

Diese Ergänzungen findet man analog auch in den für die primitiven Typen spezialisierten Stream-Klassen `IntStream`, `LongStream` sowie `DoubleStream` aus dem Package `java.util.stream`. Dort ist dann jeweils das Prädikat auf den korrespondierenden Typ angepasst, etwa `takeWhile(IntPredicate)`.

Beispiel für die Methoden `takeWhile()` und `dropWhile()`

Zur Demonstration der Methode `takeWhile()` wird ein unendlicher Stream von Ganzzahlen durch Aufruf von `iterate()` auf einem `IntStream` erzeugt, der mit der Zahl 1 beginnt. Für `dropWhile()` nutzen wir einen Stream mit dem vordefinierten Wertebereich von 7 bis 14, den wir durch Aufruf von `rangeClosed()` konstruieren. Zur Darstellung einer Besonderheit bei der Verarbeitung mit `dropWhile()` verwenden wir schließlich einen Stream mit vordefinierten Werten, der durch einen Aufruf von `of()` erzeugt wird:

```
public static void main(final String[] args)
{
    // Unendliche Wertefolge erzeugen und alle bis 10 abgreifen
    final IntStream stream1 = IntStream.iterate(1, n -> n + 1);
    System.out.println("takeWhile: " + stream1.takeWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(joining(", ")));

    // Wertebereich von 7 bis 14 erzeugen und alle kleiner 10 überspringen
    final IntStream stream2 = IntStream.rangeClosed(7, 14);
    System.out.println("dropWhile 1: " + stream2.dropWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(joining(", ")));

    // Demonstration von dropWhile() bei gemischter Wertefolge
    final IntStream stream3 = IntStream.of(7, 9, 11, 13, 15, 5, 3, 1);
    System.out.println("dropWhile 2: " + stream3.dropWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(joining(", ")));
}
```

Listing 3.16 Ausführbar als 'STREAMSDROPANDTAKEWILEEXAMPLE'

Durch den Aufruf von `mapToObj(Integer::toString)`¹⁰ konvertieren wir die Zahlen in einen String und bereiten mit `collect(joining(", "))` eine komma-separierte Darstellung auf. Dabei stammt die Methode `joining()` aus der Klasse `java.util.stream.Collectors` und wurde zur besseren Lesbarkeit statisch importiert. Dann ist es leicht nachvollziehbar, dass es zu den folgenden Ausgaben kommt, wenn man das Programm `STREAMSDROPANDTAKEWILEEXAMPLE` startet:

```
takeWhile: 1, 2, 3, 4, 5, 6, 7, 8, 9
dropWhile 1: 10, 11, 12, 13, 14
dropWhile 2: 11, 13, 15, 5, 3, 1
```

¹⁰Alternativ wäre natürlich auch der Einsatz des Lambda-Ausdrucks `num -> "" + num` möglich gewesen, der jedoch die Intention der Konvertierung in einen String nicht so deutlich macht wie die Methodenreferenz `Integer::toString`.

Die Ausgabe der Zahlen hinter dem Text `dropWhile 2` verdeutlicht, dass bei Aufrufen von `dropWhile()` nur zu Beginn die Einhaltung der Bedingung überprüft wird. Gilt diese einmal, so erfolgt danach keine weitere Prüfung mehr und es werden im Anschluss möglicherweise Elemente konsumiert, die gegen die angegebene Bedingung verstößen. Dieser Fall kann für `takeWhile()` so nicht auftreten, da dort die Verarbeitung sofort abgebrochen würde.

Beide Methoden in Kombination Auch in Kombination können die beiden Methoden sinnvoll eingesetzt werden. Das gilt etwa immer dann, wenn zunächst Informationen so lange aussortiert werden sollen, bis diese einem gewissen Gütekriterium oder Wert entsprechen, und dann im Anschluss so lange gelesen werden sollen, bis eine Abbruchbedingung erfüllt ist.

Als Beispiel werden die Informationen, die zwischen den Markierungen `<START>` und `<END>` liegen, aus einem `Stream<String>` extrahiert:

```
public static void main(final String[] args)
{
    Stream<String> words = Stream.of("ab", "bla", "<START>",
                                      "Hier", "steht", "der", "Text", "zwischen",
                                      "den", "Start-", "und", "Ende-Begrenzern",
                                      "<END>", "saas", "bla");

    Stream<String> content = words.dropWhile(word -> !word.equals("<START>"))
        .skip(1)
        .takeWhile(word -> !word.equals("<END>"));

    content.forEach(System.out::println);
}
```

Listing 3.17 Ausführbar als 'DROPANDTAKEWILECOMBINATIONEXAMPLE'

Das `skip(1)` ist nötig, um den Begrenzer `<START>` nicht mit in die Ergebnisliste aufzunehmen. Auf ähnliche Weise könnte man übrigens auch die Header- oder Body-Informationen eines HTML-Dokuments extrahieren.

Die Ausgaben des Programms `DROPANDTAKEWILECOMBINATIONEXAMPLE` zeigen sehr schön die Extraktion:

```
Hier
steht
der
Text
zwischen
den
Start-
und
Ende-Begrenzern
```

Weitere neue Methoden

Neben den Neuerungen in Form der Methoden `takeWhile()` und `dropWhile()` findet man für Streams folgende neue Methoden:

- `ofNullable(T)` – Liefert einen `Stream<T>` mit einem Element, sofern das übergebene Element ungleich `null` ist. Ansonsten wird ein leerer Stream erzeugt.
- `iterate(T, Predicate<? super T>, UnaryOperator<T>)` – Es wird ein `Stream<T>` mit dem als ersten Parameter übergebenen Startwert erzeugt. Die folgenden Werte werden durch den `java.util.function.UnaryOperator` berechnet. Im Gegensatz zu der bereits mit JDK 8 existierenden Methode `iterate(T, UnaryOperator<T>)` wird hierbei auch noch das übergebene `java.util.function.Predicate<T>` geprüft und die Erzeugung gestoppt, sobald dieses nicht mehr erfüllt ist.

Die erste Methode ist relativ selbsterklärend, daher möchte ich hier nur für die zweite Methode ein Beispiel analog zu dem für die Methode `takeWhile()` präsentieren. Dort haben wir zur Ausgabe der Zahlen von 1 bis 9 die Prüfung von $n \rightarrow n < 10$. ein `IntPredicate` verwendet. Stattdessen können wir diese Prüfung auch direkt im Aufruf von `iterate()` durchführen. Ausgehend vom Wert 1 erzeugen wir mit dem Lambda $n \rightarrow n + 1$ als `IntUnaryOperator` eine aufsteigende Zahlenfolge, deren Ende der Berechnung über die Bedingung $n \rightarrow n < 10$ gesteuert wird.

```
public static void main(final String[] args)
{
    // iterate() unterstützt seit JDK 9 eine Bedingung
    System.out.println("iterate with predicate");
    final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);
    System.out.println(stream.mapToInt(Integer::toString).
        collect(joining(", ")));
}
```

Listing 3.18 Ausführbar als 'STREAMSITERATEEXAMPLE'

Das Programm `STREAMSITERATEEXAMPLE` gibt erwartungsgemäß Folgendes aus:

```
iterate with predicate
1, 2, 3, 4, 5, 6, 7, 8, 9
```

Die Methode `iterate()` bietet damit eine sehr ähnliche Funktionalität wie eine klassische `for`-Schleife, allerdings mit dem Unterschied, dass die Aktionen im Stream lazy, also erst durch eine Terminal Operation (vgl. Abschnitt A.2) wie etwa das obige `collect()`, ausgeführt werden.

Fazit

Die mit JDK 9 ergänzten Methoden im Interface `Stream<T>` stellen eine sinnvolle Komplettierung des mit JDK 8 eingeführten Stream-APIs dar und runden dieses ab.

3.1.7 Erweiterungen in der Klasse LocalDate

Kommen wir nun zu einigen Erweiterungen in der Klasse `java.time.LocalDate` aus dem mit JDK 8 eingeführten Date and Time API. Dieses API war an sich schon recht komplett, sodass sich mit JDK 9 dort nur kleine Ergänzungen finden.

Die Klasse `LocalDate` besitzt nun eine überladene Methode `datesUntil()`. Diese erzeugt einen `Stream<LocalDate>` zwischen zwei `LocalDate`-Instanzen und erlaubt es, optional eine Schrittweite vorzugeben.

Im nachfolgenden Beispiel verwenden wir den Geburtstag des Autors und den Heiligabend desselben Jahres zur Demonstration von Berechnungen mit der Methode `datesUntil()`:

```
public static void main(final String[] args)
{
    final LocalDate myBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate christmas = LocalDate.of(1971, Month.DECEMBER, 24);

    System.out.println("Day-Stream");
    final Stream<LocalDate> daysUntil = birthday.datesUntil(christmas);
    daysUntil.skip(150).limit(8).forEach(System.out::println);

    System.out.println("\nMonth-Stream");
    final Stream<LocalDate> monthsUntil =
        birthday.datesUntil(christmas, Period.ofMonths(1));
    monthsUntil.limit(5).forEach(System.out::println);
}
```

Listing 3.19 Ausführbar als 'DATESUNTILEXAMPLE'

Führen wir das Programm `DATESUNTILEXAMPLE` aus, so werden startend vom 7. Februar zunächst 150 Tage in die Zukunft gesprungen, wodurch man am 7. Juli landet. Dann werden acht Werte eines Streams von Tagen ausgegeben. Zudem zeigen die zweiten Ausgaben die Vorgabe einer Schrittweite, hier Monate. Ausgehend vom 7. Februar werden zusätzlich zu diesem Datum noch vier Monate in die Zukunft aufgelistet:

```
Day-Stream
1971-07-07
1971-07-08
1971-07-09
1971-07-10
1971-07-11
1971-07-12
1971-07-13
1971-07-14

Month-Stream
1971-02-07
1971-03-07
1971-04-07
1971-05-07
1971-06-07
```

Neben dieser Neuerung findet man in der Klasse `LocalDate` folgende zwei erwähnenswerte Methoden:

- `toEpochSecond(LocalTime, ZoneOffset)` – Konvertiert eine `LocalDate`-Instanz in einen `long`, der den seit dem Referenzwert 1970-01-01T00:00:00Z vergangenen Sekunden entspricht. Weil ein `LocalDate` keine Zeitinformation besitzt, muss hier ein `LocalTime`- sowie ein `java.time.ZoneOffset`-Objekt übergeben werden.
- `ofInstant(Instant, ZoneId)` – Wandelt ein `java.time.Instant`-Objekt in ein `LocalDate` um, wobei eine Zeitzone in Form einer `java.time.ZoneId` benötigt wird.

Auch für diese beiden Methoden schreiben wir ein kleines Beispiel:

```
public static void main(final String[] args)
{
    final LocalDate someday = LocalDate.of(1971, 2, 7);
    final LocalTime time = LocalTime.of(2, 15);
    final Instant instant = Instant.ofEpochMilli(0);

    System.out.println("toEpochSecond: " + someday.toEpochSecond(time,
        ZoneOffset.ofHours(-10)));

    System.out.println("ofInstant: " + LocalDate.ofInstant(instant,
        ZoneId.of("Europe/Zurich")));
}
```

Listing 3.20 Ausführbar als 'LOCALDATECONVERSIONEXAMPLE'

Das Programm LOCALDATECONVERSIONEXAMPLE gibt Folgendes aus:

```
toEpochSecond: 34758900
ofInstant: 1970-01-01
```

3.1.8 Erweiterungen in der Klasse Arrays

Auch die Utility-Klasse `java.util.Arrays` wurde in JDK 9 erweitert. Das betrifft vor allem folgende Methoden und Funktionalitäten:

- `equals()` – Vergleicht zwei Arrays elementweise per `equals()` und bietet seit JDK 9 eine Bereichseinschränkung.
- `mismatch()` – Prüft, ob zwei Arrays sich unterscheiden, und liefert die Position der ersten Differenz. Durch die Angabe von Indizes können Teilebereiche von Arrays untersucht werden. Wird kein Unterschied gefunden, so ist die Rückgabe `-1`.
- `compare()` – Vergleicht zwei Arrays analog zu Komparatoren. Auch hier kann der Vergleich auf einem Teilebereich der Arrays erfolgen.

Zur Demonstration der drei Methoden definieren wir drei Texte, die wir als `byte[]` aufbereiten, die einige gleiche Werte enthalten, sich aber auch an einigen Stellen unter-

scheiden. Zunächst vergleichen wir die ersten sechs und dann sieben Werte der Arrays `a` und `b`. Danach nutzen wir die Arrays `b` und `c`, wobei wir die Teilbereiche variieren.

```

public static void main(final String[] args) throws Exception
{
    final byte[] a = "Hallo Welt".getBytes();
    final byte[] b = "Hallo JDK 9".getBytes();
    final byte[] c = "JDK 9 Release".getBytes();

    executeEquals(a, b, c);
    executeMismatch(a, b, c);
    executeCompare(a, b, c);
}

private static void executeEquals(final byte[] a, final byte[] b,
                                 final byte[] c) throws Exception
{
    perform("\nequals", a, 0, 6, b, 0, 6,
           () -> Arrays.equals(a, 0, 6, b, 0, 6));

    perform("equals", a, 0, 7, b, 0, 7,
           () -> Arrays.equals(a, 0, 7, b, 0, 7));

    perform("equals", c, 0, 5, b, 6, 11,
           () -> Arrays.equals(c, 0, 5, b, 6, 11));
}

private static void executeMismatch(final byte[] a, final byte[] b,
                                    final byte[] c) throws Exception
{
    perform("\nmismatch", a, 0, 6, b, 0, 6,
           () -> Arrays.mismatch(a, 0, 6, b, 0, 6));

    perform("mismatch", a, 0, 7, b, 0, 7,
           () -> Arrays.mismatch(a, 0, 7, b, 0, 7));
}

private static void executeCompare(final byte[] a, final byte[] b,
                                   final byte[] c) throws Exception
{
    perform("\ncompare", a, 0, 6, b, 0, 6,
           () -> Arrays.compare(a, 0, 6, b, 0, 6));

    perform("compare", a, 0, 7, b, 0, 7,
           () -> Arrays.compare(a, 0, 7, b, 0, 7));

    perform("compare", b, 0, 5, c, 0, 5,
           () -> Arrays.compare(b, 0, 5, c, 0, 5));
}

private static void perform(final String info, final byte[] array1,
                           final int start1, final int end1, final byte[] array2, final int start2,
                           final int end2, final Callable<Object> action) throws Exception
{
    final String value1 = new String(array1, start1, end1-start1);
    final String value2 = new String(array2, start2, end2-start2);
    System.out.print(info + "(" + value1 + ", " + value2 + ") => ");
    System.out.println(action.call());
}

```

Listing 3.21 Ausführbar als 'ARRAYSEXAMPLE'

Der Start der Programms `ARRAYSEXAMPLE` liefert folgende Konsolenausgaben:

```
equals('Hallo ', 'Hallo ') => true
equals('Hallo W', 'Hallo J') => false
equals('JDK 9', 'JDK 9') => true

mismatch('Hallo ', 'Hallo ') => -1
mismatch('Hallo W', 'Hallo J') => 6

compare('Hallo ', 'Hallo ') => 0
compare('Hallo W', 'Hallo J') => 13
compare('Hallo', 'JDK 9') => -2
```

Die Arrays `a` und `b` sind in den ersten sechs Positionen gleich. Erweitert man den Vergleichsbereich, so kommt es zu einem Unterschied und der Rückgabe von `false`. Der Text `JDK 9` findet sich in Array `b` und versetzt in Array `c`. Deshalb ist der letzte `equals()`-Vergleich erfolgreich.

Bei `mismatch()` wird für die ersten sechs Werte von `a` und `b` kein Unterschied erkannt und somit `-1` zurückgegeben. Danach wird in einem größeren Bereich nach einem Unterschied gesucht und an letzter Position gefunden. Schließlich liefert `compare()` für den ersten Wertebereich konsistent zu `equals()` und `mismatch()` den Wert `0`. Analog zu Komparatoren wird dann für die anderen beiden Wertebereiche einmal größer (`13`) und kleiner (`-2`) zurückgeliefert. Dabei entsprechen die absoluten Werte jeweils dem Abstand der Buchstaben im Alphabet und das Vorzeichen besagt größer oder kleiner. Konkret liegt das `W` aus Welt 13 Buchstaben hinter dem `J` aus `JDK`. Die `-2` ergibt sich aus der Differenz zwischen `H` und `J`.

3.1.9 Erweiterungen in der Klasse Objects

Die Utility-Klasse `java.util.Objects` erlaubt es, durch die Methode `requireNonNull()` eine elegante Prüfung von Preconditions bezüglich `null` durchzuführen.

Mit JDK 9 wird das Ganze noch handlicher: Es ist nun analog zu einigen Methoden aus `Optional<T>` möglich, mit `requireNonNullElse()` bzw. `requireNonNullElseGet()` einen Alternativwert im Falle eines `null`-Werts bereitzustellen bzw. durch einen `Supplier<T>` zu berechnen.

Als Beispiel dient die Methode `generateMsg()`, die zwei `String`-Parameter besitzt und daraus eine Nachricht erzeugt. Dabei erfolgt ein Null-Handling unter Einsatz der genannten Methoden.

```
private static String generateMsg(final String msg, final String param)
{
    final String message = Objects.requireNonNull(msg, "Default-Msg");
    final String parameter =
        Objects.requireNonNullElseGet(param, () -> "No Param");

    return message + " : " + parameter;
}
```

In der `main()`-Methode übergeben wir zwei Mal den Wert `null`, um das zu prüfen:

```
public static void main(final String[] args)
{
    System.out.println(generateMsg(null, null));
}
```

Listing 3.22 Ausführbar als 'OBJECTSNONNULLEXAMPLE'

Startet man das Programm `OBJECTSNONNULLEXAMPLE`, so wird die Methode `generateMsg()` mit zwei `null`-Werten aufgerufen. In der Methode selbst werden diese mithilfe der zuvor genannten Methoden entsprechend behandelt und direkt in den `String Default-Msg` und im zweiten Fall als `Supplier<String>` in den Wert `No Param` transformiert. Somit kommt es zu folgender Ausgabe:

```
Default-Msg : No Param
```

3.1.10 Erweiterungen in der Klasse `CompletableFuture<T>`

Die Klasse `java.util.concurrent.CompletableFuture<T>` wurde in Java 8 eingeführt und bietet eine Erleichterung bei der Programmierung asynchroner Abläufe. Diese lassen sich bezüglich Lesbarkeit und Komplexität deutlich besser als Lösungen mit Threads gestalten.

Zum Einstieg stelle ich an einem Beispiel die Möglichkeiten mit JDK 8 vor. Danach gehe ich auf die Neuerungen in JDK 9 ein. Eine umfangreichere Einführung zur Klasse `CompletableFuture<T>` finden Sie in Anhang A.

Die Klasse `CompletableFuture` mit JDK 8

Mithilfe der Klasse `CompletableFuture<T>` lassen sich asynchrone Abläufe und deren Verarbeitungsschritte elegant formulieren. Dabei kann man auf Ereignisse reagieren, die in Zukunft auftreten werden.

Das nachfolgende Beispiel zeigt, wie man Verarbeitungsketten aufbauen kann, etwa um Aktionen parallel zum Aufrufer oder nach Abschluss der Abarbeitung auszuführen. Dafür nutzen wir folgende Methoden:

1. `supplyAsync()` – Als Ausgangspunkt einer Verarbeitung wird mit `supplyAsync()` asynchron eine Aktion abgearbeitet.
2. `thenAccept()` – Als Nächstes wird mit `thenAccept()` eine Aktion nach Abschluss der potenziell aufwendigen Verarbeitung ausgeführt.
3. `exceptionally()` – Schließlich lässt sich durch `exceptionally()` elegant auf das Auftreten von Exceptions reagieren.

Mit diesem Wissen implementieren wir verschiedene asynchrone Abläufe als Beispiel zu `CompletableFuture<T>` in der Methode `perform()` wie folgt:

```

public static void main(final String[] args) throws InterruptedException,
    ExecutionException
{
    new CompletableFutureExample().perform();
}

public void perform() throws InterruptedException, ExecutionException
{
    // 1) supplyAsync(): eine einzelne Aktion parallel zum Aufrufer ausführen
    final CompletableFuture<String> supplyAsync =
        CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5));

    // 2) thenAccept(): Aktion nach Abschluss der Ausführung
    supplyAsync.thenAccept(this::notifySubscribers);

    // 3) exceptionally(): Exception-Mapping
    CompletableFuture.supplyAsync(this::failingMsg)
        .exceptionally(ex -> "FAILED")
        .thenAccept(this::notifySubscribers);

    System.out.println(Thread.currentThread() + " perform()");
}

public String longRunningCreateMsg(final int durationInSecs)
{
    System.out.println(Thread.currentThread() + " >>> longRunningCreateMsg");

    sleepInSeconds(durationInSecs);
    System.out.println(Thread.currentThread() + " <<< longRunningCreateMsg");

    return "longRunningCreateMsg";
}

public String getCurrentThread()
{
    return Thread.currentThread().getName();
}

public void notifySubscribers(final String msg)
{
    System.out.println(Thread.currentThread() + " notifySubscribers: " + msg);
}

public String failingMsg()
{
    throw new IllegalStateException("ISE");
}

private void sleepInSeconds(final int durationInSeconds)
{
    try
    {
        TimeUnit.SECONDS.sleep(durationInSeconds);
    }
    catch (InterruptedException e)
    { /* not possible here */ }
}

```

Listing 3.23 Ausführbar als 'COMPLETABLEFUTUREEXAMPLE'

Zunächst wird eine Nachricht per `longRunningCreateMsg()` mit einer Abarbeitungszeit von 5 Sekunden erzeugt und mit `supplyAsync()` asynchron (parallel zum aufrufenden Thread, in diesem Fall dem `main()`-Thread) ausgeführt. Als darauffolgende Aktion wird `notifySubscribers()` durch Aufruf von `thenAccept()` registriert und nach Abschluss der ersten Aktion abgearbeitet. Zum Abschluss sehen wir ein Beispiel für das Exception Handling.

Startet man das Programm `COMPLETABLEFUTUREEXAMPLE`, so kommt es zu den nachfolgenden Ausgaben, die die beschriebenen Abläufe zeigen:

```
ForkJoinPool.commonPool-worker-9 >>> longRunningCreateMsg
main notifySubscribers: FAILED
main perform()
```

Mit diesem Beispiel und eventuell einem ergänzenden Blick in Anhang A sollten Sie ein erstes Verständnis für die Arbeitsweise der Klasse `CompletableFuture<T>` aufgebaut haben. Kommen wir nun zu den Neuerungen in JDK 9.

Die Klasse `CompletableFuture` mit JDK 9

In der Klasse `CompletableFuture<T>` wurden in Java 9 diverse Methoden ergänzt, unter anderem folgende:

- `completeAsync(Supplier<? extends T>)` und `completeAsync(Supplier<? extends T>, Executor)` – Erfüllt das `CompletableFuture<T>` mit dem vom übergebenen `Supplier<T>` gelieferten Ergebnis. Dieses wird asynchron von einem Task berechnet, der entweder vom Default Executor oder dem übergebenen ausgeführt wird.
- `orTimeout(long, TimeUnit)` – Sofern das `CompletableFuture<T>` nicht zuvor erfolgreich ausgeführt wurde, wird es mit einer `java.util.concurrent.TimeoutException` beendet, wenn die angegebene Time-out-Zeit erreicht ist.
- `completeOnTimeout(T, long, TimeUnit)` – Das `CompletableFuture<T>` wird mit dem übergebenen Wert erfüllt, falls die Berechnungen nicht innerhalb der gegebenen Time-out-Zeit zum Ergebnis führen.
- `failedFuture(Throwable)` – Gibt ein `CompletableFuture<T>` zurück, das bereits durch die übergebene Exception erfüllt wurde. Dies kann man zum Signalisieren von Fehlerzuständen während einer asynchronen Berechnung nutzen.

Für die aufgelisteten Methoden wollen wir ein Beispiel erstellen. Dabei kommen die gleichen Hilfsmethoden wie zuvor zum Einsatz und werden nicht nochmals abgebildet:

```
public static void main(final String[] args) throws ExecutionException
{
    CompletableFutureJdk9Example example = new CompletableFutureJdk9Example();
    example.perform();
}
```

```

public void perform() throws ExecutionException
{
    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .completeAsync(() -> "COMPLETE")
        .thenAccept(this::notifySubscribers);

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .orTimeout(3, TimeUnit.SECONDS)
        .exceptionally(ex -> "exception occurred: " + ex)
        .thenAccept(this::notifySubscribers);

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .completeOnTimeout("TIMEOUT-FALLBACK", 2, TimeUnit.SECONDS)
        .thenAccept(this::notifySubscribers);

    CompletableFuture.failedFuture(new IllegalStateException())
        .exceptionally(ex -> {
            System.out.println("ALWAYS FAILING");
            return -1;
        });
}

sleepInSeconds(10); // Auf die Terminierung des CompletableFutures warten
}

```

Listing 3.24 Ausführbar als 'COMPLETABLEFUTUREJDK9EXAMPLE'

Zunächst wird asynchron eine rund fünf Sekunden dauernde Methode `longRunningCreateMsg()` ausgeführt. Deren Verarbeitung kann man durch `completeAsync()` vorzeitig als beendet markieren. Hier wird als Ergebnis `COMPLETE`, wie schon im Beispiel zuvor, mithilfe der Aufrufe von `thenAccept()` und `notifySubscribers()` ausgegeben. Die zweite Variante führt wieder die lang dauernde Methode asynchron aus, diesmal kommt es aber nach drei Sekunden zu einem Time-out. Das führt zur Ausgabe von `exception occurred: java.util.concurrent.TimeoutException`. Die dritte Variante zeigt, wie man bei einem Time-out einen Wert zurückliefern kann. Dies geschieht im Falle eines Time-outs, für den hier zwei Sekunden gewählt wurden. Verbleibt noch die Methode `failedFuture()`. Hiermit lässt sich das `CompletableFuture<T>` mit einem Fehler kompletieren. In diesem Fall wird dadurch `ALWAYS FAILING` ausgegeben.

Startet man das Programm `COMPLETABLEFUTUREJDK9EXAMPLE`, so kommt es zu folgenden Ausgaben, wobei die Reihenfolge durch die Nebenläufigkeit abweichen kann:

```

ForkJoinPool.commonPool-worker-9 >>> longRunningCreateMsg
main notifySubscribers: COMPLETE
ForkJoinPool.commonPool-worker-2 >>> longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 >>> longRunningCreateMsg
ALWAYS FAILING
CompletableFutureDelayScheduler notifySubscribers: TIMEOUT-FALLBACK
CompletableFutureDelayScheduler notifySubscribers: exception occurred: java.util
    .concurrent.TimeoutException
ForkJoinPool.commonPool-worker-9 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-2 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 <<< longRunningCreateMsg

```