( p.getItems().get( 0 ) );  // Lasso
```



```
    p.getItems().clear();  // ☠ java.lang.UnsupportedOperationException
  }
}
```

Die Rückgaben von `Collections.unmodifiableXXX(…)` sind Wrapper um die Datenstruktur, die die guten Lesemethoden durchlassen und die ungewollten Modifikationen durch eine `UnsupportedOperationException` abblocken. Für den Aufrufer ändert sich die Schnittstelle nicht, es bleibt in beiden Fällen bei `List`. Wichtig zu verstehen ist, dass die Rückgabe der `unmodifiableXXX(…)`-Methoden immer nur ein Schnittstellentyp ist und der konkrete Klassentyp der Datenstruktur nicht mit durchgereicht wird. Das ist in der Praxis selten hinderlich, wobei es wünschenswert wäre, wenn `Queue` und `Deque` noch als Typ unterstützt würden, denn `peek()` zum Beispiel gibt es in keiner der von `unmodifiableXXX(…)`-unterstützten Schnittstellen.

`class java.util.`**`Collections`**

- `static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)`
- `static <T> List<T> unmodifiableList(List<? extends T> list)`
- `static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)`
- `static <T> Set<T> unmodifiableSet(Set<? extends T> s)`
- `static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)`
- `static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)`
- `static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> s)`
- `static <K,V> NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K,? extends V> m)`

Hinweis

Der Aufruf der `unmodifiableXXX(…)`-Methoden liefert immer eine neue Datenstruktur, was etwas Laufzeit kostet. Das Problem durch diese immer neuen Objekte irritiert auch OR-Mapper wie Hibernate, denn es sind immer neue Exemplare. Auch wichtig: Der Wrapper von `unmodifiableCollection(…)` delegiert die `equals(…)` and `hashCode()`-Methode *nicht* an die Methoden der darunterliegenden Sammlung, alle andere unmodifiable-Wrapper haben eine Implementierung wie `return o == this || collection.equals(o)`.[10]

Zudem ist es Auslegungssache, ob die Methodenimplementierung der von `unmodifiableXXX(…)` zurückgegebenen Objekte wirklich bei `clear()` und `addAll(jede leere Collection)`, `removeAll(jede leere Collection)` eine `UnsupportedOperationException` auslösen muss, denn die Operation ist gültig und könnte unterstützt werden. Wenn das Verhalten so bleibt, lässt sich so zumindest herausfinden, ob die Sammlung unmodifiable ist oder nicht, denn darüber gibt es sonst keine Auskunft: Es gibt weder eine besondere Testmethode noch eine Schnittstelle oder Annotation.

10 Das Gleiche gilt übrigens bei `synchronizedCollection(…)` und `checkedCollection(…)`.

API-Design *

Aus objektorientierter Sicht mag es komisch vorkommen, wenn eine Methode da ist, sie aber nicht genutzt werden kann, weil sie eine Ausnahme auslöst. Die ganzen Bemühungen auch mit Generics gehen in die Richtung, mehr Fehler zur Compilezeit zu finden statt irgendwann später zur Laufzeit. Die Collection-API ist um das Konzept der *optionalen Operationen* gebaut. Das heißt, die Java-API sagt ziemlich klar, ob eine Methode optional ist, also von der implementierenden Klasse nicht zwingend angeboten werden muss. So löst `new HashMap<String, String>().values().add("")` eine `UnsupportedOperationException` aus, denn die Operation kann nicht gelingen. Auch die Methode `remove()` vom `Iterator` ist optional und löst eine `UnsupportedOperationException` aus, wenn der Iterator keine Daten löschen kann. Es gibt keine zwei Schnittstellen `ReadOnlyIterator` und `Iterator`. Weiterhin gilt: Alle die von den `unmodifiableXXX(…)`-geblockten Methoden sind in den Schnittstellen als »optional« gekennzeichnet, daher kann der Entwickler ihre Implementierung nicht erwarten.

Natürlich wäre es schöner, wenn nicht die API-Dokumentation sagt, ob eine Methode vorhanden ist, sondern eine Schnittstelle eine Modifikationsmethode schlichtweg nicht anbieten würde. Der Grund wurde schon kurz im Abschnitt »Optionale Methoden und UnsupportedOperation-Exception« in Abschnitt 4.1.5 angesprochen: Es würden sehr viel mehr Schnittstellen werden, und die Java-Architekten haben sich vor 15 Jahren dagegen entschieden.[11] Zudem könnte es je nach Umsetzung mit Nur-Lese-Schnittstellen doch wieder Probleme geben. Das sei am Beispiel von `CharSequence` erklärt. Die Schnittstelle deklariert Nur-Lese-Operationen auf Zeichenfolgen und wird zum Beispiel von `String` und `StringBuilder` implementiert. Bei einer Deklaration wie

```
public class Buffer {
  private StringBuilder buffer = new StringBuilder();
  public CharSequence getBuffer() { return buffer; }
}
```

könnte ein Entwickler `((StringBuffer)new Buffer().getBuffer()).setLength(0)` schreiben, also einfach einen expliziten Typecast durchführen, und damit doch wieder Unsinn anrichten.

Das gleiche Problem würde sich ergeben, wenn veränderbare Datenstrukturen einfach nur die Nur-Lese-Schnittstelle implementieren, aber immer noch den konkreten Typ zur Laufzeit darstellen und dann etwa unsere Methode `getItems()` – unter der Annahme, es gibt eine Schnittstelle `ReadOnlyList` – als `ReadOnlyList getItems() { return items; }` implementiert würde. Zwar ist eine explizite Typumwandlung vom unartigen Entwickler zurück auf den wahren Typ alles andere als guter Stil, jedoch sollte durch solche einfachen Tricks nicht das ganze Sicherheitsgefüge aus den Angeln gehoben werden. Gegen Zugriff auf private Attribute kann ein Sicherheitsmanager helfen, gegen ungewünschte Typumwandlungen nicht. Doch auch hier könnten im Prinzip die `unmodifiableXXX(…)`-Wrapper helfen, denn die Miniklassen würden die Nur-Lese-Schnittstellen implementieren und an die tatsächliche Implementierung delegieren.

11 Mehr zu der Designentscheidung gibt es unter *https://docs.oracle.com/javase/9/docs/api/java/util/doc-files/coll-designfaq.html#a1*.

4.7.5 Statische ofXXX(...)-Methoden zum Aufbau unveränderbarer Set-, List-, Map-Datenstrukturen

In Java 9 sind echte immutable Datenstrukturen dazugekommen, die sich über statische `ofXXX(…)`-Methoden der Schnittstellen `List`, `Set` und `Map` aufbauen lassen. Jede versuchte Änderung an den Datenstrukturen führt zu einer `UnsupportedOperationException`. Damit eignen sie sich hervorragend für konstante Sammlungen, die problemlos herumgereicht können.

Aus Performance-Gründen sind die `of(…)`-Methoden überladen, das ändert aber nichts an ihren Aufrufvarianten. `null`-Elemente sind grundsätzlich verboten und führen zu einer `NullPointerException`.

```
interface java.util.List<E>
extends Collection<E>
```

- `static <E> List<E> of(E... elements)`
  Erzeugt eine neue immutable Liste aus den Elementen. Vor dem praktischen `List.of(…)` wurden Listen in der Regel mit `Array.asList(…)` aufgebaut. Doch die sind nicht immutable und schreiben auf das Array durch.

```
interface java.util.Set<E>
extends Collection<E>
```

- `static <E> Set<E> of(E... elements)`
  Erzeugt eine Menge aus den gegebenen Elementen. Doppelte Einträge sind verboten und führen zu einer `IllegalArgumentException`. Der Versuch, schon vorhandene Elemente in eine »normale« `HashSet` oder `TreeSet` hinzuzufügen, ist aber völlig legitim. Wie die Implementierung der Menge genau ist, ist verborgen.

Beispiel

Zeige an, welche Superhelden in einem String vorkommen:

```
Set<String> heros = Set.of( "Batman", "Spider-Man", "Hellboy" );
new Scanner( "Batman trifft auf Superman" )
       .tokens().filter( heros::contains )
       .forEach( System.out::println );    // Batman
```

Zum Aufbau von Assoziationsspeichern gibt es zwei Varianten. Einmal über die `of(…)`-Methode, die Schlüssel und Wert einfach hintereinander aufnimmt, und einmal mit `ofEntries(…)` über ein Vararg von `Entry`-Objekten. Eine neue statische Methode hilft, diese `Entry`-Examplare einfach aufzubauen:

```
interface java.util.Map<K,V>
```

- `static <K, V> Map<K, V> of()`
- `static <K, V> Map<K, V> of(K k1, V v1)`
- `static <K, V> Map<K, V> of(K k1, V v1 … )`
- `static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)`
- `static <K, V> Map<K, V> ofEntries(Entry<? extends K, ? extends V>... entries)`
- `static <K, V> Entry<K, V> entry(K k, V v)`

Beispiel

Baue eine `Map` mit Java-Versionen und deren Erscheinungsdaten auf und gib sie aus:

```
Map<String, LocalDate> map =
  Map.ofEntries( Map.entry( "JDK 1.0", LocalDate.of( 1996, Month.JANUARY, 23 ) ),
                 Map.entry( "JDK 1.1", LocalDate.of( 1997, Month.FEBRUARY, 19 ) )
 );
map.forEach( (k, v) -> System.out.println( k + "=" + v ) );
```

Best Practice und weise Worte

Die neuen `ofXXX(…)`-Methoden sind eine Bereicherung, aber auch mit Vorsicht einzusetzen – die alten API-Methoden werden dadurch nicht langweilig:

- Da die `of(…)`-Methoden überladen sind, lässt sich prinzipiell auch `Collections.emptyXXX()` durch `of()` und `Collections.singleton(element)` durch `of(element)` ersetzen – allerdings sagen die `Collections`-Methodennamen gut aus, was hier passiert, und sind vielleicht expliziter.
- Auf einem existierenden Array hat `Arrays.asList(…)` zwar den Nachteil, dass die Array-Elemente ausgetauscht werden können, allerdings ist der Speicherbedarf minimal, da der Adapter `asList(…)` keine Kopie anlegt, wohingegen `List.of(…)` zum Aufbau einer neuen internen Datenstruktur führt, die Speicher kostet.
- Falls `null`-Einträge in der Sammlung sein sollen, dürfen keine `ofXXX(…)`-Methoden verwendet werden.

- Beim Refactoring könnten Entwickler geneigt sein, existierenden Code mit der `Collections`-Methode durch die `ofXXX(…)`-Methoden zu ersetzen. Das kann zum Problem bei serialisierten Daten werden, denn das Serialisierungsformat ist ein anderes.
- Bei `Set` und `Map` wird ein künstlicher `SALT` eingesetzt, der die Reihenfolge der Elemente bei jedem JVM-Start immer ändert. Das heißt, der Iterator von `Set.of("a", "b", "c")` kann einmal »a«, »b«, »c« liefern, dann beim nächsten Programmstart »b«, »c«, »a«.

4.8 Mit einem Iterator durch die Daten wandern

Wenn wir mit einer `ArrayList` oder `LinkedList` arbeiten, so haben wir zumindest eine gemeinsame Schnittstelle `List`, über die wir an die Daten kommen. Doch was vereinigt eine Menge (`Set`) und eine Liste, sodass sich die Elemente der Sammlungen mit gleichem Programmcode erfragen lassen? Listen geben als Sequenz den Elementen zwar Positionen, aber in einer Menge hat kein Element eine Position. Hier bieten sich *Iteratoren* bzw. *Enumeratoren* an, die unabhängig von der Datenstruktur alle Elemente auslesen – wir sagen dann, dass sie »über die Datenstruktur iterieren«. Und nicht nur eine Datenstruktur kann Daten liefern; eine Dateioperation könnte genauso gut Datengeber für alle Zeilen sein.

In Java gibt es für Iteratoren zum einen die Schnittstelle `java.util.Iterator` und zum anderen die ältere `java.util.Enumeration`. Die Schnittstelle `Enumeration` ist nicht mehr aktuell, daher konzentrieren wir uns zunächst auf die aktuelle Schnittstelle.

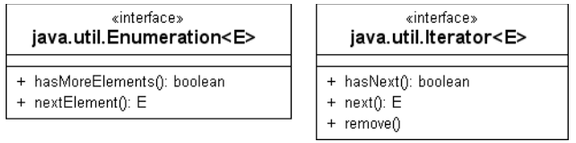

Abbildung 4.5 Iterator und Enumeration

4.8.1 Iterator-Schnittstelle

Die `Iterator`-Schnittstelle ist klein und schreibt zwei Methoden vor; dazu kommen zwei Default-Methoden.

```
interface java.util.Iterator<E>
```

- `boolean hasNext()`
  Liefert `true`, falls die Iteration weitere Elemente bietet.
- `E next()`
  Liefert das nächste Element in der Aufzählung oder `NoSuchElementException`, wenn keine weiteren Elemente mehr vorhanden sind.

Beispiel

Da jede `Collection` eine Methode `iterator()` besitzt, lassen sich alle Elemente einer Sammlung wie folgt auf dem Bildschirm ausgeben:

```
Collection<String> set = new TreeSet<>();
Collections.addAll( set, "Horst", "Schlämmer", "Hape" , "Kerkeling" );
for ( Iterator<String> iter = set.iterator(); iter.hasNext(); )
  System.out.println( iter.next() );
```

Das erweiterte `for` macht das Ablaufen aber noch einfacher, und der gleiche Iterator steckt dahinter.

4.8.2 Der Iterator kann (eventuell auch) löschen

Die Schnittstelle `Iterator` bietet prinzipiell die Möglichkeit, das zuletzt aufgezählte Element aus dem zugrunde liegenden Container mit `remove()` zu entfernen. Vor dem Aufruf muss also `next()` das zu löschende Element als Ergebnis geliefert haben. Eine `Enumeration` kann die aufgezählte Datenstruktur grundsätzlich nicht verändern.

```
interface java.util.Iterator<E>
```

- `default void remove()`
  Entfernt das Element, das der Iterator zuletzt bei `next()` geliefert hat. Als Default-Methode im Rumpf mit `throw new UnsupportedOperationException("remove");` implementiert.

In der Dokumentation ist die Methode `remove()` als optional gekennzeichnet. Das heißt, ein konkreter Iterator muss kein `remove()` können – auch eine `UnsupportedOperationException` ist möglich. Das ist etwa dann der Fall, wenn ein Iterator von einer unveränderbaren Datenstruktur kommt.

Hinweis

Warum es die Methode `remove()` im Iterator gibt, ist eine interessante Frage. Die Erklärung dafür: Der Iterator kennt die Stelle, an der sich die Daten befinden (eine Art Cursor). Darum können die Daten dort auch effizient und direkt gelöscht werden. Das erklärt jedoch nicht unbedingt, warum es keine Einfüge-Methode gibt. Ein allgemeiner Grund mag sein, dass bei vielen Container-Typen das Einfügen an einer bestimmten Stelle keinen Sinn ergibt, etwa bei einem sortierten `NavigableSet` oder einer `NavigableMap`. Dort ist die Einfügeposition durch die Sortierung vorgegeben oder belanglos (bzw. bei `HashSet` durch die interne Realisierung bestimmt), also kein Fall für einen Iterator. Dazu wirft das Einfügen weitere Fragen auf: vor oder nach dem zuletzt per `next()` gelieferten Element? Soll das neue Element mit aufgezählt werden oder nicht? Soll es auch dann nicht aufgezählt werden, wenn es in der Sortierung erst später an die Reihe käme? Eine Löschen-Methode ist problemloser und universell anwendbar.

4.8.3 Operationen auf allen Elementen durchführen

Mit der Default-Methode `forEachRemaining(…)` lässt sich Programmcode (genannt Konsument) für jedes (verbleibende) Element des Iterators aufrufen. Statt also von Hand in einer Schleife das Paar `hasNext()`/`next()` zu verwenden, läuft `forEachRemaining(…)` für uns den Iterator komplett ab. Verschiedene Implementierungen des Iterators können dabei frei machen, was sie wollen, etwa die Abarbeitung in einem Extra-Thread setzen.

```
interface java.util.Iterator<E>
```

- `default void forEachRemaining(Consumer<? super E> action)`

4.8.4 Einen Zufallszahlen-Iterator schreiben

Zur Übung wollen wir einen Iterator schreiben, der Zufallszahlen liefert. Der Konstruktor soll dazu die maximale Zufallszahl entgegennehmen (exklusiv), die Methode `next()` liefert anschließend immer die Zufallszahl und `hasNext()` immer `true`. Da auch `remove()` von der Schnittstelle `Iterator` vorgeschrieben ist, müssen wir die Methode implementieren, lösen jedoch eine Ausnahme aus, da es nichts zu löschen gibt.

Listing 4.24 src/main/java/com/tutego/insel/util/RandomIterator.java

```
package com.tutego.insel.util;

import java.util.*;

/**
 * Iterator for pseudorandom numbers.
 */
public class RandomIterator implements Iterator<Integer> {

  private final Random random = new Random();
  private final int bound;

  /**
   * Initializes this iterator with a maximum value (exclusive) for
   * pseudorandom numbers.
   * @param bound  Maximum (exclusive) pseudorandom
   */
  public RandomIterator( int bound ) {
    this.bound = bound;
  }
```

```
/**
 * Always true.
 * @return {@code true}.
 */
@Override
public boolean hasNext() {
  return true;
}

/**
 * Returns a pseudorandom, uniformly distributed {@code Integer} value
 * between 0 (inclusive) and the specified value (exclusive).
 * @return Next pseudorandom.
 */
@Override
public Integer next() {
  return random.nextInt( bound );
}
```

Ein Beispiel:

Listing 4.25 src/main/java/com/tutego/insel/util/RandomIteratorDemo.java, main()

```
Iterator<Integer> random = new RandomIterator( 6 );
int dice1 = random.next();
int dice2 = random.next();
System.out.println( dice1 );
System.out.println( dice2 );
```

Als Übung kann jeder den Iterator so verändern, dass auch ein Startwert definiert werden kann (jetzt ist er 0).

4.8.5 Iteratoren von Sammlungen, das erweiterte for und Iterable

Jede `Collection` wie `ArrayList` oder `HashSet` liefert mit `iterator()` einen `Iterator`.

```
interface java.util.Collection<E>
extends Iterable<E>
```

- `Iterator<E> iterator()`
  Liefert den Iterator der Datenstruktur.

`Collection` erweitert eine Schnittstelle `Iterable`, die diese Methode auch vorschreibt. `Iterable` kam erst in Java 5 in die Bibliothek, und da gab es `iterator()` schon, sonst wäre die Methode in `Collection` nicht noch einmal nötig.

```
interface java.util.Iterable<T>
```

- `Iterator<T> iterator()`
  Liefert den Iterator für eine Sammlung.

Der typisierte Iterator

Von einer typisierten `Collection` liefert `iterator()` ebenfalls einen typisierten Iterator. Das heißt, die Datenstruktur überträgt den generischen Typ auf den Iterator. Nehmen wir eine mit `String` typisierte Sammlung an:

```
Collection<String> c = new LinkedList<>();
```

Ein Aufruf von `c.iterator()` liefert nun `Iterator<String>`, und beim Durchlaufen über den Iterator kann die explizite Typumwandlung beim `next()` entfallen:

```
for ( Iterator<String> i = c.iterator(); i.hasNext(); ) {
  String s = i.next();
  …
}
```

Iterator und erweitertes for

Stößt der Compiler auf ein erweitertes `for` und erkennt er rechts vom Doppelpunkt den Typ `Iterable`, so erzeugt er Bytecode für eine Schleife, die den `Iterator` und seine bekannten Methoden `hasNext()` und `next()` nutzt. Nehmen wir eine statische Methode `totalStringLength(List)` an, die ermitteln soll, wie viele Zeichen alle Strings zusammen besitzen. Aus

```
static int totalStringLength( List<String> strings ) {
  int result = 0;

  for ( String s : strings )
    result += s.length();

  return result;
}
```

erzeugt der Compiler selbstständig:

```
static int totalStringLength( List<String> strings ) {
  int result = 0;
```

```
  for ( Iterator<String> iter = strings.iterator(); iter.hasNext(); )
    result += iter.next().length();

  return result;
}
```

Da die erweiterte Schleife das Ablaufen einer Datenstruktur vereinfacht, wird ein explizit ausprogrammierter Iterator selten benötigt. Doch der Iterator kann ein Element über die `remove()`-Methode des Iterators löschen, was über das erweiterte `for` nicht möglich ist.

Ein eigener Iterator und Iterable-Implementierung

Die Konzepte `Iterator` und `Iterable` müssen sauber getrennt werden: Ein `Iterator` ist das Objekt, das durch eine Datensammlung läuft, während ein `Iterable` das Objekt ist, das einen anderen `Iterator` liefert. Eine Klasse kann beide Schnittstellen implementieren, doch oft kommt das nicht vor. Für unser nächstes Beispiel ergibt das aber Sinn – es nutzt einem Iterator, der durch eine Sammlung läuft und immer dann, wenn er an das Ende kommt, wieder von vorne beginnt. Zunächst der Einsatz:

Listing 4.26 src/main/java/com/tutego/insel/util/RotatingIteratorDemo.java, main()

```
int i = 0;
for ( String s : new RotatingIterator<>( "Bohnen", "Eintopf" ) ) {
  System.out.println( "Toll, heute gibt es " + s );
  if ( i++ == 7 )
    break;
}
```

Zur Implementierung:

Listing 4.27 src/main/java/com/tutego/insel/util/RotatingIterator.java

```
package com.tutego.insel.util;

import java.util.Arrays;
import java.util.Collection;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An {@link Iterator} that goes over a given {@link Collection}
 * but rolls back to the start when it reaches the end.
 * If the underling {@link Collection} contains no elements the method
```



```
 * {@link #hasNext()} will return {@code false}. Removing elements is
 * supported if the Iterator of the underlying Collection
 * supports {@link Iterator#remove()}. Iterating over the underling
 * Collection and modifying it at the same time will probably result in a
 * {@link ConcurrentModificationException}.
 */
public class RotatingIterator<E> implements Iterator<E>, Iterable<E> {

  private final Collection<? extends E> collection;
  private Iterator<? extends E>         iterator;

  public RotatingIterator( Collection<? extends E> collection ) {
    this.collection = Objects.requireNonNull( collection,
                                              "Collection darf nicht null sein" );
    iterator = collection.iterator();
  }

  public RotatingIterator( E... elements ) {
    this( Arrays.asList( elements ) );
  }

  @Override
  public boolean hasNext() {
    return ! collection.isEmpty();
  }

  @Override
  public E next() {
    if ( ! hasNext() )
      throw new NoSuchElementException( "Keine Elemente in der Collection" );

    if ( ! iterator.hasNext() )
      iterator = collection.iterator();

    return iterator.next();
  }

  @Override
  public void remove() {
    iterator.remove();
  }

  @Override
```

```
  public Iterator<E> iterator() {
    return this;
  }
}
```

4.8.6 Fail-Fast-Iterator und die ConcurrentModificationException

Beackern zwei Programmstellen gleichzeitig eine Datenstruktur, so kann es zu Schwierigkeiten kommen, wenn sich die Datenstruktur strukturell verändert, also neue Elemente hinzukommen oder wegfallen. Probleme treten leicht auf, wenn ein Verweis auf die Datenstruktur im Programm an verschiedenen Stellen weitergegeben wird. Führt nun eine Stelle strukturelle Änderungen mit Methoden wie `remove(…)` oder `add(…)` durch und verändert sich damit die Größe der Liste, dann kann das zu Zugriffsfehlern führen, wenn die andere Stelle von der Änderung nichts mitbekommt.

Nehmen wir an, zwei Programmteile greifen auf eine gemeinsame Liste zurück. Ein Programmteil merkt sich eine Suchstelle und will anschließend auf das gefundene Element zurückgreifen. Zwischen diesen beiden Operationen verändert jedoch ein anderer Programmteil die Liste, und die Position des Elements verschiebt sich:

Listing 4.28 src/main/java/com/tutego/insel/util/ConcurrentModification.java, main()

```
List<String> list = new ArrayList<>( Arrays.asList( "Trullo", "Zippus" ) );
int posOfZippus = list.indexOf( "Zippus" );
System.out.println( list.get( posOfZippus ) );  // Zippus
list.add( 0, "Apulien" );
System.out.println( list.get( posOfZippus ) );  // Trullo
```

Nach dem Einfügen von `"Apulien"` ist `posOfZippus` also veraltet. Es wäre eine Hilfe, wenn `get(…)` die strukturelle Veränderung bemerken würde.

Beim Erfragen über Listen-Iteratoren ist das anders. Die Entwickler der Java-Bibliothek haben einen Entwurf gewählt, bei dem konfliktträchtige Änderungen über die Listen-Iteratoren auffallen und zu einer `ConcurrentModificationException` führen:

Listing 4.29 src/main/java/com/tutego/insel/util/ConcurrentModificationExceptionDemo.java, main()

```
List<String> list = new ArrayList<>( Arrays.asList( "Stunden", "der" ) );
Iterator<String> iterator = list.iterator();  // modCount = 0, expectedModCount = 0
list.get( 0 );                                // Anfragen ändert modCount nicht
list.add( "Entspannung" );                    // modCount = 1, expectedModCount = 0
iterator.next();                              // modCount != expectedModCount => CME
```


Das funktioniert so, dass sich die Liste die Anzahl struktureller Änderungen merkt – bei der Standardimplementierung vom JDK intern in der Variablen `modCount`. Der Iterator registriert also die Änderungen, da sich der `modCount` in der Zwischenzeit durch das `add(…)` verändert hat. Wird der Iterator initialisiert, erfragt er den `modCount` der Liste und speichert den Wert in `expectedModCount`. Jede strukturelle Änderung in der Liste führt zum Inkrement vom `modCount`. Wird später eine Operation über den Iterator durchgeführt, testet er, ob der gemerkte `expectedModCount` mit dem aktuellen `modCount` übereinstimmt. In unserem Fall tut er das nicht, denn `add(…)` ist eine strukturelle Änderung und `modCount` wird 1. Beim nachfolgenden `next()` kommt es zu einer `ConcurrentModificationException`.

Hinweis

Eine strukturelle Änderung ist nur, wenn die Datenstruktur vergrößert oder verkleinert wird, nicht, wenn zwei Stellen gleichzeitig über den Iterator schreiben.

CurrentModificationException vermeiden

Die `CurrentModificationException` lässt sich in der Regel durch Umbau des Programmcodes vermeiden. Rekapitulieren wir, woher die Ausnahme eigentlich kommt: Sie kommt vom parallelen Verändern, also dem Einfügen oder Löschen der Datenstruktur, während ein Lesedurchlauf über den `Iterator` stattfindet. Die Lösung besteht darin, die Veränderungen über den `Iterator` vorzunehmen, denn dieser besitzt Einfüge-/Löschoperationen. Der Standard-Iterator bietet nur `remove()`, doch der `ListIterator` bietet auch eine `set(…)` bzw. `add(…)`-Methode.

Beispiel: Die folgenden Zeilen sollen eine Datenstruktur `result` ablaufen und alle Personen rausschmeißen, die verrückt sind. Lösung eins führt zur `CurrentModificationException`:

```
for ( Person p : result )
  if ( p.isWired() )
    result.remove( p );    // ☠ CurrentModificationException
```

Übertragen wir das Löschen dem `Iterator`, funktioniert es:

```
for ( Iterator<Person> iterator = result.iterator(); iterator.hasNext(); )
  if ( iterator.next().isWired() )
    iterator.remove();
```

4.8.7 Die Schnittstelle Enumeration *

Für Iteratoren deklariert die Java-Bibliothek zwei unterschiedliche Schnittstellen. Das hat historische Gründe: Die Schnittstelle `Enumeration`[12] gibt es seit den ersten Java-Tagen; die Schnittstelle `Iterator` gibt es seit Java 1.2, seit der Collection-API. Der Typ `Iterator` ist jedoch

12 Nicht `Enumerable`!

deutlich weiter verbreitet. Die Namen der Operationen unterscheiden sich in den Schnittstellen ein wenig und sind beim `Iterator` kürzer.

	Hast du mehr?	Gib mir das Nächste!
`Iterator`	`hasNext()`	`next()`
`Enumeration`	`hasMoreElements()`	`nextElement()`

Tabelle 4.4 Methoden von Iterator und Enumeration

```
interface java.util.Enumeration<E>
```

- `boolean hasMoreElements()`
  Testet, ob ein weiteres Element aufgezählt werden kann.
- `E nextElement()`
  Liefert das nächste Element der Enumeration zurück. Diese Methode kann später eine `NoSuchElementException` (eine `RuntimeException`) auslösen, wenn `nextElement()` aufgerufen und das Ergebnis `false` beim Aufruf von `hasMoreElements()` ignoriert wird.
- `default Iterator<E> asIterator()`
  Adapter-Methode, damit sich die `Enumeration` als `Iterator` verwenden lässt. Neu in Java 9. Einige alte APIs liefern Daten nur über eine `Enumeration` aus, sodass die Integration in Iterator-basierte Programme erleichtert wird.

Wie auch `Iterable` erweitert `Enumeration` keine weitere Schnittstelle.

Blick über den Tellerrand

Unter C++ ist die *Standard Template Library* (STL) eine Bibliothek, die Datenstrukturen und Algorithmen implementiert. Ein Vergleich der STL mit der Collection-API ist schwierig, obwohl beide Konzepte wie Listen, Mengen und Iteratoren besitzen. So gibt es in Java nur eine Klasse pro Datenstruktur (auch durch Generics), was die STL durch Template-Klassen (daher *Template* Library) realisiert, die auch primitive Werte (ohne die lästigen Java-Wrapper) speichert. Während Java über den globalen Basistyp `Object` jedes Objekt in jeder Datenstruktur erlaubt und sich auf die Prüfung zur Laufzeit verlässt, ist C++ da viel strenger. Die Java-Datenstrukturen referenzieren die Objekte und speichern nicht ihre Zustände an sich, wie es bei der STL üblich ist – hier steht Referenz-Semantik gegen Wert-Semantik. Fehler werden von der Collection-API über Exceptions angezeigt, während die STL kaum Ausnahmen auslöst und oft undefinierte Zustände hinterlässt; Entwickler müssen eben korrekte Anfragen stellen. STL nutzt überladene Operatoren wie `==` statt des Java-`equals(…)`, `<` für die Ordnung in sortierten Sammlungen, `[]` beim Zugriff und Überschreiben und den Operator `++` für Iteratoren. Iteratoren spielen bei der STL eine sehr große Rolle – es gibt auch einen Random-Access-Iterator, der Indexierung durch `[]` erlaubt. Die STL-Algorithmen sind von den Datenstrukturen abgetrennt – anders als in Java – und bekommen die Elemente über Iteratoren.

Kapitel 19
JavaFX

»Die Reparatur alter Fehler kostet oft mehr als die Anschaffung neuer.«
– Wieslaw Brudzinski (1920–1996)

JavaFX bietet attraktive Fähigkeiten zum Design moderner grafischer Oberflächen. Dieses Kapitel beleuchtet auf der Basis von JavaFX 8 Aspekte wie den Aufbau der Oberflächen über Programmcode oder deklarative XML-Dateien, Charts und weitere Grundlagen.

19.1 Das erste Programm mit JavaFX

Das erste JavaFX-Programm soll die Grundsätze von JavaFX demonstrieren. Im Mittelpunkt des Interesses stehen der Startprozess und die Art des Objektaufbaus:

Listing 19.1 src/main/java/com/tutego/insel/javafx/HelloJavaFX.java

```
package com.tutego.insel.javafx;

import javafx.application.Application;
import javafx.scene.*;
import javafx.scene.effect.Reflection;
import javafx.scene.text.*;
import javafx.stage.Stage;

public class HelloJavaFX extends Application {

  public static void main( String[] args ) {
    launch( args );
  }

  @Override
  public void start( Stage stage ) {
    Text t = new Text( "Die Java-Insel grüßt JavaFX" );
    t.setX( 10.0f );
    t.setY( 50.0f );
```

```
    t.setFont( Font.font( "Calibri", FontWeight.NORMAL, 30 ) );
    t.setEffect( new Reflection() );

    Group root = new Group( t );
    Scene scene = new Scene( root, 400, 100 );
    stage.setScene( scene );
    stage.setTitle( "JavaFX Demo" );
    stage.show();
  }
}
```

Abbildung 19.1 Erstes Pixelquickie mit JavaFX

Eine JavaFX-Anwendung ist eine javafx.application.Application

JavaFX-Anwendungen erweitern die Basisklasse `Application`. Sie vererbt der eigenen Klasse Lebenszyklusmethoden wie `init()`, `destroy()`, `start(…)` oder `stop()`. Die Methoden werden je nach Wunsch überschrieben – wir überschreiben nur `start(Stage)`.

Die eigene statische `main(String[])`-Methode leitet an die statische `launch(String[])`-Methode der `Application`-Klasse weiter und übergibt ihr alle Kommandozeilenoptionen. Da die Klassenmethode `launch(…)` weiß, in welcher Klasse sie aufgerufen wurde, erzeugt sie ein Exemplar dieser Klasse und ruft dann die Lebenszyklusmethoden auf. Soll `launch(…)` von der eigenen Klasse entkoppelt werden, lässt sich eine überladene Methode einsetzen, die über ein `Class`-Objekt eine andere zu startende JavaFX-Klasse bestimmt.

Start der Anwendung

JavaFX ruft im Lebenszyklus der JavaFX-Anwendung nach dem `init()` die Methode `start(Stage primaryStage)` auf. Unsere Klasse überschreibt `start(…)`, um Programmcode abzuarbeiten und die grafische Ausgabe vorzubereiten. JavaFX übergibt der Methode eine *Stage* (zu Deutsch Bühne), was der Aufgabe eines Haupt-Containers zukommt und am ehesten mit dem Startfenster verglichen werden kann. Diesem `Stage`-Objekt kann wie bei einem Fenster ein Titel zugewiesen werden, auch die Ausmaße und Dekorationen, oder das Fenster kann in den Vollbildmodus gesetzt werden. Am wichtigsten ist aber die Zuweisung einer Szene, die auf den eigentlichen Inhalt verweist. Um eine Analogie zu gebrauchen: Die Szene ist die Leinwand und die Stage der Bilderrahmen mit Leinwand, also die gesamte Bühne. Während die primäre Stage von JavaFX kommt, sind wir es, die den Szenegraphen aufbauen, mit Elementen füllen und dann der Stage zuweisen.

Hinweis

Während ein AWT- oder Swing-Programm selbst ein Fensterobjekt aufbaut, ist die Herangehensweise bei JavaFX fundamental anders. JavaFX baut das Fenster auf und übermittelt es einmalig in der `start(Stage)`-Methode. Dieses `Stage` müssen wir uns gut merken, denn alle Oberflächenelemente kommen später auf diesen Container. `Stage` ist bei Swing mit `JFrame` vergleichbar.

Der Szenegraph

Zentraler Dreh- und Angelpunkt bei JavaFX ist eine besondere Datenstruktur, genannt *Szenegraph* (engl. *scene graph*). Dieser nimmt grafische Objekte wie Linien, Flächen, Textfelder, Browser, 3D-Objekte auf und verwaltet sie. Zu jedem Zeitpunkt sind also alle grafischen Objekte einer JavaFX-Anwendung im Speicher präsent, und sie existieren nicht nur zum Zeitpunkt des Zeichnens. (Für sehr große Datenmengen lassen sich andere Lösungen finden, etwa dass die Grafiken gezeichnet und dann in den Szenegraphen gehängt werden.)

JavaFX repräsentiert über die Klasse `Scene` den Szenegraphen, der dann mit `stage.setScene(scene)` auf die Bühne kommt. Jetzt müssen nur noch die Objekte auf die `Scene`.

Auf die Bühne

Ein Szenegraph ist eine hierarchische Datenstruktur, die ein Wurzelelement haben muss. Daher wird bei Aufbau eines `Scene`-Objekts im Konstruktor ein Wurzelobjekt übergeben, das ist entweder ein einzelnes GUI-Element oder ein Container. Üblicherweise weisen wir `Scene` einen Knoten-Container vom Typ `Group` zu. Eine `Group` ist selbst ein Knoten, sodass sich Gruppen auch schachteln lassen – es ist das typische Composite-Pattern.

`Group` ist der Behälter für Knoten, die intern in einer beobachtbaren Datenstruktur gehalten werden. Diese Datenstruktur muss mit `getChildren()` erfragt werden – das Ergebnis ist eine `ObservableList<Node>` –, und daran können wir unsere Objekte hängen – ein direktes `add(…)` hat `Group` nicht. Group kann auch gleich im Vararg-Konstruktor alle Knoten annehmen.

Knoten

Alle Objekte im Szenegraphen sind vom Typ `Node`. Ein Untertyp von `Node` ist `javafx.scene.shape.Shape`, das Elemente wie `Line`, `Text` oder `javafx.scene.control.Control` repräsentiert, wozu Texteingabefelder oder Schaltflächen zählen.

`Text`-Objekte repräsentieren Beschriftungen, die über den Konstruktor oder über `setContent(String)` gesetzt werden können. Als Zeilentrenner wird »\n« unterstützt. Wir nutzen einige Setter im Beispiel:

- `setX(float)`/`setY(float)` setzen die Koordinaten.
- `setFill(Paint)` setzt eine Farbe oder Gradienten – bei Farben sind es `javafx.scene.paint.Color`-Objekte.

- `setFont(Font)` setzt einen neuen Zeichensatz.
- `setEffect(…)` setzt in unserem Fall einen Spiegeleffekt. Fast 20 Effekte bringt JavaFX mit, darunter `GaussianBlur`, `Glow`, `SepiaTone` oder `Shadow`.

Alle Knoten wie unser Textelement müssen dem Szenegraphen hinzugefügt werden, wozu wir sie der `Group` hinzufügen.

JavaFX versus Swing/AWT

Die Java 2D-API brachte gegenüber dem AWT die Neuerung, dass es erstmalig grafische Objekte vom Typ `java.awt.Shape` gab. Sie können in einer Datenstruktur gesammelt werden und repräsentieren die Eigenschaften wie Koordinaten in einem Objekt. Linien mussten also nicht mehr direkt über `drawLine()` gezeichnet werden, sondern konnten als `java.awt.geom.Line2D.Double`-Objekte aufgebaut und verwaltet werden. JavaFX geht noch einen Schritt weiter und zeigt uns gar keinen Grafikkontext mehr, sondern gibt uns nur die Möglichkeit, Objekte zu sammeln und in den Graphen zu hängen. Dabei nutzt JavaFX auch `Shape`-Typen, aber nicht eine Schnittstelle `java.awt.Shape` mit Implementierungen im Paket `java.awt.geom`, sondern ganz eigene Typen – JavaFX hat mit den `java.awt`-Typen überhaupt nichts gemeinsam. So nutzt JavaFX auch eigene Typen für Farben und Fonts, etwa `javafx.scene.paint.Color` statt `java.awt.Color`.

19.2 Zentrale Typen in JavaFX

JavaFX hat ein vielschichtiges Klassenkonzept, und diverse Muster finden sich dort wieder.

19.2.1 Szenegraph-Knoten und Container-Typen

Die wichtigsten Klassen aus dem Paket `javafx.scene` sind folgende:

- Eine zentrale Oberklasse in JavaFX ist `Node`. Jeder Knoten und jedes Element im Szenegraphen ist ein Untertyp der abstrakten Klasse `Node`. Jeder Knoten darf nur einmal an einer Stelle im Szenegraphen auftauchen.
- Eine Unterklasse von `Node` ist `Parent`. `Parent` ist ebenfalls eine abstrakte Klasse und Basistyp aller Objekte, die Kinder haben, also Container sind. Ein `Text` zum Beispiel ist nicht vom Typ `Parent`, da ein `Text` nur einen Text darstellt, aber keine eigenen Kinder hat. Von `Parent` gibt es vier Unterklassen: `Control`, `Group`, `Region` und `WebView`.
- `Control` ist ein Basistyp für Komponenten, also etwa für Schaltflächen oder Beschriftungen. Jede `Control`-Komponente lässt sich über CSS optisch stylen. Die meisten `Control`-Objekte verarbeiten Ereignisse und nehmen am Fokuswechsel teil.

- Eine `Group` ist ein einfacher Container, der mehrere Elemente zu einem Knoten zusammenfasst. Interessant ist eine `Group` deswegen, weil zum Beispiel Transformationen oder Effekte auf eine Gruppe angewendet werden können und dann diese Transformationen auf alle Elemente der Gruppe delegiert werden. Gruppen lassen sich beliebig schachteln. Ein `Group`-Objekt hat eine Methode `getChildren()`, die eine `ObservableList<Node>` liefert, der dann Knoten zugeführt werden können. Die Gruppe selbst gibt den Kindern aber keine Positionen, was sie von einem typischen Swing-Container wie `JPanel` unterscheidet.
- Eine `Region` ist ein besonderer `Parent`, also ein Container, dem nicht nur Elemente hinzugefügt werden können, sondern der selbst als Container über CSS optisch angepasst werden kann. Von `Region` gibt es eine Unterklasse `Pane`, die die Kinder über öffentliche Methoden nach außen gibt. `FlowPane` ist eine Unterklasse von `Pane`, die Kinder der Reihe nach anordnet und wenn nötig umbricht. Da eine `FlowPane` eine `Pane` ist und eine `Pane` eine `Region`, kann etwa eine Reihe von Komponenten über CSS zum Beispiel extern mit einer Hintergrundfarbe konfiguriert werden. Eine `Group` und eine `Region` haben in der Vererbungshierarchie nur `Parent` über sich, stehen also nebeneinander. Um den Unterschied noch einmal zu betonen: Eine `Group` kümmert sich nicht um die Positionen der Kinder, und sie kann nicht über CSS angepasst werden, eine `Region` schon.

19.2.2 Datenstrukturen

Das Paket `javafx.collections` deklariert Datenstrukturen, auf die JavaFX zurückgreift, etwa bei einer `Group`. Diese neuen Typen wurden in Kapitel 4, »Datenstrukturen und Algorithmen«, schon ausführlich vorgestellt, deshalb hier nur noch einmal eine kompakte Zusammenfassung:

Beispiel

Füge einer neuen Gruppe zwei Textobjekte, `text1` und `text2`, hinzu:

```
Group group = new Group();
group.getChildren().addAll( text1, text2 );
```

Neue Container zu deklarieren hat einen guten Grund, denn die JavaFX-Datenstrukturen lösen bei Veränderungen Ereignisse aus. Die Sammlungen aus `java.util` können zum Beispiel nicht melden, wenn ein Element verschwindet oder hinzugekommen ist.

Für Listen und Assoziativspeicher deklariert JavaFX zwei neue Schnittstellen:

- `ObservableList<E>` (erweitert `java.util.List<E>`)
- `ObservableMap<K,V>` (erweitert `java.util.Map<K,V>`)

Es gibt keine benutzersichtbaren implementierenden Klassen, sondern eine Utility-Klasse `FXCollections` mit statischen Methoden, die Exemplare dieses Typs liefern.

Beispiel

Lege eine `ObservableList` an, und beobachte die Änderungen:

```
ObservableList<String> list = FXCollections.observableArrayList();
list.addListener( (ListChangeListener<String>) e -> {
  …
} );
list.add( "1" );
list.add( "2" );
list.remove( 1 );
list.clear();
```

Ein `ListChangeListener.Change`-Objekt liefert Informationen, etwa ob die Liste wächst oder schrumpft oder an welcher Stelle es Veränderungen gab.

19.3 JavaFX-Komponenten und Layout-Container-Klassen

JavaFX kommt wie Swing mit einem Satz von Standardkomponenten im Paket `javafx.scene.control` und hat `Control` als Basisklasse.

19.3.1 Überblick über die Komponenten

Die Klassennamen sind selbsterklärend:

- `Label`
- `Hyperlink`
- `Button`
- `RadioButton`
- `ToggleButton`
- `CheckBox`
- `ChoiceBox`
- `TextField`
- `PasswordField`
- `ListView`
- `TableView`
- `TitledPane`
- `ScrollPane`
- `ProgressBar`
- `ProgressIndicator`
- `Pane` mit Unterklassen für unterschiedliche Layouts

Styling

Cascading Style Sheets (CSS) können das Aussehen anpassen. Der Vorteil ist, dass dies deklarativ ohne Programmieraufwand möglich ist. Die CSS-Deklaration lässt sich in eine Datei auslagern und für eine ganze JavaFX-Applikation anwenden oder auch lokal mit `setStyle()` setzen. Neben dem Styling lässt sich eine komplett neue Skin definieren, etwa für Oberflächen mit besonders starken Kontrasten.

Bei der Konfiguration findet sich keine Klasse vom AWT oder von Swing wieder. Im eingehenden Beispiel ließ sich das schon an der JavaFX-eigenen Klasse `Font` ablesen. Das geht weiter bei den Icons, die als `ImageView`-Objekte zum Beispiel an Schaltflächen gesetzt werden. Für Farben gibt es die eigene `Color`-Klasse mit reichlich vorgefertigten Farbkonstanten und einer Methode `Color.web()`, die die Farbangaben hexadezimal webüblich annimmt. JavaFX ist als vollständig neue Technologie zu verstehen, die alte Zöpfe abschneidet.

19.3.2 Listener/Handler zur Ereignisbeobachtung

Die Ereignisbehandlung folgt dem bekannten Listener-Prinzip. Es ist unnötig, zu betonen, dass neue JavaFX-Typen verwendet werden. Einen anderen Unterschied gibt es, der an GWT erinnert: Jede Komponente ist ein Knoten, und jedem Knoten lässt sich im Prinzip jeder Listener zuordnen, wobei der Listener bei JavaFX *Handler* genannt wird.

Allgemeiner Event-Handler

Die Klasse für alle Knoten `Node` deklariert:

- `addEventHandler(EventType<T>, EventHandler<? super T>)` zum Hinzufügen der Handler
- `removeEventHandler(EventType<T>, EventHandler<? super T>)` zum Abmelden der Handler

`addEventHandler()` ist also eine allgemeine Möglichkeit, etwa einer Schaltfläche einen Horcher zuzuordnen. Für einen `Button b` könnte das so aussehen:

```
b.addEventHandler( ActionEvent.ACTION, e -> {
…
} );
```

Spezielle Handler

Neben der Möglichkeit, über die `addEventHandler(…)`-Methode einen Handler hinzuzufügen, bieten die Komponentenklassen auch eigene Hinzufügemethoden, etwa der `Button` die Methode `setOnAction(EventHandler<ActionEvent>)`; der `EventHandler<T>` ist eine funktionale generische Schnittstelle mit einer Methode `handle(T e)`, in unserem Fall muss also `handle(ActionEvent e)` implementiert werden. Das geht am einfachsten über einen Lambda-Ausdruck.

Das nächste Beispiel deaktiviert beim Klick auf die Schaltfläche diese:

Listing 19.2 src/main/java/com/tutego/insel/javafx/ButtonSetOnActionDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  Button b = new Button( "OK" );
  b.setFont( Font.font( "Calibri", 30 ) );
  Image image = new Image( getClass().getResourceAsStream( "/images/ok.png" ) );
```

```
    b.setGraphic( new ImageView( image ) );
    b.setOnAction( e -> b.setDisable( true ) );

    stage.setScene( new Scene( b ) );
    stage.show();
  }
```

19.3.3 Panels mit speziellen Layouts

Unser Beispiel für die zur Positionierung absolute Angaben mit `setX(…)` und `setY(…)`. Das ist für Komponenten eher unüblich, und eine Anordnung nach speziellen Layouts ist zu bevorzugen. JavaFX bietet spezielle Container-Klassen, die jeweils ein individuelles Layout realisieren.

> **Vergleich zwischen Swing und JavaFX**
>
> Das ist anders als bei Swing, bei dem das Strategiemuster zum Zuge kommt: Ein Container ist mit einem Layoutmanager assoziiert, der die Kinder anordnet.

JavaFX-Container

Es stehen in JavaFX folgende Container-Klassen zur Verfügung (in Klammern stehen die vergleichbaren Layoutmanager in Swing, die etwa einem `JPanel` zugewiesen werden):

- `BorderPane` (vergleichbar mit `BorderLayout`)
- `FlowPane` (vergleichbar mit `FlowLayout`)
- `HBox`, `VBox` (vergleichbar dem Container über `Box.createHorizontalBox()` bzw. `Box.createVerticalBox()`)
- `StackPane` (vergleichbar mit `CardLayout`)
- `GridPane` (wie ein einfacheres `GridBagLayout`)
- `TilePane` (vergleichbar mit `GridLayout`)
- `AnchorPane` (vergleichbar mit `BorderLayout` und Komponenten in allen Ecken)

Beispiel

Im nächsten Beispiel soll ein kleines Formular Vor- und Nachname erfassen. Die `GridPane` ist für Anordnungen in einem Raster bestens geeignet.

Listing 19.3 src/main/java/com/tutego/insel/javafx/GridPaneDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  GridPane grid = new GridPane();
  grid.setHgap( 10 );
  grid.setVgap( 10 );
  grid.setPadding( new Insets( 10, 10, 10, 10 ) );

  Label header = new Label( "Kontaktdaten" );
  header.setFont( new Font( 20 ) );
  GridPane.setHalignment( header, HPos.CENTER );

  grid.add( header,
            0 /*x*/, 0 /*y*/,
            2 /*colSpan*/, 1 /*rowSpan*/ );
  grid.add( new Label( "Vorname" ), 0, 1 );
  grid.add( new TextField(), 1, 1 );
  grid.add( new Label( "Nachname" ), 0, 2 );
  grid.add( new TextField(), 1, 2 );

  stage.setScene( new Scene( grid ) );
  stage.show();
}
```

Abbildung 19.2 Screenshot der Anwendung GridPaneDemo

Die `GridPane` wächst automatisch und muss nicht mit einer festen Anzahl von Spalten/Zeilen erzeugt werden. Die `add(…)`-Methode setzt später eine Komponente an die gewünschte Stelle, wobei es eine überladene `add(…)`-Methode gibt, die die Komponente über mehrere Zellen verteilen kann. Wie die Komponente dann in ihrem Bereich ausgerichtet wird, bestimmen die statischen Methoden `setHalignment(…)` und `setValignment(…)`, denen die auszurichtende Komponente und die Orientierung mitgegeben werden.

19.4 Webbrowser

Eine ganz besondere Komponente ist `WebView`, die mit einer referenzierten `WebEngine` HTML rendern kann. JavaFX basiert auf der nativen quelloffenen Bibliothek *WebKit* (*http://www.webkit.org/*), die von Apple, Google, Nokia, Adobe und weiteren IT-Schwergewichten entwickelt wird bzw. wurde. Mit dieser schönen Komponente lassen sich einfach existierende Webanwendungen einbetten, etwa Karten, Bezahlsysteme, E-Mail-Clients oder Kommunikationssoftware.

Um HTML darzustellen, steht am Anfang der Aufbau eines `WebView`-Objekts. Von ihm wird die `WebEngine` erfragt, die weitere Methoden bietet, wie `load(…)` oder `loadContent(…)`; der Ladeprozess lässt sich dann über `stop()` abbrechen oder über `reload()` neu anstoßen.

Verbinden wir das zu einem kleinen Beispiel:

Listing 19.4 src/main/java/com/tutego/insel/javafx/WebViewDemo.java

```
package com.tutego.insel.javafx;

import javafx.application.Application;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class WebViewDemo extends Application
{
  @Override
  public void start( Stage stage )
  {
    WebView htmlViewer1 = new WebView();
    htmlViewer1.getEngine().loadContent( "<html>Hallo <b>WebView</b>.</html>" );

    WebView htmlViewer2 = new WebView();
    htmlViewer2.getEngine().load( "https://wiki.openjdk.java.net/display/OpenJFX/Main" );

    SplitPane splitPane = new SplitPane();
    ScrollPane scrollPane = new ScrollPane( htmlViewer2 );
    scrollPane.setFitToWidth( true );
    splitPane.getItems().addAll( htmlViewer1, scrollPane );
    splitPane.setDividerPositions( 0.2f );
    stage.setScene( new Scene( splitPane ) );
    stage.show();
  }
  public static void main( String[] args )
  {
    launch( args );
  }
}
```


Abbildung 19.3 Screenshot der Anwendung WebViewDemo

Die `SplitPane` stellt zwei HTML-Dokumente links und rechts dar. Im rechten Bereich steht ein über eine URL geladenes Dokument, links ein statisch generiertes Dokument. Mit `load()` auf dem `WebEngine`-Objekt lässt sich jederzeit ein neues HTML-Dokument laden. Interessant ist auch die Methode `executeScript()`, die die `WebEngine` anweist, JavaScript auszuführen. Damit lässt sich aus Java heraus auf das laufende JavaScript-Programm Einfluss nehmen und Zustände im DOM ändern. Aktivierte Hyperlinks verfolgt der Browser automatisch.

19.5 Geometrische Objekte

Die JavaFX-Bibliothek repräsentiert geometrische Formen wie Linien, Polygone oder Kurven durch Objekte. Diese zweidimensionalen Formenobjekte sind abgeleitet von einer abstrakten Basisklasse `javafx.scene.shape.Shape`, die dreidimensionalen von `javafx.scene.shape.Shape3D`. Konkrete Formen realisieren Unterklassen, von `Shape` sind das `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, `Polyline`, `Path`, `CubicCurve`, `QuadCurve`, `SVGPath` und `Text`. Bis auf `Text` liegen alle `Shape`-Unterklassen im Paket `javafx.scene.shape`.

Erstes Beispiel

Die Oberklasse `Shape` erweitert `javafx.scene.Node`, und somit sind die konkreten Formen allesamt gültige Objekte im Szenegraphen. Fünf konkrete `Shape`-Objekte kommen in unserem Beispiel in eine `Group` und dann auf den Bildschirm:

Listing 19.5 src/main/java/com/tutego/insel/javafx/ShapeDemo.java, Ausschnitt

```
Shape text = new Text( 10, 50, "Die Java-Insel grüßt JavaFX" );
Shape arc = new Arc( 100, 100, 20, 40, 0, 180 );
Shape circle = new Ellipse( 100, 200, 20, 20 );
OfInt rnd = new Random().ints( 10, 780 ).iterator();
Shape poly1 = new Polyline( rnd.nextInt(), rnd.nextInt(), rnd.nextInt(),
                            rnd.nextInt(), rnd.nextInt(), rnd.nextInt() );
Shape poly2 = new Polygon(  rnd.nextInt(), rnd.nextInt(), rnd.nextInt(),
                            rnd.nextInt(), rnd.nextInt(), rnd.nextInt() );
SVGPath svg = new SVGPath();
svg.setContent( "M300 50 L175 200 L425 200 Z" );

Group group = new Group( text, arc, circle, poly1, poly2, svg );
stage.setScene( new Scene( group, 800, 800 ) );
stage.show();
```

Hinweis

Um Formen zu zeichnen, gibt es in JavaFX zwei Möglichkeiten: Die eine ist – wie im Beispiel gesehen –, `Shape`-Objekte zu nutzen und diese in den Szenegraphen zu hängen. Die andere ist, eine Zeichenfläche (`Canvas`) zu erzeugen, sich darüber einen `GraphicsContext` zu besorgen und Methoden wie `fillOval(…)` oder `drawImage(…)` einzusetzen, um damit auf einer Zeichenfläche zu zeichnen, die dann selbst als Objekt in den Szenegraphen gehängt werden kann. Was es in JavaFX nicht gibt, ist eine Zeichenmethode wie `draw(Shape)`.

19.5.1 Linien und Rechtecke

Bei Linien müssen wir uns von der Vorstellung trennen, die uns die analytische Geometrie nahelegt. Laut Euklid ist eine Linie als kürzeste Verbindung zwischen zwei Punkten definiert. Da Linien eindimensional sind, besitzen sie eine Länge aus unendlich vielen Punkten, doch keine wirkliche Breite.

Auf dem Bildschirm besteht eine Linie nur aus endlich vielen Punkten, und wenn eine Linie gezeichnet wird, werden Pixel gesetzt, die nahe an der wirklichen Linie sind. Die Punkte müssen passend in ein Raster gesetzt werden, und so kommt es vor, dass die Linie in Stücke zerbrochen wird. Dieses Problem gibt es bei allen grafischen Operationen, da von Fließkommawerten eine Abbildung auf Ganzzahlen, in unserem Fall absolute Koordinaten des Bildschirms, durchgeführt werden muss. Eine bessere Darstellung der Linien und Kurven ist durch *Anti-aliasing* zu erreichen. Dies ist eine Art Weichzeichnung mit nicht nur einer Farbe, sondern mit Abstufungen, sodass die Qualität auf dem Bildschirm wesentlich besser ist. Auch bei Zeichensätzen ist dadurch eine merkliche Verbesserung der Lesbarkeit auf dem Bildschirm zu erzielen.

Linien

Im ersten Beispiel haben wir schon `javafx.scene.shape.Line` verwendet, das Objekt verwaltet Start- und Endkoordinate:

Property	Property-Typ	Funktion
`startX`, `startY`	`DoubleProperty`	Start-Koordinate x/y
`endX`, `endY`	`DoubleProperty`	End-Koordinate x/y

Tabelle 19.1 Properties von Line

Neben den Settern/Gettern, einem Standard-Konstruktor und dem Konstruktor `Line(double startX, double startY, double endX, double endY)` hat die Klasse nichts zu bieten.

Rechtecke

Rechtecke sind im Grunde nichts anderes als vier Linien. Doch Rechtecke kann JavaFX auch füllen, und zudem können sie abgerundet sein, wofür die Klasse die Angabe eines Bogen-Durchmessers für die Ecken erlaubt – alle Ecken können nur die gleiche Rundung haben.

Property	Property-Typ	Funktion
`X`, `y`	`DoubleProperty`	Start-Koordinate x/y
`width`, `height`	`DoubleProperty`	Breite/Höhe des Rechtecks
`arcHeight`	`DoubleProperty`	Durchmesser für Eckenbogen

Tabelle 19.2 Properties von Rectangle

Die Klasse hat die zu erwartenden Setter/Getter und vier Konstruktoren:

```
class javafx.scene.shape.Rectangle
extends Shape
```

- `Rectangle()`
- `Rectangle(double width, double height)`
- `Rectangle(double x, double y, double width, double height)`
- `Rectangle(double width, double height, Paint fill)`

19.5.2 Kreise, Ellipsen, Kreisförmiges

Ein Kreis wird in JavaFX von der Klasse `Circle` repräsentiert, eine Ellipse durch `Ellipse`. Beide verfügen über einen Mittelpunkt. Während der Kreis nur einen Radius hat, hat die Ellipse zwei Radien, die Breite und Höhe bestimmen:

Klasse	Property	Property-Typ	Funktion
`Circle`, `Ellipse`	`centerX`	`DoubleProperty`	Mittelpunkt, x-Achse
`Circle`, `Ellipse`	`centerY`	`DoubleProperty`	Mittelpunkt, y-Achse
`Circle`	`radius`	`DoubleProperty`	Radius des Kreises
`Ellipse`	`radiusX`	`DoubleProperty`	x-Radius der Ellipse
`Ellipse`	`radiusY`	`DoubleProperty`	y-Radius der Ellipse

Tabelle 19.3 Properties von Circle und Ellipse

Beide Klassen haben einen Standard-Konstruktor und dazu folgende weitere parametrisierte Konstruktoren:

```
class javafx.scene.shape.Circle
extends Shape
```

- `Circle(double radius)`
- `Circle(double centerX, double centerY, double radius)`
- `Circle(double centerX, double centerY, double radius, Paint fill)`
- `Circle(double radius, Paint fill)`

```
class javafx.scene.shape.Ellipse
extends Shape
```

- `Ellipse(double radiusX, double radiusY)`
- `Ellipse(double centerX, double centerY, double radiusX, double radiusY)`

Die Ellipse kann also bisher nicht direkt mit Füllfarbe initialisiert werden.

Kreisbogen

Die Klasse `javafx.scene.shape.Arc` repräsentiert einen Kreisbogen. Diese Bögen ähneln Ellipsen, nur dass sie einen Startwinkel (in Grad) und relativ von dort einen Bogenwinkel haben. Und da ein Kreisbogen an sich »offen« ist, gibt es einige Optionen, ob das so bleibt:

- `ArcType.OPEN`: Kreisbogen bildet eine einfache Kreislinie, der verbleibende Teil ist offen.

- `ArcType.CHORD`: Start- und Endpunkt des Bogens werden durch eine Linie verbunden.
- `ArcType.ROUND`: Start- und Endpunkt des Bogens werden mit dem Mittelpunkt des Kreises verbunden, was dann so aussieht wie ein Kuchen (unter Java 2D noch »Pie« genannt).

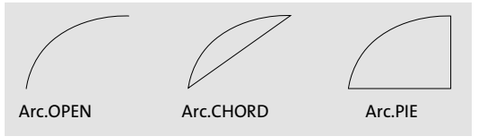

Abbildung 19.4 Die drei Bogentypen

Damit ist jeder Kreisbogen durch folgende Properties bestimmt:

Property	Property-Typ	Funktion
`centerX`, `centerY`	`DoubleProperty`	x-/y-Achse vom Mittelpunkt des Kreisbogens
`radiusX`, `radiusY`	`DoubleProperty`	Horizontaler/vertikaler Radius
`startAngle`	`DoubleProperty`	Startwinkel
`length`	`DoubleProperty`	Winkel des Bogens
`type`	`ObjectProperty<ArcType>`	`ArcType.OPEN`, `ArcType.CHORD` oder `ArcType.ROUND`

Tabelle 19.4 Properties von Arc

Neben den Settern/Gettern für die Properties hat die Klasse `Arc` nichts zu bieten. Zwei Konstruktoren gibt es:

```
class javafx.scene.shape.Arc
extends Shape
```

- `Arc()`
  Baut einen Kreisbogen ohne irgendwelche gesetzten Properties auf.
- `Arc(double centerX, double centerY, double radiusX, double radiusY, double startAngle, double length)`
  Baut einen Kreisbogen mit Zustand auf.

19.5.3 Es werde kurvig – quadratische und kubische Splines

Die Klasse `QuadCurve` beschreibt *quadratische Kurvensegmente*. Dies sind Kurven, die durch zwei Endpunkte und einen dazwischenliegenden Kontrollpunkt definiert sind. Eine weitere Klasse `CubicCurve` beschreibt *kubische Kurvensegmente*, die durch zwei Endpunkte und zwei

Kontrollpunkte definiert sind. Kubische Kurvensegmente werden auch *Bézier-Kurven* genannt.

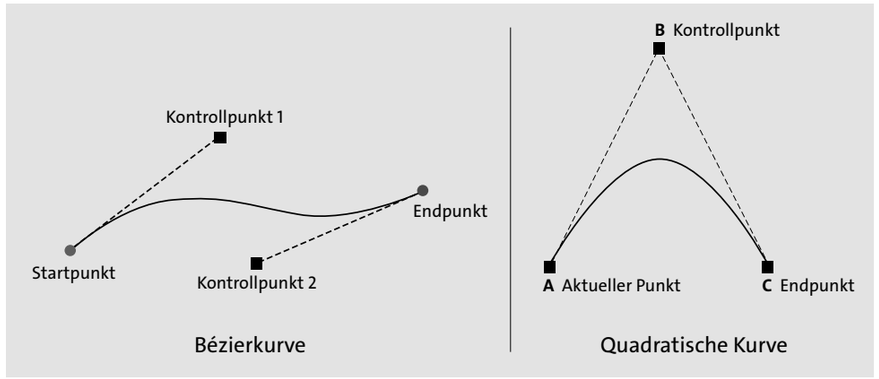

Abbildung 19.5 Kontrollpunkte der Kurven

Listing 19.6 src/main/java/com/tutego/insel/javafx/CubicCurveDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  double startX = 200, startY = 200;
  DoubleProperty controlX1 = new SimpleDoubleProperty( 20 );
  DoubleProperty controlY1 = new SimpleDoubleProperty( 20 );
  DoubleProperty controlX2 = new SimpleDoubleProperty( 400 );
  DoubleProperty controlY2 = new SimpleDoubleProperty( 20 );
  double endX = 300, endY = 200;

  // Linie von [controlX1, controlY1] nach [startX, startY]
  Line line1 = new Line( 0, 0, startX, startY );
  line1.startXProperty().bind( controlX1 );
  line1.startYProperty().bind( controlY1 );
  line1.setStrokeWidth( 2 );

  // Linie von [controlX2, controlY2] nach [endX, endY]
  Line line2 = new Line( 0, 0, endX, endY );
  line2.startXProperty().bind( controlX2 );
  line2.startYProperty().bind( controlY2 );
  line2.setStrokeWidth( 2 );

  // Animierte Kontrollpunkte
  Timeline timeline = new Timeline( new KeyFrame( Duration.millis( 1000 ),
                                    new KeyValue( controlX1, 300 ),
                                    new KeyValue( controlY2, 300 ) ) );
```



```
  timeline.setCycleCount( Animation.INDEFINITE );
  timeline.setAutoReverse( true );
  timeline.play();

  CubicCurve curve = new CubicCurve( startX, startY, 0, 0, 0, 0, endX, endY );
  curve.controlX1Property().bind( controlX1 );
  curve.controlY1Property().bind( controlY1 );
  curve.controlX2Property().bind( controlX2 );
  curve.controlY2Property().bind( controlY2 );
  curve.setFill( null );
  curve.setStroke( Color.BLUEVIOLET );
  curve.setStrokeWidth( 3 );

  stage.setScene( new Scene( new Group( line1, line2, curve ), 450, 300 ) );
  stage.show();
}
```

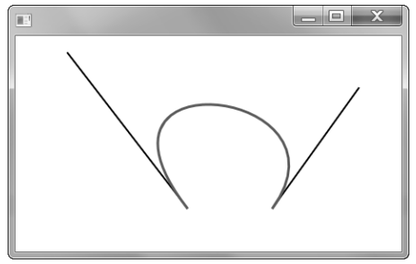

Abbildung 19.6 Screenshot der Anwendung CubicCurveDemo

19.5.4 Pfade *

Ein Pfad besteht aus zusammengesetzten Segmenten, die miteinander verbunden sind. Es ist ein wenig wie »von Punkt zu Punkt« malen, bei dem der Stift von einem Punkt zum anderen gezogen wird, aber auch abgesetzt und neu positioniert werden darf. Die Segmente bestehen nicht wie bei Polygonen ausschließlich aus Linien, sondern können auch quadratische oder kubische Kurven sein.

Die Klasse `javafx.scene.shape.Path` erbt von `Shape` und repräsentiert eine Liste dieser Pfadsegmente, die in JavaFX vom Typ `PathElement` sind. Die Pfadsegmente sind also Objekte und werden dem `Path` entweder am Anfang über den Konstruktor übergeben oder später der Liste angehängt. Von der abstrakten Basisklasse `PathElement` gibt es folgende Unterklassen: `LineTo`, `HLineTo`, `VLineTo`, `ArcTo`, `CubicCurveTo`, `QuadCurveTo`, `ClosePath` und `MoveTo`. Die letzten beiden Typen nehmen eine gewisse Sonderstellung ein, da `ClosePath` den Pfad abschließt und ein `MoveTo`-Objekt den Zeichenstift absetzt und neu positioniert; da der Startpunkt automatisch bei (0,0) liegt, kann `MoveTo` ihn auch zu Anfang gut setzen.

Beispiel

Zeichnen einer Linie als Pfad, Variante 1:

Listing 19.7 src/main/java/com/tutego/insel/javafx/PathDemo.java, Ausschnitt

```
Shape path = new Path( new MoveTo( 10, 10 ), new LineTo( 100, 20 ) );
```

Alternativ dazu Variante 2: erst das `Path`-Objekt aufbauen, dann die Liste der Segmente erfragen und schließlich anhängen:

```
Path path = new Path();
path.getElements().add( new MoveTo( 10, 10 ) );
path.getElements().add( new LineTo( 100, 20 ) );
```

Natürlich hätten wir in diesem Fall auch gleich ein `Polyline`- oder ein `Line`-Objekt nehmen können. Doch dieses Beispiel zeigt einfach, wie ein Pfad aufgebaut ist. Zunächst bewegen wir den Zeichenstift mit einem über den `MoveTo(double, double)`-Konstruktor erzeugten Objekt auf die Startposition, und anschließend zeichnen wir eine Linie mit einem über den Konstruktor `LineTo(double, double)` erzeugten Objekt.

Die Eigenschaften der Klasse `Path` sind daher auch übersichtlich:

```
class javafx.scene.shape.Path
extends Shape
```

- `Path()`
- `Path(Collection<? extends PathElement> elements)`
- `Path(PathElement... elements)`
- `ObservableList<PathElement> getElements()`

Des Weiteren gibt es eine `Property fillRule` für die Windungsregel und daher Setter/Getter.

Windungsregel *

Eine wichtige Eigenschaft der Pfade für gefüllte Objekte ist die *Windungsregel* (engl. *winding rule*). Diese Regel beschreibt eine Aufzählung `javafx.scene.shape.FillRule` und kann entweder `NON_ZERO` oder `EVEN_ODD` sein. Wenn Zeichenoperationen aus einer Form herausführen und wir uns dann wieder in der Figur befinden, sagt `EVEN_ODD` aus, dass dann innen und außen umgedreht wird. Wenn wir also zwei Rechtecke durch einen Pfad ineinander positionieren und der Pfad gefüllt wird, bekommt die Form in der Mitte ein Loch.

Das folgende Programm zeichnet ein blaues Rechteck mit `NON_ZERO` und ein rotes Rechteck mit `EVEN_ODD`. Mit der Konstanten `NON_ZERO` bei `setFillRule(FillRule)` wird das innere Rechteck mit ausgefüllt. Ausschlaggebend dafür, ob das innere Rechteck gezeichnet wird, ist die Anzahl der Schnittpunkte nach außen – »außen« heißt in diesem Fall, dass es unendlich viele

Schnittpunkte gibt. Diese Regel wird aber nur dann wichtig, wenn wir mit nichtkonvexen Formen arbeiten. Solange sich die Linien nicht schneiden, ist dies kein Problem:

Listing 19.8 src/main/java/com/tutego/insel/javafx/FillRuleDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  Rectangle rect1 = new Rectangle( 70, 70, 130, 50 );
  rect1.setFill( Color.YELLOW );

  Path rect2 = makeRect( 100, 80, 50, 50 );
  rect2.setFill( Color.BLUE );
  rect2.setFillRule( FillRule.NON_ZERO );

  Path rect3 =  makeRect( 200, 80, 50, 50 );
  rect3.setFill( Color.RED );
  rect3.setFillRule( FillRule.EVEN_ODD );

  Group group = new Group( rect1, rect2, rect3 );
  stage.setScene( new Scene( group, 300, 200 ) );
  stage.show();
}
```

Die eigene statische Methode `makeRect(int, int, int, int)` definiert den Pfad für die Rechtecke mit den Mittelpunktkoordinaten `x` und `y`. Das erste Rechteck besitzt die Breite `width` sowie die Höhe `height`, und das innere Rechteck ist halb so groß:

Listing 19.9 src/main/java/com/tutego/insel/javafx/FillRuleDemo.java, Ausschnitt

```
private static Path makeRect( int x, int y, int width, int height ) {
  Path p = new Path(
    new MoveTo( x + width/2, y - height/2 ), new VLineTo( y + height/2 ),
    new HLineTo( x - width/2 ), new VLineTo( y - height/2 ),
    new ClosePath(),
    new MoveTo( x + width/4, y - height/4 ), new VLineTo( y + height/4 ),
    new HLineTo( x - width/4 ), new VLineTo( y - height/4 ),
    new ClosePath() );

  return p;
}
```

Mit `MoveTo(double, double)` bewegen wir uns zum ersten Punkt. Die weiteren Linien-Direktiven formen das Rechteck. Die Form muss geschlossen werden, doch dieses Beispiel

macht durch das innere Rechteck anschaulich, dass die Figuren eines `Path`-Objekts nicht zusammenhängend sein müssen. Das innere Rechteck wird genauso gezeichnet wie das äußere.

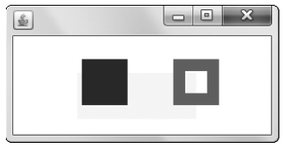

Abbildung 19.7 Die Windungsregeln NO_ZERO und EVEN_ODD

19.5.5 Polygone und Polylines

Eine *Polyline* besteht aus einer Menge von Linien, die einen Linienzug beschreiben. Dieser Linienzug muss nicht geschlossen sein. Ist er es dennoch, sprechen wir von einem *Polygon*. Für beides gibt es in JavaFX Klassen: `Polygon` und `Polyline`. Aufgebaut werden sie über den Standard-Konstruktor oder über einen parametrisierten Konstruktor, der mit `(double... points)` eine Liste von x-/y-Koordinaten annimmt. Die Klassen konvertieren die Punkte in eine interne `ObservableList<Double>`, die mit `getPoints()` zugänglich ist. Die Liste aus der Rückgabe kann problemlos verändert werden. Beide Objekte haben selbst keine Properties, daher gibt es neben `getPoints()` keine Setter/Getter in beiden Klassen.

19.5.6 Beschriftungen, Texte, Fonts

Eine einfach formatierte Beschriftung realisiert die Klasse `Text`. Gesetzt wird ein String an eine Position, wobei der String mit »\n« einen Umbruch enthalten kann, was JavaFX respektiert. Die Koordinaten bestimmen die Position der *Schriftlinie* – auch *Grundlinie* genannt (engl. *baseline*) –, auf der die Buchstaben stehen.

Damit ergeben sich die ersten drei Properties:

Property	Property-Typ	Funktion
`x` und `y`	`DoubleProperty`	x-/y-Koordinate
`Text`	`StringProperty`	Beschriftung

Tabelle 19.5 Properties von Text

Da die Eigenschaften oft gesetzt werden, nehmen zwei Konstruktoren sie gleich an:

- `Text(String text)`
- `Text(double x, double y, String text)`

Einen Standard-Konstruktor hat die Klasse natürlich auch. Die Klasse hat weitere Properties, die die Javadoc aufzeigt, etwa ob der Text durch-/unterstrichen ist oder nicht, die Ausrichtung, wo umbrochen werden soll oder den `FontSmoothingType`.

Zeichensätze/Fonts

JavaFX bringt die Unicode-Zeichenketten auf den Bildschirm, die aus so genannten *Glyphen* bestehen, das sind konkrete grafische Darstellungen eines Zeichens. Die Darstellung von Zeichen übernimmt in Java ein Font-Renderer, der in JavaFX fest verdrahtet ist.

Mit jeder Beschriftung ist standardmäßig ein Zeichensatz verbunden, der sich ändern lässt, üblicherweise mit `setFont(Font)`. Argument der Methode ist ein `javafx.scene.text.Font`-Objekt, das im Wesentlichen den Zeichensatz, die Größe und Ausrichtung repräsentiert. Aufbauen lässt sich das `Font`-Objekt über zwei Konstruktoren oder über fünf Fabrikmethoden:

```
class javafx.scene.text.Text
extends Shape
```

- `Font(double size)`
- `Font(String name, double size)`
- `static Font font(String family, double size)`
- `static Font font(String family, FontPosture posture, double size)`
- `static Font font(String family, FontWeight weight, double size)`
- `static Font font(String family, FontWeight weight, FontPosture posture, double size)`

Bis auf `FontPosture` (das eine Aufzählung mit `ITALIC` und `REGULAR` ist) und `FontWeight` (ebenfalls eine Aufzählung aus `THIN`, `EXTRA_LIGHT`, `LIGHT`, `NORMAL`, `MEDIUM`, `SEMI_BOLD`, `BOLD`, `EXTRA_BOLD`, `BLACK`) sind die Parameternamen selbsterklärend: Wir können den Namen des Zeichensatzes angeben und die Größe. Die Größe ist in Punkt angegeben, wobei 1 Punkt 1/72 Zoll (in etwa 0,376 mm) entspricht. Wird ein `Font` einem `Text` zugewiesen, der ja ein `Node` ist, kann der Knoten natürlich noch transformiert, etwa skaliert werden, sodass sich die Größe ändert.

Ist der Font einmal aufgebaut, gibt es die vier Getter, die von dem immutable Objekt die Zustände erfragen: `String getName()`, `double getSize()`, `String getStyle()` und `String getFamily()` – über JavaFX-Properties verfügt die Klasse nicht.

Hinweis

Die Dokumentation spricht zwar von »Punkt«, in Java sind aber Punkt und Pixel bisher identisch. Würde das Grafiksystem wirklich in Punkt arbeiten, müsste es die Bildschirmauflösung und den Monitor mit berücksichtigen.

Neue TrueType-Fonts in Java nutzen

Die auf allen Systemen vordefinierten Standardzeichensätze sind etwas dürftig, obwohl die `Font`-Klasse selbst jeden installierten Zeichensatz mit `getFontNames()` und `getFamilies()` einlesen kann. Da ein Java-Programm aber nicht von der Existenz eines bestimmten Zeichensatzes ausgehen kann, ist es praktisch, einen Zeichensatz mit der Installation auszuliefern und dann diesen zu laden; das kann die `Font`-Klasse mit `loadFont(…)` machen. Die beiden statischen Methoden `loadFont(InputStream in, double size)`, `loadFont(String urlStr, double size)` lesen einen Zeichensatz (in der Regel TrueType) ein und erstellen das entsprechende `Font`-Objekt. Es gibt die Rückgabe `null`, wenn JavaFX den Zeichensatz nicht einlesen kann, eine Ausnahme sollte es bei einem falschen Format nicht sein.

19.5.7 Die Oberklasse Shape

Die Klassen `Shape` und `Shape3D` erben von `Node`, was jeder Form eine Reihe von Fähigkeiten gibt, etwa:

- Methoden zur Transformation und zum Setzen von Effekten
- Abfangen von diversen Events
- Punkt-in-Form-Test
- Abfragen der umgebenden Box
- Setzen der Deckkraft (engl. *opacity*), das Gegenteil von Transparenz
- Nehmen eines Bildschirmabzugs (Screenshot) von der Form mit `snapshot(…)`

`Shape` ist selbst abstrakt, doch gibt es keine abstrakte Methode, die eine Unterklasse implementieren muss – die Klasse ist nur daher abstrakt, weil es keinen Sinn ergibt, von diesem speziellen Knotentyp ein Exemplar zu bilden.

Fähigkeiten jeder Form

Die Unterklassen von `Shape` bekommen Funktionalität nicht nur von `Shape`s Oberklasse `Node`, sondern `Shape` liefert den Formen selbst auch noch eine Reihe von Möglichkeiten, wie das Setzen von Muster oder Farbe. Insgesamt deklariert `Shape` folgende Properties:

Property	Property-Typ	Funktion
`smooth`	`BooleanProperty`	Antialiasing ein/aus
`fill`	`ObjectProperty<Paint>`	Wie das Innere der Form zu füllen ist, in der Regel mit Farbe

Tabelle 19.6 Properties eines Shape

Property	Property-Typ	Funktion
`stroke`	`ObjectProperty<Paint>`	Wie Außenlinien der Form zu zeichnen sind, in der Regel Linienfarbe
`strokeDashOffset`	`DoubleProperty`	Abstand im Linienmuster
`strokeLineCap`	`ObjectProperty<StrokeLineCap>`	Abschluss einer einzelnen Linie, zum Beispiel abgerundet
`strokeLineJoin`	`ObjectProperty<StrokeLineJoin>`	Form von Linienzusammenschlüssen
`strokeMiterLimit`	`DoubleProperty`	Präzisiert `strokeLineJoin`, veranschaulicht, wie »spitz« zwei Linien sich treffen.
`strokeWidth`	`DoubleProperty`	Stiftbreite
`strokeType`	`ObjectProperty<StrokeType>`	Ort, wo die Stiftbreite gilt, etwa außen um die Form

Tabelle 19.6 Properties eines Shape (Forts.)

Konstruktive Flächengeometrie *

Die Klasse `Shape` bietet statische Methoden, mit der sich zwei Formen zu neuen Formen verknüpfen lassen. Die Verknüpfungen sind Addition (Vereinigung), Subtraktion, und Schnitt (ein XOR gibt es nicht).

Die Signaturen der Methoden sind:

```
abstract class javafx.scene.shape.Shape
implements EventTarget
```

- `static Shape union(Shape shape1, Shape shape2)`
  Bildet eine Vereinigung zweier Formen. Ein Haus lässt sich zum Beispiel auf diese Weise durch ein `Polygon` mit dreieckiger Form, vereinigt mit einem `Rectangle`, darstellen.
- `static Shape subtract(Shape shape1, Shape shape2)`
  Schneidet die Form `shape2` aus `shape1` aus, sozusagen `shape1` minus `shape2`.
- `static Shape intersect(Shape shape1, Shape shape2)`
  Bildet den Schnitt von zwei Formen, also alles, wo beide Formen Pixel haben, bleibt im Ergebnis.

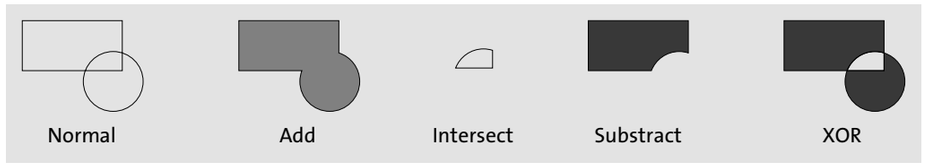

Abbildung 19.8 Beispiele für konstruktive Flächengeometrie mit Rechteck und Kreis

Die Verknüpfungen werden auch *CAG (Constructive Area Geometry)*, zu Deutsch *konstruktive Flächengeometrie*, genannt.

19.6 Füllart von Formen

Ist eine Form gefüllt, so gibt es verschiedene Möglichkeiten, diese Füllung zu beschreiben, wofür JavaFX den Typ `javafx.scene.paint.Paint` deklariert. Folgende Klassen aus dem `javafx.scene.paint`-Paket erweitern sie:

- `Color`: Repräsentiert sRGB-Farben und Alpha-Werte (Transparenz). Das ist die übliche Füllung.
- `ImagePattern`: Füllt Formen mit einer Kachel aus Grafiken.
- `LinearGradient`, `RadialGradient`: Füllen Formen mit Farbverläufen.

Die Zuweisung eines `Paint`-Objekts übernehmen Methoden wie `setFill(Paint)` oder `setStroke(Paint)`.

Bemerkung

Viele Formen werden standardmäßig gefüllt. Ist keine Füllung gewünscht, ist ein `setFill(null)` nötig.

19.6.1 Farben mit der Klasse Color

Der Einsatz von Farben und Transparenzen ist in Java-Programmen dank der Klasse `Color` einfach. Ein `javafx.scene.paint.Color`-Objekt repräsentiert einen Wert aus dem sRGB-Farbraum (Standard-RGB), andere Farbräume sind nicht vorgesehen.

Die Klasse `Color` stellt Konstanten für Standardfarben und einige Fabrikmethoden (keine Konstruktoren) sowie Anfragemethoden bereit. Außerdem gibt es Methoden, die abgewandelte `Color`-Objekte liefern – das ist nötig, da `Color`-Objekte wie `String` immutable sind. Die wichtigsten Methoden zum Erzeugen von `Color`-Objekten sind:


```
class javafx.scene.paint.Color
extends Paint
implements Interpolatable<Color>
```

- `static Color color(double red, double green, double blue)`
- `static Color color(double red, double green, double blue, double opacity)`
- `static Color rgb(int red, int green, int blue)`
- `static Color rgb(int red, int green, int blue, double opacity)`
  Liefert ein neues `Color`-Objekt aus den Bestandteilen Rot, Grün, Blau, einmal im Wertebereich 0.0 bis 1.0 und dann bei dem Parametertyp `int` im Bereich 0 bis 255.

Des Weiteren sind für hexadezimalkodierte Farben der Form `#rrggbb` die Methoden `web(String colorString)` – synonym dazu `valueOf(String)` – und `web(String colorString, double opacity)` deklariert. Bei Angabe von Farbton (auch Tönung genannt), Sättigung und Helligkeit sind `hsb(double hue, double saturation, double brightness)` und `hsb(double hue, double saturation, double brightness, double opacity)` von Nutzen. Die `web(…)`-Methode erlaubt die gleichen Strings wie auch bei CSS, so sind gültig `web("#f68", 0.5)` – was `color(1.0, 0.4, 0.8, 0.5)` entspricht – oder `web("rgb(255,50%,50%,0.25)", 0.5)` – was gleichkommt mit `color(1.0, 0.5, 0.5, 0.125)`.

Daneben gibt es zur Erzeugung von Grautönen ein paar Spezialmethoden: `gray(double gray)`, `gray(double gray, double opacity)`, `grayRgb(int gray)`, `grayRgb(int gray, double opacity)`.

Hinweis

Menschen unterscheiden Farben an den drei Eigenschaften Farbton, Helligkeit und Sättigung. Die menschliche Farbwahrnehmung kann etwa 200 Farbtöne unterscheiden. Diese werden durch die Wellenlänge des Lichts bestimmt. Die Lichtintensität und Empfindlichkeit unserer Rezeptoren lässt uns etwa 500 Helligkeitsstufen unterscheiden. Bei der Sättigung handelt es sich um eine Mischung mit weißem Licht. Hier erkennen wir etwa 20 Stufen. Unser visuelles System kann somit ungefähr 2 Millionen (200 × 500 × 20) Farbnuancen unterscheiden.

Zufällige Farbblöcke zeichnen

Um die Möglichkeiten der Farbgestaltung einmal zu beobachten, betrachten wir die Ausgabe eines Programms, das Rechtecke mit wahllosen Farben anzeigt:

Listing 19.10 src/main/java/com/tutego/insel/javafx/ColorDemo.java, Ausschnitt

```
Group group = new Group();
stage.setResizable( true );
stage.setScene( new Scene( group, 800, 800 ) );
Random r = new Random();
```

```
for ( int y = 12; y <  stage.getScene().getHeight() - 25; y += 30 )
  for ( int x = 12; x < stage.getScene().getWidth() - 25; x += 30 ) {
    Rectangle rect = new Rectangle( x, y, 25, 25 );
    Color color = Color.rgb( r.nextInt( 256 ), r.nextInt( 256 ), r.nextInt( 256 ) );
    rect.setFill( color );
    rect.setStroke( color.grayscale() );
    group.getChildren().add( rect );
  }
```

Das Fenster der Applikation hat eine gewisse Größe, die wir später in der Breite und Höhe abfragen, falls wir die Fensterausmaße einmal ändern wollen. Anschließend erzeugen wir Blöcke, die mit einer zufälligen Farbe gefüllt sind. `setFill(…)` übernimmt diese Aufgabe. Für den Rahmen der gefüllten Rechtecke setzen wir die Zeichenstiftfarbe mit der Methode `setStroke(…)`. Die Farbe des Randes leiten wir von der Zufallsfarbe ab.

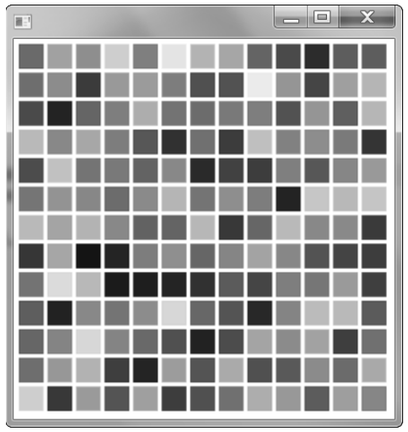

Abbildung 19.9 Programmierter Neoplastizismus

Vordefinierte Farben

Für den Fall, dass wir Farben benutzen wollen, sind schon an die 150 Werte vordefiniert, wie `Color.WHITE`, `Color.PLUM`. Für eine genauere Übersicht lohnt ein Blick in die Javadoc; neben den Farbkonstanten zeigt die Webseite auch ein Rechteck in der Farbe selbst. Die Farbnamen können auch in den Strings verwendet werden, da es bekannte HTML/CSS-Farben sind, etwa über `Color.web("orange")`.

Farbanteile zurückgeben

Mitunter müssen wir den umgekehrten Weg gehen und von einem gegebenen `Color`-Objekt wieder die Rot-, Grün-, Blau- oder Transparenzanteile extrahieren. Dies ist einfach, und die JavaFX-Bibliothek bietet Entsprechendes an:


```
class javafx.scene.paint.Color
extends Paint
implements Interpolatable<Color>
```

- `int getRed()`
- `int getGreen()`
- `int getBlue()`
  Liefert Rot-, Grün- und Blau-Anteile des Farbobjekts im Bereich von 0 bis 255.
- `int getOpacity()`
  Gibt den Alpha-Anteil zurück.
- `boolean isOpaque()`
  Ist die Farbe komplett durchgängig?
- `int getHue()`
- `int getSaturation()`
- `int getbrightness()`
  Liefert Farbton, Sättigung und Helligkeit.

Eine Methode wie `int getRGB()`, die die RGB-Farbe auf einen Schlag zurückgibt, existiert merkwürdigerweise nicht.

Die Farbmodelle HSB und RGB *

Zwei Farbmodelle sind in der Computergrafik geläufig: das RGB-Modell, bei dem die Farben durch einen Rot-, Grün- und Blau-Anteil definiert werden, und das HSB-Modell, das die Farben durch *Farbton* (engl. *hue*), *Farbsättigung* (engl. *saturation*) und *Helligkeit* (engl. *brightness*) definiert. Die Farbmodelle können die gleichen Farben beschreiben und ineinander umgerechnet werden. Dazu wird zunächst das `Color`-Objekt entweder mit `color(…)` oder mit `hsb(…)` aufgebaut und dann mit den gegenteiligen `getXXX()`-Methoden erfragt. Direkte Umrechenmethoden gibt es nicht.

Farbtöne ableiten

`brighter()` und `darker()` liefern ein Farbobjekt zurück, das jeweils eine Farbnuance heller bzw. dunkler ist. Diese beiden Methoden sind nützlich, um zum Beispiel bei angedeuteten 3D-Objekten die Ränder in einem helleren oder dunkleren Farbton zu zeichnen.

`desaturate()` ist eine Methode, die die Sättigung herausnimmt, `saturate()` verstärkt sie. `grayscale()` konvertiert komplett die Farbe in einen Grauton, und `invert()` dreht die Farbe um. Eine besondere Methode ist `interpolate(Color endValue, double t)`, die eine Farbe liefert, die von dem gegebenen Startwert hin zum Endwert laufen kann, wobei `t` von 0 bis 1 geht und sagt, wie nahe die Zielfarbe am Start- bzw. Endwert liegt.

19.7 Grafiken

Bilder sind neben dem Text das wichtigste visuelle Gestaltungsmittel. In Java können Grafiken an verschiedenen Stellen eingebunden werden, so zum Beispiel als Grafiken in Zeichengebieten (Canvas) oder als Icons in Schaltflächen, die angeklickt werden. Über Java können problemlos GIF-, PNG- und JPEG-Bilder geladen werden.

JavaFX bietet ein eigenes Paket `javafx.scene.image` mit diversen Typen rund um Bilder.

Hinweis

Das GIF-Format (Graphics Interchange Format) ist ein komprimierendes Verfahren, das 1987 von CompuServe-Betreibern zum Austausch von Bildern entwickelt wurde. GIF-Bilder können bis zu 1.600 × 1.600 Punkte umfassen. Die Komprimierung nach einem veränderten LZW[1]-Packverfahren hat keinen Einfluss auf die Bildqualität (sie ist verlustfrei). Jedes GIF-Bild kann aus maximal 256 Farben bestehen – bei einer Palette aus 16,7 Millionen Farben. Entsprechend dem Standard von 1989 können mehrere GIF-Bilder in einer Datei gespeichert werden. JPEG-Bilder sind dagegen in der Regel verlustbehaftet, und das Komprimierungsverfahren speichert die Bilder mit einer 24-Bit-Farbpalette. Der Komprimierungsfaktor kann prozentual eingestellt werden.

19.7.1 Klasse Image

JavaFX repräsentiert Grafiken als `javafx.scene.image.Image`-Objekte. Über den Konstruktor der `Image`-Klasse wird eine Quelle angegeben, aus der die Daten für das Bild kommen. Die beiden populären Konstruktoren sind:

- `Image(InputStream is)`
- `Image(String url)`

Ist die Grafik geladen, können Eigenschaften wie Höhe und Breite über Getter und auch über JavaFX-Properties erfragt werden.

Die Klasse `Image` repräsentiert das Bild, aber es ist kein Knoten für den Szenegraphen. Image-Objekte werden in JavaFX an unterschiedlicher Stelle eingesetzt:

- als Grafik für die Fensterdekoration
- als Vorgabe für den Maus-Cursor (`javafx.scene.ImageCursor`)
- als Bild, das auf eine Zeichenfläche (`javafx.scene.canvas.Canvas`) gemalt wird
- bei Effekten
- als Objekt, wenn eine Grafik im der Zwischenablage ist

1 Benannt nach den Erfindern Lempel, Ziv und Welch

Hinweis

Die Klasse `Image` hat in JavaFX zwei Funktionen: Einmal lädt sie Bilder, und einmal repräsentiert sie Bilder. Das ist in AWT/Swing anders, hier gibt es mit `Image` eine Bildrepräsentation, aber das Laden/Speichern übernehmen andere Klassen, etwa `ImageIO`.

19.7.2 ImageView

Um eine Grafik in den Szenegraphen zu hängen, wird sie in eine `ImageView` verpackt; `ImageView` ist eine direkte Unterklasse von `Node`. Zum Aufbau gibt es drei Konstruktoren:

- `ImageView()`
- `ImageView(Image image)`
- `ImageView(String url)`

Die Grafik kann mit den JavaFX-Properties `fitHeight` und `fitWidth` (beide `DoubleProperty`) in eine Box gepresst werden. Eine Skalierung geschieht automatisch, und die `BooleanProperty preserveRatio` bestimmt, ob das Verhältnis von Breite zu Höhe gleich bleibt. Eine weitere `BooleanProperty smooth` sagt, ob eine bessere oder schnellere Variante zur Berechnung einer skalierten Version verwendet werden soll. Mit einem Viewport lässt sich ein Teil der Grafik zeigen.

Beispiel

Erzeuge eine `ImageView` mit einer Grafik `image`. Zeige von der Grafik einen Ausschnitt (Viewport), und skaliere die Grafik auf 200 Pixel in der Höhe, wobei sich die Breite im Verhältnis mit ändern soll. Die Skalierung soll eine gute Qualität haben und das Ergebnis aus Performance-Gründen im Cache gehalten werden.

```
ImageView node = new ImageView( image );
Rectangle2D viewportRect = new Rectangle2D( 10, 10, 80, 80 );
node.setViewport( viewportRect );
node.setFitHeight( 200 );
node.setPreserveRatio( true );
node.setSmooth( true );
node.setCache( true );
```

Icon-Sammlungen *

Wer für seine grafischen Oberflächen Icons einsetzt, der findet beim *Tango Desktop Project* (*http://tango.freedesktop.org/*) viele Standard-Icons in den Auflösungen 16 × 16, 22 × 22, 32 × 32 und ebenso im SVG-Format. Die Website *http://www.iconfinder.net/* bietet eine Suche nach bestimmten Begriffen und findet freie Icons nach weiteren Kriterien wie Hintergrund-

farbe/Transparenz, Größe wie auch kommerziell nutzbare Icons. *Crystal Clear* (*http://commons.wikimedia.org/wiki/Crystal_Clear*) steht unter der Lizenzform LGPL und ist damit auch für kommerzielle Anwendungen nutzbar.

19.7.3 Programm-Icon/Fenster-Icon setzen

Zumindest unter Windows ist jedem Fenster ein kleines Bildchen zugeordnet, das ganz links in der Titelzeile untergebracht ist. Das Programm-Icon lässt sich in JavaFX über das `Stage`-Objekt ändern; wir erinnern uns, dieses Objekt wird in der JavaFX-Startmethode `start(Stage stage)` übergeben. Die `Stage` hat assoziierte Icons, die mit `getIcons()` erfragt werden, das Ergebnis ist vom Typ `ObservableList<Image>`. Interessant ist die Tatsache, dass es nicht nur ein Icon geben kann, sondern mehrere. Eine Grafik der Größe 16 × 16 Pixel ist sinnvoll, doch auch größere Größen sind angebracht, weil das Icon auch beim Minimieren der Anwendung gezeigt wird. Jedes Icon wird aber immer auf die notwendige Größe skaliert.

Beispiel

Setze ein Programm-Icon:

```
Image programIcon = …
stage.getIcons().add( programIcon );
```

19.7.4 Zugriff auf die Pixel und neue Pixel-Bilder *

Die Klasse `Image` hat keine Methode zum Abfragen und Setzen eines Farbwerts an einer bestimmten Koordinate, das ist ausgelagert in die Extratypen `PixelReader` und `PixelWriter`.

Klasse PixelReader

Ein `Image`-Objekt liefert mit `getPixelReader()` diesen `PixelReader`, und dann sind zum Beispiel mit `int getArgb(int x, int y)` und `Color getColor(int x, int y)` Abfragen möglich oder auch ganze Bereichsanfragen mit Feld-Rückgabe.

Beispiel

Lies die Farbwerte eines `Image`-Objekts `image`, und zerlege die Rückgabe in die Farbwerte Rot, Grün, Blau und den Alpha-Wert:

```
int argb  = image.getPixelReader().getArgb( x, y );
int alpha = (argb >> 24) & 0xff;
int red   = (argb >> 16) & 0xff;
int green = (argb >> 8)  & 0xff;
int blue  = (argb)       & 0xff;
```


Die Methode `getArgb(…)` liefert als Rückgabe einen Wert im Standard-RGB-Modell – unabhängig von der tatsächlichen physikalischen Kodierung – und im Standard-RGB-Farbraum. Sind nur die rohen Farbwerte nötig, würde auch `getColor(…)` zum Ziel führen, doch hier hätten wir es immer mit einem Zwischenobjekt zu tun, was bei vielen Abfragen nicht so optimal ist.

Ausblick

Grafiken müssen im Speicher repräsentiert werden, und dafür gibt es ein Speichermodell. Anschaulich gesagt drückt es aus, wie im Speicher die Farbinformationen eines Bildes abgelegt sind, ob etwa je 8 Bit für Rot/Grün/Blau oder ob, wie bei GIF, ein Index und eine Farbtabelle verwendet werden. JavaFX ist bei den Farb- und Speichermodellen flexibel und abstrahiert in einen Typ `javafx.scene.image.PixelFormat`.

WritableImage und PixelWriter

Nicht immer kommen die Bilder vom Datensystem oder aus dem Internet. Mit der JavaFX-Bibliothek lassen sich einfach neue Grafiken anlegen und die Pixel setzen. Dazu bietet JavaFX eine Unterklasse von `Image`, und zwar `WritableImage`. Während `Image` selbst immutable ist, erlaubt `WritableImage` eine Veränderung der Pixel.

Da `WritableImage` von `Image` erbt, erbt es auch alle Eigenschaften und Properties, wie etwa `getPixelReader()`. Es kommen nur drei Konstruktoren und eine Methode hinzu, die Klasse ist also schlank:

```
class javafx.scene.image.WritableImage
extends Image
```

- `WritableImage(int width, int height)`
- `WritableImage(PixelReader reader, int width, int height)`
- `WritableImage(PixelReader reader, int x, int y, int width, int height)`
  Initialisiert ein `WritableImage` mit einer Größe. Ein anderes Bild kann von einem `PixelReader` übernommen werden, es entsteht also eine Kopie.
- `PixelWriter getPixelWriter()`
  Liefert einen `PixelWriter`, der schreibenden Zugriff auf die Pixel des Bildes ermöglicht.

Bei `WritableImage` finden wir die Methode `getPixelWriter()`, die symmetrisch ist zum `PixelReader` und entsprechende `getXXX(…)`-Methode bietet.

Ein Beispiel zum Lesen und Schreiben von Pixeln

Das folgende Programm lädt ein Bild und setzt einen Bewegungssensor an die `ImageView`, in der ein `WritableImage` eingebettet ist. Bewegt der Nutzer den Mauszeiger über das Bild,

erfragen wir den Farbwert unter der Koordinate und lassen uns über die `Color`-Klasse einen invertierten Farbton geben, womit wir den vorherigen Farbwert für das Pixel überschreiben:

Listing 19.11 src/main/java/com/tutego/insel/javafx/WritableImageDemo.java, Ausschnitt

```
private ImageView getImageView( String filename ) {
  try ( InputStream stream = getClass().getResourceAsStream( filename  ) ) {
    if ( stream == null )
      return new ImageView();

    Image img = new Image( stream );
    WritableImage wrImg = new WritableImage( img.getPixelReader(),
                          (int) img.getWidth(), (int) img.getHeight() );
    ImageView imageView = new ImageView( wrImg );

    imageView.setOnMouseMoved( e -> {
      Color c = wrImg.getPixelReader().getColor( (int) e.getX(), (int) e.getY() );
      wrImg.getPixelWriter().setColor( (int) e.getX(), (int) e.getY(), c.invert() );
    } );
    return imageView;
  }
  catch ( IOException e ) { // Exception von close()
    throw new UncheckedIOException( e );
  }
}
```

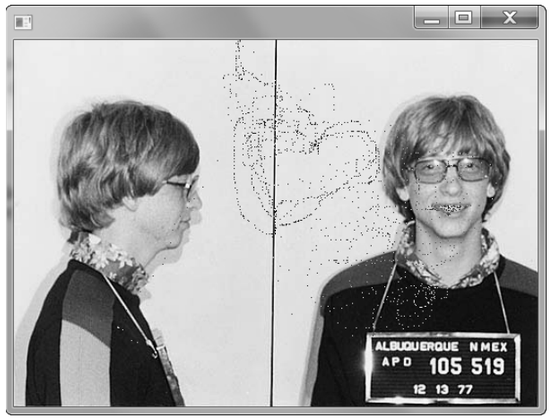

Abbildung 19.10 Screenshot der Anwendung WritableImageDemo

19.8 Deklarative Oberflächen mit FXML

Den Szenegraphen über Java-Programmcode aufzubauen, ist eine Möglichkeit, doch JavaFX erlaubt es auch, die Objekte über XML zu konfigurieren, in so genannten *FXML-Dateien*. Das macht es viel einfacher, grafische Oberflächen über GUI-Builder aufzubauen und sauber das Layout (was) vom Code (wie) zu trennen.

Die hierarchische Struktur von XML passt natürlich prima zu der Hierarchie, die es bei grafischen Oberflächen gibt: Ein Fenster enthält Container, die wiederum Elemente enthalten usw. Für unser kleines Beispiel soll eine Oberfläche drei Elemente bieten: eine Beschriftung, ein Textfeld und eine Schaltfläche. Drückt der Anwender die Schaltfläche, soll der Text im Textfeld in Großbuchstaben konvertiert werden.

FXML-Datei

Die XML-Datei sieht so aus:

Listing 19.12 src/main/resources/com/tutego/insel/javafx/covert2UpperCase.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>

<HBox xmlns:fx="http://javafx.com/fxml"
      fx:controller="com.tutego.insel.javafx.ButtonController">
  <children>
    <Label text="Eingabe: " />
    <TextField fx:id="input" />
    <Button text="Konvertiere" onAction="#convertAction" />
  </children>
</HBox>
```

Die Hierarchie ist gut zu erkennen. Die interessanten Dinge sind andere:

1. Zu Beginn gibt es eine Art `import`-Anweisung, um Typnamen nicht voll qualifizieren zu müssen. Die Typen in FXML (etwa `HBox`, `Label`) heißen genauso wie die Klassennamen. Natürlich könnten auch eigene Klassen über `<?import>` bekannt gemacht werden.
2. `HBox` hat zwei Attribute: Das erste, `xmlns:fx`, deklariert den Namensraum `fx` in der XML-Datei. Dieses Attribut ist keine spezifische Eigenschaft der `HBox`, sondern würde bei allen XML-Wurzelelementen genutzt werden. Das zweite Attribut spezifiziert den *Controller*, der später die Ereignisbehandlung für den Klick übernimmt.
3. Das `TextField` bekommt mit dem Attribut `id` eine ID zugewiesen, unter der das Textfeld später erfragt werden kann. JavaFX geht noch einen Schritt weiter und bildet das Objekt

mit der ID automatisch auf ein Attribut der Controller-Klasse ab. Label und Schaltfläche brauchen keine IDs, da sie nicht erfragt werden müssen.

4. Das Attribut `onAction` der Schaltfläche referenziert Programmcode, der immer aufgerufen wird, wenn die Schaltfläche angeklickt wird. Hier kann direkt Java-Quellcode stehen, oder, wie in unserem Fall, ein # und der Name einer Methode, die in einem Controller deklariert werden muss. Den Klassennamen des Controllers haben wir am Wurzelelement deklariert.

Controller-Klasse

Die Ereignisbehandlung ist komplett aus der FXML-Datei herausgezogen und wandert in Controller-Klassen. Die eigene Klasse `ButtonController`, die voll qualifiziert bei `fx:controller` genannt wurde, enthält:

Listing 19.13 src/main/java/com/tutego/insel/javafx/ButtonController.java

```
package com.tutego.insel.javafx;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.TextField;

public class ButtonController {

  @FXML
  private TextField input;

  @FXML
  protected void convertAction( ActionEvent event ) {
    input.setText( input.getText().trim().toUpperCase() );
  }
}
```

Drei Dinge springen ins Auge:

1. Die Controller-Klasse erweitert keine Schnittstelle.
2. Die Annotation `@FXML` sagt, dass der Verweis auf das `TextField`-Objekt aus dem Szenegraphen in die Variable `input` injiziert werden soll.
3. Da in der FXML-Datei an der Schaltfläche ein `onAction="#convertAction"` steht, muss das einer Methode zugeordnet werden. Die Annotation `@FXML` an der Methode unter dem Namen `convertAction` stellt diese Beziehung her.

Hauptanwendung

Das Hauptprogramm ist nun relativ einfach:

Listing 19.14 src/main/java/com/tutego/insel/javafx/FXMLDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) throws Exception {
  Parent p = FXMLLoader.load( getClass().getResource( "covert2UpperCase.fxml" ) );
  stage.setScene( new Scene( p, 500, 400 ) );
  stage.show();
}
```

Weitere Informationen liefert *http://download.java.net/jdk8/jfxdocs/javafx/fxml/doc-files/introduction_to_fxml.html*.

19.9 Diagramme (Charts)

Neben den Standardkomponenten bietet JavaFX zusätzliche Klassen für Diagramme – und die sehen toll aus. Als Basisklasse deklariert JavaFX dazu `Chart`, wovon es zwei zentrale Ableitungen gibt: `PieChart` für Kuchendiagramme und `XYChart` für alle Charts, die Daten irgendwie Koordinaten im zweidimensionalen Raum zuordnen. Weil es unterschiedliche 2D-Diagrammtypen gibt, bietet das Framework weitere Unterklassen von `XYChart`. Die Klassenhierarchie sitzt vollständig im Paket `javafx.scene.chart` und sieht so aus:

```
javafx.scene.chart.Chart
  PieChart
  XYChart<X,Y>
    AreaChart<X,Y>
    BarChart<X,Y>
    BubbleChart<X,Y>
    LineChart<X,Y>
    ScatterChart<X,Y>
```

Als Beispiel für `XYChart` wollen wir gleich ein Balkendiagramm aufbauen. Eine ganze Reihe anderer Diagramme liegen für Entwickler zur Introspektion zugänglich unter dem Verzeichnis *ChartsSampler\src\chartssampler* der JavaFX-Demos.

19.9.1 Kuchendiagramm

Kuchendiagramme sind mit JavaFX sehr einfach aufzubauen. Es muss lediglich eine spezielle Sammlung mit `String`-`double`-Paaren (für Titel und Wert) der `PieChart`-Klasse übergeben werden:

Listing 19.15 src/main/java/com/tutego/insel/javafx/BarChartDemo.java

```
package com.tutego.insel.javafx;

import java.util.Map;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.stage.Stage;

public class PieChartDemo extends Application {
  @Override
  public void start( Stage stage ) {

    ObservableList<PieChart.Data> data = FXCollections.observableArrayList();
    Map.of( "Black Sun", 123, "Long Delight", 88, "Careless Whisper", 45, "Silky Milky", 30 )
       .forEach( (k, v) -> data.add( new PieChart.Data( k, v ) ) );

    PieChart chart = new PieChart( data );
    chart.setTitle( "Die beliebtesten Haarpflegeprodukte" );

    stage.setScene( new Scene( chart ) );
    stage.show();
  }

  public static void main( String[] args ) {
    launch( args );
  }
}
```

Das Beispiel zeigt, dass der `PieChart`-Konstruktor die Dateien in einem speziellen Container vom Typ `ObservableList` erwartet. Eine Implementierung der Schnittstelle wird mit der Methode `observableArrayList(…)` der Utility-Klasse `FXCollections` bereitgestellt. Die Methode `observableArrayList(…)` ist überladen, und hier können gleich über eine variable Argumentliste die Elemente übergeben werden. Das `PieChart` erwartet in der Sammlung `Data`-Objekte, die den Titel und einen Wert angeben. Die Werte werden automatisch in der richtigen Proportion auf den 360-Grad-Kreis gebracht.

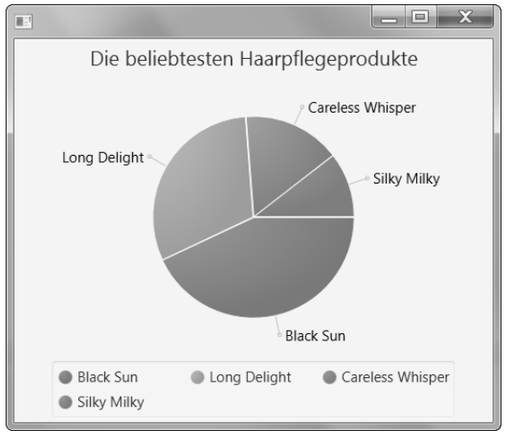

Abbildung 19.11 Screenshot des Diagramms

19.9.2 Balkendiagramm

Die Diagrammklasse `BarChart` ist einfach zu nutzen; nur die Vorbereitung der Daten ist etwas Schreibarbeit. Alles ist generisch deklariert, und so wimmelt es in JavaFX-Diagramm-Programmen nur so von spitzen Klammern.

Beginnen wir mit einem kompletten Beispiel zur Darstellung der Bevölkerungsentwicklung in Deutschland, und betrachten wir es dann im Anschluss genauer:

Listing 19.16 src/main/java/com/tutego/insel/javafx/BarChartDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  CategoryAxis xAxis = new CategoryAxis();
  xAxis.setLabel( "Jahrgang" );

  NumberAxis yAxis = new NumberAxis( "Bevölkerung",
                                     0        /* lowerBound */,
                                     90000000 /* upperBound */,
                                     30000000 /* tickUnit */ );

  ObservableList<XYChart.Data<String,Number>> series
      = FXCollections.observableArrayList(
         new XYChart.Data<String,Number>( "1950", 69_346_000 ),
         new XYChart.Data<String,Number>( "1960", 73_147_000 ),
         new XYChart.Data<String,Number>( "1970", 78_069_000 ),
         new XYChart.Data<String,Number>( "1980", 78_397_000 ),
```

```
            new XYChart.Data<String,Number>( "1990", 79_753_000 ),
            new XYChart.Data<String,Number>( "2000", 82_260_000 ),
            new XYChart.Data<String,Number>( "2010", 81_752_000 ),
            new XYChart.Data<String,Number>( "2015", 82_175_684 )
        );

    ObservableList<XYChart.Series<String,Number>> data =
         FXCollections.observableArrayList();
    data.add( new BarChart.Series<>( series ) );

    BarChart<String,Number> chart = new BarChart<>( xAxis, yAxis );
    chart.setTitle( "Bevölkerungsentwicklung in Deutschland" );
    chart.setData( data );

    stage.setScene( new Scene( chart, 500, 400 ) );
    stage.show();
}
```

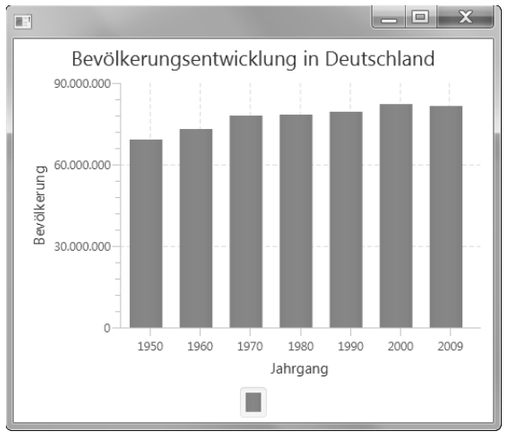

Abbildung 19.12 Screenshot des Balkendiagramms

Zum Aufbau eines Balkendiagramms lassen sich drei Schritte ausmachen:

1. **Aufbau der Achsen**: Achsen sind in JavaFX durch den Basistyp `Axis<T>` repräsentiert und werden später dem Diagramm zugewiesen. Von `Axis` gibt es Unterklassen wie `CategoryAxis` und `ValueAxis` (mit der Unterklasse `NumberAxis`). Wir nutzen beide Typen, denn `CategoryAxis` ist für Strings gedacht und `NumberAxis` für numerische Werte.
2. **Datenserien aufbauen**: Die Daten kommen wieder in den speziellen JavaFX-Container `ObservableList` und sind vom Typ `Data`, einer inneren Klasse von `XYChart`. Die enthaltenen Datentypen können beliebig sein, daher ist die Klasse generisch deklariert. In unserem

Fall sind die Typen `String` (x-Achse) und `Double` (`Number` für y-Achse). Nachdem eine Serie aufgebaut wurde, kommt diese Serie in einen anderen Container. Das liegt daran, dass etwa ein Balkendiagramm für die Bevölkerungsentwicklung auch mehrere Länder darstellen könnte, etwa für Deutschland Balken in Gelb und für Nigeria Balken in Rot. Der Container kann befüllt werden – einmal, indem die Elemente direkt der Methode `observableArrayList(…)` übergeben werden (erster Fall), oder nachträglich über `add(…)` wie im zweiten Fall.

3. **Chart fertigstellen**: Die Diagrammklasse typisieren wir mit `BarChart<String,Number>` und setzen alle Eigenschaften: die Achsen über den Konstruktor, den Titel und die Daten über Setter.

19.10 Animationen

Grafische Programme müssen sich in AWT oder Swing selbst um Animationen kümmern, etwa indem sie einen Timer-Thread starten und dann ein Neuzeichnen (Repaint) motivieren. In der Zeichenroutine müssen sich die Komponenten dann selbst bewegen, ausblenden – oder was auch immer die Animation fordert.

JavaFX bringt Animationsunterstützung mit und unterstützt Entwickler in dem Trend, dass moderne grafische Oberflächen heutzutage nicht einfach Komponenten irgendwo hinsetzen oder sie verschwinden lassen, sondern den Betrachter subtil über Veränderungen informieren.

Bei Animationen werden Eigenschaften wie Farbe, Transparenz oder Position verändert. Ein zentraler Unterschied ist nur, ob eine Animation etwa eine Bewegung automatisch vornimmt oder ob wir selbst die Koordinaten für eine Bewegung von Hand setzen. Daher unterscheidet JavaFX zwei Typen von Animationen:

- **Übergangsanimationen (Transitions)**: Dazu zählen automatische Animationen, wie das Ausblenden, die Skalierung oder die Bewegung entlang eines Pfades. Es ist lediglich der Start- und Endzustand anzugeben, und den Rest regelt JavaFX automatisch. Die Transitionen können auch kombiniert werden, etwa sequenziell hintereinander, oder parallel gepaart werden.
- **Zeitleistenanimationen (Timeline)**: Bei den Transitionen geschieht innerhalb einer gewissen Zeit eine Animation, aber es ist nicht offensichtlich, bei welchem Zeitpunkt welche Koordinate oder welcher Zustand modifiziert wird. Das ist bei den Timeline-Animationen anders. Hier stehen die Zeitpunkte im Mittelpunkt, an denen etwas passiert. Innerhalb dieser Zeitleiste arbeitet das System Key-Frames ab. Zwischen dem Start-Key-Frame und dem End-Key-Frame können wir Properties verändern (den `KeyValue` setzen) oder uns Ticks, also Events, generieren lassen, an denen wir selbst individuell Eigenschaften ändern können.

19.10.1 FadeTransition

Zum Ausblenden von Knoten dient die `FadeTransition`. Die zentralen Parameter sind Dauer, Starttransparenz, Endtransparenz oder das Ein-/Ausblenden, das zyklisch ist. Das nächste Beispiel zeigt, wie der Klick auf die Schaltfläche diese verschwinden lässt:

Listing 19.17 src/main/java/com/tutego/insel/javafx/FadeTransitionDemo.java, Ausschnitt

```
Button b = new Button( "Klick mich so hart du kannst" );
b.setOnAction( e -> {
  FadeTransition t = new FadeTransition( Duration.seconds( 0.5 ), b );
  t.setFromValue( 1.0f );
  t.setToValue( 0.0f );
  t.play();
} );
stage.setScene( new Scene( b ) );
stage.show();
```

Die Vorbelegung der `From`/`To`-Werte ist 1, sodass `setFromValue(1.0f)` unnötig ist.

Wer einen Effekt mit »aus- und dann wieder einblenden« wünscht, also von Animation A1 auf eine A2 gehen möchte und zurück, also A1 → A2 → A1, der kann die Zeilen

```
t.setCycleCount( 2 );
t.setAutoReverse( true );
```

mit aufnehmen. Dann gibt es zwei Animationen, bei denen nach dem ersten Ablauf JavaFX die zweite Animation von hinten nach vorne abspielt. Soll endlos animiert werden, ist ein Aufruf von `setCycleCount(Timeline.INDEFINITE)` möglich.

19.10.2 ScaleTransition

Die `ScaleTransition` skaliert ein Objekt in der x/y-Achse. Werte kleiner 1 lassen es schrumpfen, und Werte größer 1 blasen es auf. Unser nächstes Beispiel zeigt, wie eine Schaltfläche innerhalb von 2 Sekunden auf das Doppelte in der Horizontalen und gegen null auf der Vertikalen skaliert wird:

Listing 19.18 src/main/java/com/tutego/insel/javafx/ScaleTransitionDemo.java, Ausschnitt

```
Button button = new Button( "Klick mich" );
button.setOnAction( e -> {
  ScaleTransition t = new ScaleTransition( Duration.seconds( 2 ), button );
  t.setFromX( 1 );
  t.setFromY( 1 );
```



```
  t.setToX( 2 );
  t.setToY( 0 );
  t.play();
} );

stage.setScene( new Scene( new VBox( new Label( "Vor" ),button, new Label( "Nach" ) ) ) );
stage.show();
```

Standardmäßig stehen auch hier die Werte auf 1, sodass wir uns `setFromX(1)` und `setFromY(1)` hätten sparen können.

19.10.3 Transitionen parallel oder sequenziell durchführen

Transitionen lassen sich zu Folgen verbinden, wobei zu unterscheiden ist, ob JavaFX die Transitionen parallel oder sequenziell abarbeitet. Dazu wird ein Objekt vom Typ `ParallelTransition` oder `SequentialTransition` aufgebaut, und dem Container werden dann Transitionen als Kinder zugewiesen.

Unser nächstes Beispiel blendet die Schaltfläche aus und lässt gleichzeitig die y-Achse auf 0 schrumpfen:

Listing 19.19 src/main/java/com/tutego/insel/javafx/ParallelTransitionDemo.java, Ausschnitt

```
@Override
public void start( Stage stage ) {
  final Button b = new Button( "Klick mich" );
  b.setOnAction( new EventHandler<ActionEvent>() {
    @Override public void handle( ActionEvent e ) {
      ScaleTransition t1 = new ScaleTransition( Duration.seconds( 2 ), b );
      t1.setToY( 0 );

      FadeTransition t2 = new FadeTransition( Duration.seconds( 2 ), b );
      t2.setToValue( 0.0f );

      ParallelTransition parallelTransition = new ParallelTransition();
      parallelTransition.getChildren().addAll( t1, t2 );
      parallelTransition.play();
    }
  } );
  stage.setScene( new Scene( b ) );
  stage.show();
}
```

19.11 Medien abspielen

JavaFX unterstützt das Abspielen von Audio- und Videoformaten, wobei es im Moment gar nicht so schlecht aussieht, da Oracle auf die Container-Formate/Codecs der quelloffenen GStreamer-Bibliothek[2] zurückgreift und so MP3, WAV, ACC, AIFF, PCM, MPEG-4 (H.264/AVC mit AAC) und FLV (VP6 mit MP3) abspielen kann.[3]

Die API ist sehr einfach, und um einen Hintergrundsound abzuspielen, ist bloß ein Einzeiler nötig. Folgende drei Typen im Paket `javafx.scene.media` stehen im Mittelpunkt:

- `Media`: Die Klasse repräsentiert eine Audio- oder Videoressource. Sie wird über einen URI aufgebaut und hat Eigenschaften wie Dauer, Ausmaße (bei Videos) und Metadaten.
- `MediaPlayer`: Die Klasse repräsentiert Zustände eines Medienspielers mit Eigenschaften wie Lautstärke, Anfangs-/Endposition und Abspielgeschwindigkeit und bietet Methoden wie `play()`, `pause()` oder `stop()`. Dabei kann der Media-Player Ereignisse auslösen. Eine grafische Repräsentation ist der `MediaPlayer` nicht.
- `MediaView`: Die `MediaView`-Klasse stellt das Video tatsächlich dar und lässt sich als normaler JavaFX-Knoten in den Szenegraphen setzen.

Beispiel

Binde eine Videowiedergabe ein, die nach dem Laden sofort losspielt:

```
URI uri = new File( path ).toURI();
Media media = new Media( uri.toString() );
MediaPlayer player = new MediaPlayer( media );
player.setAutoPlay( true );
MediaView mediaView = new MediaView( player );
stage.setScene( new Scene( new Group( mediaView ) ) );
```

Für kleine Audiodateien gibt es eine spezielle Klasse `AudioClip`, die ohne große Verarbeitung schnell Audiodateien abspielt. Die Dateien sollten nicht zu groß sein, denn sie werden unkomprimiert im Speicher gehalten.

Beispiel

Spiele eine Audiodatei ab:

```
new AudioClip( "http://....." ).play();
```

2 Siehe *https://gstreamer.freedesktop.org/*

3 Unterstützte Formate zählt auch die Javadoc unter *http://download.java.net/jdk8/jfxdocs/javafx/scene/media/package-summary.html* auf.

19.12 Java 3D

Neben zweidimensionalen Grafiken besitzt JavaFX Klassen und Typen für dreidimensionale Objekte, etwa `Box`, das wie `Cylinder`, `MeshView` und `Sphere` von `javafx.scene.shape.Shape3D` abgeleitet ist. Zusätzlich muss eine Kamera positioniert werden, und eine kleine Box ist schnell positioniert; so könnte eine `start(…)`-Methode aussehen:

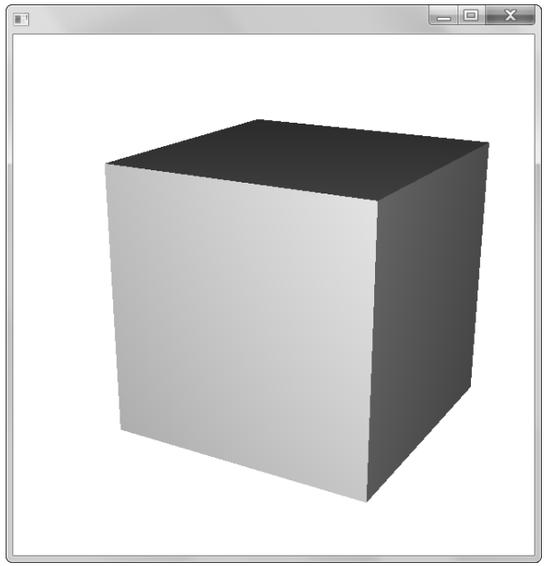

Abbildung 19.13 Blick auf die Box

```
Box box = new Box( 5, 5, 5 );
PerspectiveCamera camera = new PerspectiveCamera( true );
camera.getTransforms().addAll( new Translate( 8, -6, -15 ),
                               new Rotate( -30, Rotate.Y_AXIS ),
                               new Rotate( -20, Rotate.X_AXIS ) );
Scene scene = new Scene( new Group( camera, box ), 1024, 768 );
scene.setCamera( camera );
stage.setScene( scene );
stage.show();
```

19.13 Das Geometry-Paket *

Punkte selbst lassen sich nicht in den Szenegraphen einfügen, weil Punkte keine Ausdehnung haben. Wer also einen Punkt benötigt, muss etwa einen Minikreis, ein Rechteck oder eine Linie mit einer sehr kleinen Ausdehnung zeichnen.

Auch wenn JavaFX kein spezielles `Shape`-Objekt für Punkte bietet, so gibt es dennoch Klassen, die Punkte repräsentieren können, und mit ihnen Hilfsmethoden. Denn geometrische Fragen, etwa wie nach dem Abstand eines Punktes zu einer Linie oder ob sich zwei Boxen schneiden, werden nicht von den JavaFX-`Shape`-Objekten beantwortet, sondern von Klassen aus dem `javafx.geometry`-Paket. Diese sind sehr leichtgewichtige Objekte, haben keinerlei Listener oder JavaFX-Properties, sondern nur einfache Setter/Getter und Hilfsmethoden. Alle folgenden Klassen sind immutable und bekommen ihre Werte einmalig über den Konstruktor:

- `Point2D`: zweidimensionaler Punkt, repräsentiert durch x/y-Koordinaten. Beantwortet etwa Fragen zum Abstand von zwei Punkten oder berechnet den Mittelpunkt zwischen zwei Punkten. `Point2D` wird auch verwendet, um Vektoren zu spezifizieren, bei denen dann der Ursprung bei (0,0) liegt. `Point2D` beantwortet auch Fragen, wie etwa nach dem Winkel zwischen zwei Vektoren, oder berechnet das Kreuzprodukt.
- `Point3D`: dreidimensionaler Punkt, gegeben durch die x/y/z-Koordinaten. Die Methoden sind wie bei `Point2D`, also etwa Abstandsberechnung, Mittelpunkt, Winkelberechnung.
- `Rectangle2D`: Rechteck im zweidimensionalen Raum, gegeben durch Startkoordinaten und Höhe, Breite. Beantwortet, ob ein Punkt im Rechteck liegt oder ob sich ein anderes Rechteck im Rechteck befindet oder es schneidet.
- `Dimension2D`: Ausmaße eines Objekts, gegeben durch Höhe und Breite. Es gibt außer Getter keine weiteren Anfragemethoden.
- `Insets`: Abstände zu oben, unten, rechts, links. Es gibt nur Getter, keine weiteren Anfragemethoden.
- `Bounds`, `BoundingBox`: Die Klasse `BoundingBox` erweitert die abstrakte Klasse `Bounds` und dient der Beschreibung von Ausmaßen und Grenzen von JavaFX-Knoten. JavaFX-`Node`-Objekte liefern zum Beispiel mit `getBoundsInLocal()`/`getBoundsInParent()` die Informationen zur Lage des Objekts als `Bounds`-Objekt.

Java 2D-API versus JavaFX

Bei den Java Foundation Classes sind diese Hilfsmethoden in den Klassen selbst integriert, oft als statische Methoden. So vereinigt `java.awt.geom.Rectangle2D` Funktionalität der JavaFX-Klassen `javafx.scene.shape.Rectangle` und `javafx.geometry.Rectangle2D`. Auch fehlt bei JavaFX eine Methode, die testet, wo sich außerhalb eines Rechtecks ein Punkt befindet. Da JavaFX im `javafx.geometry`-Paket keine Linien kennt, bleiben einige Fragen unbeantwortet, etwa die Frage, ob ein Punkt links oder rechts einer Linie liegt, was die JFC Klasse `Line2D` beantworten kann.

19.14 JavaFX-Scene in Swing-Applikationen einbetten

Obwohl JavaFX von Swing bzw. AWT vollständig entkoppelt ist, ist es doch nötig, beide Technologien zusammenzuführen. Für die nächste Zeit wird Swing (noch) die bevorzugte Platt-

form für Client-Applikationen bleiben, aber so wie Oracle im Moment die Richtung vorgibt, gibt es Innovationen ausschließlich in JavaFX und nicht in Swing oder AWT. Schon jetzt sind die Charts oder die Webkomponente aus JavaFX großartig und für Swing-Entwickler eine Bereicherung.

Es ist nicht kompliziert, den Szenengraphen in Swing-Anwendungen einzubetten. Im Mittelpunkt der Integration steht die Swing-Komponente `JFXPanel`, die einen JavaFX-Szenengraphen aufnehmen kann. Damit ist eigentlich schon alles gesagt:

Listing 19.20 src/main/java/com/tutego/insel/javafx/JFXPanelDemo.java

```
package com.tutego.insel.javafx;

import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.embed.swing.JFXPanel;
import javafx.scene.*;
import javafx.scene.chart.PieChart;
import javax.swing.*;

public class JFXPanelDemo {
  public static void main( String[] args ) {
    JFrame f = new JFrame();
    final JFXPanel fxPanel = new JFXPanel();
    f.add( new JScrollPane( fxPanel ) );
    f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    f.setBounds( 100, 100, 600, 600 );
    f.setVisible( true );

    Platform.runLater( new Runnable() {
      @Override public void run() {
        PieChart chart = new PieChart(FXCollections.observableArrayList(
                         new PieChart.Data( "Java", 199 ),
                         new PieChart.Data( "Rest", 12 ) ) );

        fxPanel.setScene( new Scene( chart ) );
      }
    } );
  }
}
```

Das `JFXPanel` ist eine Swing-Komponente, die genauso leichtgewichtig ist wie alles andere von Swing auch. Das bedeutet: Das `JFXPanel` kann anders als AWT-Komponenten problemlos in Swing-Container wie die `JScrollPane` gesetzt werden, und auch die Transparenz funktio-

niert tadellos. Wenn also `JFXPanel` auf »nicht opak« gesetzt wird, scheint der Hintergrund durch. Das erlaubt eine perfekte Integration, die sich auch in der transparenten Ereignisbehandlung widerspiegelt. Ereignisse wie ein Klick auf eine JavaFX-Komponente kommen auch dort an und versanden nicht in Swing.

Ein wenig Mühe allerdings macht das Threading, denn Swing und JavaFX nutzen zwar beide Threads, doch unterschiedliche. Aus Swing ist bekannt, dass nach der Darstellung des Fensters Veränderungen an den Komponenten nur vom EDT (*Event Dispatching Thread*) erlaubt sind. Um Programmcode von einem Nicht-EDT- – wie dem Thread, der `main()` ausführt – im EDT-Thread zu platzieren, bietet die Swing-Bibliothek `SwingUtilities.invokeLater(Runnable)`. JavaFX bietet die vergleichbare Methode `Platform.runLater(Runnable)`, um Programmcode vom JavaFX-Thread ausführen zu lassen. Das ist in unserem Beispiel auch nötig, andernfalls wird `fxPanel.setScene()` aus dem `main`-Thread ausgeführt, was JavaFX mit einer `IllegalStateException` »Not on FX application thread; currentThread = main« bestraft. In JavaFX 8 sollten der JavaFX-Thread und der Event-Dispatch-Thread zusammenwachsen, doch ist das nicht standardmäßig aktiviert und muss erst über eine Property aktiviert werden.

19.15 Zum Weiterlesen

JavaFX 8 ist im Vergleich zu Swing oder SWT relativ neu und die Menge an gedruckter Literatur übersichtlich. Am besten ist es, den Dokumentationen der Webseite *https://docs.oracle.com/javase/8/javase-clienttechnologies.htm* zu folgen. Die News-Site *http://fxexperience.com/* veröffentlicht regelmäßig eine Link-Sammlung zu Beiträgen, die Neuerungen beschreiben.