PROGRAMMER'S CHOICE

PEARSON Education

Berthold Daum

Java 6

Programmieren mit der
Java Standard Edition

ADDISON-WESLEY

4 Kernklassen der JRE

Während die Sprache Java relativ schlank ist, hat das *Java Runtime Environment* (JRE), also die Java-Laufzeitumgebung, inzwischen eine solche ausufernde Mächtigkeit erreicht, dass es kaum noch von einzelnen Programmierern komplett zu beherrschen ist. Spezialisierung ist also angesagt. Der eigentliche Kern des JRE, das Archiv `rt.jar`, umfasst vier Gruppen von Packages:

- Die im Namen mit `com.sun.` bzw. `sun.` beginnenden Packages sind interner Natur und sollten in der Regel von Anwendungsprogrammierern nicht benutzt werden.
- Die mit `java.` beginnenden Packages enthalten Klassen, die wesentliche Funktionen der Sprache Java unterstützen. Dazu gehört vor allem das Standard-Package `java.lang` (siehe Abschnitt 4.1). Das Package `java.util` (siehe Abschnitt 4.2) definiert weitere häufig benutzte Typen. Weitere Packages bzw. Package-Gruppen sind: `java.applet` (Applet-Infrastruktur), `java.awt` (Basis-GUI), `java.beans` (komponentenbasiertes Programmieren), `java.io` und `java.nio` (Ein- und Ausgabefunktionen, siehe Abschnitt 10.1), `java.net` (Netzwerke), `java.rmi` (entfernter Aufruf von Methoden), `java.security` (Sicherheitsfunktionen, siehe Abschnitt 5.6), `java.sql` (Unterstützung relationaler Datenbanken, siehe Abschnitt 10.4.2) und `java.text` (Textverarbeitung).

- Die mit `javax.` beginnenden Packages enthalten Klassen, welche die Funktionalität von Java für spezifische Aufgabenfelder erweitern. Dazu gehören insbesondere: `javax.accessibility` (barrierefreie Software), `javax.annotation` (vordefinierte Annotationstypen, siehe Abschnitt 5.4), `javax.imageio` (IO-Unterstützung für verschiedene Bildformate), `javax.jws` (Webservices, siehe Kapitel 11), `javax.lang.model` (Metamodell der Sprache), `javax.management` (Überwachung, Leistungsmessung und Verwaltung von Anwendungen), `javax.naming` (Unterstützung für Namensdienste), `javax.print` (Druckerunterstützung), `javax.rmi` (Erweiterungen für `java.rmi`), `javax.script` (Unterstützung für eingebettete Skripte), `javax.security` (zusätzliche Sicherheitsfunktionen, insbesondere Authentifizierung), `javax.sound` (Audio), `javax.sql` (zusätzliche Unterstützung für relationale Datenbanken), `javax.swing` (Swing-basierte GUIs), `javax.tools` (APIs für Programmierwerkzeuge), `javax.transaction` (Unterstützung für Transaktionssysteme, z. B. Datenbanken) und `javax.xml` (XML-Verarbeitung, siehe Abschnitt 10.3).
- Schließlich sind noch einige mit `org.` beginnende Packages vorhanden, die zwar nicht von Sun erstellt wurden, jedoch mit ausgeliefert werden. Dazu gehören insbesondere die CORBA-Unterstützung von OMG und die APIs des W3C für die XML-Verarbeitung.

Zusätzlich zu `rt.jar` enthält das JRE noch eine Reihe weiterer JARS, insbesondere für die sichere Datenübertragung (`jssse.jar`), die Kryptografie (`jce.jar` und `sunjce_ provider.jar`), Versiegelung und Verschlüsselung (`sunmscapi.jar`), Sicherheit (`sunpkcs11.jar`), Namensdienste (`dnsns.jar`), Zeichensätze (`charsets.jar`) und Internationalisierung (`localedata.jar`).

In den folgenden Abschnitten möchte ich einige wichtige Klassen aus den Packages `java.lang` und `java.util` vorstellen.

4.1 Das Package java.lang

Während Typen aus anderen Packages nur genutzt werden können, wenn sie entweder voll (also mit Package-Namen) qualifiziert oder aber importiert werden, werden die Typen des Package `java.lang` automatisch in jedes Java-Programm importiert. Dieses Package enthält die Definition der Basistypen, ohne die kein Java-Programm ablaufen kann, so z. B. die Definition der Klasse `Object` (siehe Abschnitt 2.2.3), die in Java an der Spitze der Vererbungshierarchie steht. Jede andere Klasse ist mittel- oder unmittelbar ein Abkömmling von `Object` und auf deren Methoden angewiesen.

Auch die Klasse `String` ist hier definiert (siehe Abschnitt 4.1.1). Für die primitiven Datentypen gibt es in `java.lang` die jeweils objektorientierte Variante: `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float` und `Double`. Alle diese Klassen enthalten Methoden, um den zugehörigen primitiven Datentyp in eine Zeichenkette umzuwandeln

(`toString()`) und umgekehrt eine Zeichenkettenrepräsentation wieder in den Datenwert zu überführen (`parseBoolean()`, `parseByte()`, `parseShort()` usw.)

Die Instanzen dieser Typen sind übrigens unveränderlich. Sie erhalten über den Konstruktor ihren Wert, der dann nicht mehr verändert werden kann. Das Gleiche gilt auch für `String`, nur dass dort kein Konstruktor nötig ist: Statt `new String("hello")` schreibt man einfache `"hello"`. Für den (primitiven) Ergebnistyp `void` (kein Ergebnis) gibt es übrigens auch ein Gegenüber: die Klasse `Void`. Benötigt wird diese Klasse z. B. bei reflexiven Anwendungen (siehe Abschnitt 5.5), wenn z. B. der Ergebnistyp einer Methode als Objekt dargestellt werden soll. Für reflexive Anwendungen ist außerdem die Metaklasse `Class` vorhanden. Weitere Typen für die Reflektion findet man im Package `java.lang.reflect`.

Die Klasse `Math` stellt die wichtigsten mathematischen Funktion wie `sin()`, `cos()`, `tan()`, `atan()`, `pow()`, `log()` usw. als statische Methoden zur Verfügung. Spezielle mathematische Typen wie `BigInteger` oder `BigDecimal` findet man dagegen im Package `java.math`.

Eine weitere große Gruppe sind die Ausnahmeereignisse – die Klassen `Throwable` und `Exception` und ihre Unterklassen (siehe Abschnitt 3.8.6).

Die Klassen `System` und `Runtime` gestatten schließlich den Zugriff auf die Ablaufumgebung (siehe Abschnitt 4.1.2). Die Klasse `ClassLoader` ist für das Laden von Java-Klassen verantwortlich. Wir werden später noch in einem Beispiel (siehe Abschnitt 5.2) sehen, wie andere Klassenlader konfiguriert und eingesetzt werden können. Die Klasse `Thread` schließlich ist der Grundbaustein für die nebenläufige Programmierung (siehe Abschnitt 5.1.1).

Außerdem finden wir in diesem Package noch einige grundlegende Interfaces wie z. B. das Interface `Runnable`, mit dessen Hilfe Funktionalität als Objekt eingepackt und so als Parameter übergeben werden kann (Java gestattet es nicht, Methoden als Parameter zu übergeben), das Interface `Clonable`, das von allen Klassen implementiert werden muss, deren Instanzen geklont werden sollen, und das Interface `Comparable`, das von allen Klassen implementiert werden muss, deren Instanzen auf größer/kleiner/gleich vergleichbar sein sollen. Zu den Interfaces gehören auch Standard-*Annotationen* wie z. B. `Override` (siehe Abschnitt 5.4).

4.1.1 Zeichen und Zeichenketten

Zeichen

Zeichen werden in Java – wie schon erwähnt – mithilfe des primitiven Datentyps `char` dargestellt. Man sollte diesen Datentyp nicht mit dem Datentyp `byte` verwechseln. Während `byte` lediglich einen Ganzzahlwert zwischen 0 und 255 repräsentiert, steht `char` für ein ganz bestimmtes Zeichen in einem gegebenen Zeichensatz. Da Java *Unicode* unterstützt, können Zeichen mehr als ein Byte benötigen.

Sollen Zeichen als volle Objekte dargestellt werden, so steht dafür die Klasse `Character` im Package `java.lang` zur Verfügung. Diese Klasse enthält zusätzlich eine Reihe nützlicher statischer Funktionen für die Zeichenverarbeitung und -klassifikation. So können mit den Methoden `Character.toLowerCase()` bzw. `Character.toLowerCase()` einzelne Zeichen in Klein- bzw. Großschreibung umgewandelt werden. Mit Methoden wie `isDigit()`, `isLetter()`, `isLowerCase()`, `isSpaceChar()` und `isWhiteSpace()` kann die Zugehörigkeit von Zeichen zu Zeichengruppen geprüft werden.

Zeichenketten

Auf niedriger Ebene lassen sich Zeichenketten durchaus als `char`-Arrays darstellen und mit den üblichen Arrayfunktionen verarbeiten. Man verschenkt dabei jedoch eine Menge Komfort. Die Klasse `String` benutzt `char`-Arrays intern als Speicherstruktur für Zeichenketten, stellt jedoch zusätzlich ein reiches Arsenal von Methoden für die Verarbeitung von Zeichenketten zur Verfügung. Insbesondere unterstützten die `String`-Konstruktoren auch die Umwandlung von `byte`-Arrays in Zeichenketten. Dabei wird entweder das Codesystem der Ablaufplattform angenommen oder es kann ein Codesystem explizit spezifiziert werden.

Mit den folgenden Anweisungen werden mithilfe von `String`-Methoden überflüssige Nullen und Dezimalpunkte am Ende einer Zeichenkette entfernt – bis auf die letzte einzelstehende Null:

```
while (displayString.endsWith("0"))
  displayString = displayString.substring(0, displayString.length() - 1);
if (displayString.endsWith("."))
  displayString = displayString.substring(0, displayString.length() - 1);
if (displayString.length() == 0)
  displayString = "0";
```

Solange die Zeichenkette `displayString` mit »0« aufhört, werden mit der Methode `substring()` alle Zeichen bis auf das letzte als neues `String`-Objekt entnommen und der Variablen `displayString` zugewiesen. Anschließend wird auch noch ein Dezimalpunkt am Ende entfernt. Bleibt nur die leere Zeichenkette übrig, wird stattdessen die Konstante »0« verwendet.

Hier sieht man aber auch eines der Probleme der Zeichenkettenverarbeitung mithilfe der Klasse `String`: Sie ist teuer. Bei jedem Schleifendurchlauf wird ein neues `String`-Objekt erzeugt und Objekterzeugung ist aufwändig. Hat man viele Nullen zu entfernen, sollte man sich ein anderes Vorgehen überlegen. Im folgenden Code wird zunächst von hinten mithilfe der Methode `charAt()` nach dem ersten Zeichen ungleich Null gesucht. Dann wird höchstens einmal mit `substring()` ein neues `String`-Objekt erzeugt.

```
        int n = displayString.length();
    loop: for (int i = n-1; i >= 0; i--) {
            char c = displayString.charAt(i);
            switch (c) {
            case '0':
              n = i;
              break;
            case '.':
              n = i;
              break loop;
            default:
              break loop;
            }
          }
          displayString = (n == 0) ? "0" : displayString.substring(0,n);
```

Kommt es bei langen Zeichenketten auf das letzte Quäntchen Performance an, so lässt sich der Overhead in den Methoden `charAt()` und `substring()` vermeiden. Dazu wandelt man das `String`-Objekt mit `toCharArray()` in ein `char`-Array um und arbeitet dann ausschließlich mit Array-Operationen. Allerdings entsteht dabei ein zusätzliches Objekt (das `char`-Array) und es wird doppelt so viel Speicherplatz benötigt:

```
        char[] chars = displayString.toCharArray();
        int n = chars.length;
    loop: for (int i = n-1; i >= 0; i--) {
            switch (chars[i]) {
            case '0':
              n = i;
              break;
            case '.':
              n = i;
              break loop;
            default:
              break loop;
            }
          }
          displayString = (n == 0) ? "0" : new String(chars, 0, n);
```

Neben diesen relativ einfachen String-Methoden wurde mit Java 1.4 eine Gruppe von Methoden (`replaceAll()` und `replaceFirst()`) eingeführt, die mithilfe von regulären Ausdrücken in der Lage sind, komplexe Umformungen von Zeichenketten vorzunehmen. Da diese Umformungen intern mithilfe der Klasse `Pattern` aus dem Package `java.util` durchgeführt werden, werden wir die Diskussion dieser Funktionen auf Abschnitt 4.2.5 verschieben.

Zeichenketten verketten

Mehrere Zeichenketten lassen sich leicht mit dem Operator »+« verketten, z. B. so:

```
String message = "This is pass " + i + " through the loop";
```

Dabei werden Objekte und auch primitive Werte automatisch in Zeichenketten umgewandelt, z. B. hier die Variable `i`. Allerdings sollte man sich auch hier darüber klar werden, wie der Compiler den String-Ausdruck auflöst. Der obige Ausdruck ist äquivalent mit:

```
StringBuilder buffer = new StringBuilder(("This is pass ");
String message = buffer.append(i).append(" through the loop").toString();
```

Dabei werden zwei Objekte erzeugt: zunächst der Arbeitspuffer `buffer` und dann als Ergebnis in der Methode `toString()` ein `String`-Objekt. Im obigen Beispiel lässt sich das kaum besser bewerkstelligen. Führt man jedoch komplexe Zeichenkettenoperationen durch, so sollte man möglichst die Zwischenergebnisse nicht als `String`-Objekte ablegen, sondern lieber die gesamte Verarbeitung in einem `StringBuilder`-Arbeitspuffer durchführen. So werden beispielsweise bei

```
String message = "This is pass " + i;
message += " through the loop";
```

bereits vier Objekte (zwei `StringBuilder`- und zwei `String`-Objekte) erzeugt. Bei

```
StringBuilder buffer = new StringBuilder(("This is pass ");
buffer.append(i);
buffer.append(" through the loop");
String message = buffer.toString();
```

dagegen bleibt es bei zwei Objekten.

Alte Java-Hasen mögen jetzt fragen, warum ich hier statt der Klasse `StringBuffer` die Klasse `StringBuilder` verwende. `StringBuilder` wurde mit Java 6 eingeführt und besitzt das gleiche API wie `StringBuffer`. Der einzige Unterschied ist, dass `StringBuffer` synchronisiert arbeitet, d. h. bei nebenläufigen Threads (siehe Abschnitt 5.1) eingesetzt werden kann, `StringBuilder` dagegen nicht. In den meisten Fällen (wie in diesem Beispiel) ist die relativ teure Synchronisierung aber überflüssig, so dass die performantere `StringBuilder`-Lösung völlig genügt.

Der +-Operator wird gern genutzt, um lange Zeichenketten auf mehrere Zeilen umzubrechen. Dabei muss man sich keine Sorgen um Performance-Einbußen machen. Da ein solcher Ausdruck nur aus Konstanten besteht, können Compiler diesen Ausdruck bereits bei der Kompilierung ausführen und durch das Ergebnis ersetzen.

Der String-Pool

Die Klasse `String` verwaltet intern einen Bereich, in dem alle als Konstanten oder Literale definierten Zeichenketten gespeichert werden. Dabei wird sichergestellt, dass der gleiche Wert nicht doppelt eingetragen wird. Das bedeutet, dass auch dann, wenn Sie die gleiche Zeichenkette an mehreren Stellen Ihres Programms explizit definieren, nur eine einzige Instanz dieser Zeichenkette existiert. Das spart Speicherplatz, hat aber auch Auswirkungen auf die Identität: Während normalerweise `String`-Objekte mit gleichem Inhalt nicht identisch sind, ist das für `String`-Literale durchaus der Fall:

```
String hello1 = "hello";
String hello2 = "hello";
```

Hier gilt also nicht nur `hello1.equals(hello2)`, sondern auch `hello1 == hello2`.

Da die Überprüfung auf Identität viel schneller ist als der zeichenweise Vergleich, liegt die Frage auf der Hand, ob man nicht auch andere Strings in den String-Pool eintragen kann. Das ist in der Tat möglich – die Klasse `String` stellt dazu die Methode `intern()` bereit. In unserem Beispiel in Abschnitt 3.11 hatten wir in der Methode `process()` der Klasse `ProcessingUnit` diese Technik bereits angewandt. Mit

```
command = command.intern();
```

hatten wir das Kommando in den internen String-Pool eingetragen und dann in der Folge die Vergleiche mit == statt mit `equals()` ausgeführt.

Die Verwendung dieser Technik ist freilich nicht ganz ungefährlich. In öffentlichen Schnittstellen sollte man niemals voraussetzen, dass Methodenparameter bereits in internalisierter Form übergeben werden. Die Gefahr wäre sonst viel zu groß, dass man das bei Methodenaufrufen übersieht – und dann funktioniert kein Vergleich mehr.

Außerdem sollte man in der Regel darauf verzichten, Zeichenketten, die aus einem unbeschränkten Wertevorrat kommen (z. B. Zeichenketten aus Benutzereingaben), zu internalisieren. Sonst besteht nämlich die Gefahr, dass sich der interne String-Pool immer weiter aufbläht, je länger die Anwendung läuft.

4.1.2 Systemanbindung

In fast jedem Java-Programm benötigt man den Zugriff auf die Systemumgebung. So will man z. B. Ein- und Ausgaben auf der Systemkonsole durchführen, will wissen, welche Konfigurationsparameter der *Java Virtual Machine* beim Start der Anwendung mitgegeben wurden, oder einen neuen Prozess anstarten. Für derartige Zwecke stellt das Package `java.lang` die Klassen `System` und `Runtime` bereit.

So steht in der Klasse `System` in den öffentlichen Feldern `in`, `out` und `err` der Standardeingabestrom zur Verfügung und die Ausgabeströme für normale Ausgaben und für Fehlermeldungen. Wir hatten ja bereits im ersten Kapitel gesehen, wie mit

```
System.out.println("Hello World!");
```

Ausgaben auf der Konsole erzeugt werden konnten. Diese Standardströme können bei Bedarf auch mit den entsprechenden `set`*XXX*`()`-Methoden geändert werden.

Mit der Methode `getProperty()` lassen sich Konfigurationsparameter abfragen, die beim Start der JVM entweder automatisch gesetzt oder mit dem Kommandozeilenparameter `-D` spezifiziert wurden. Über die Methode `getenv()` erhält man sogar die im Betriebssystem gesetzten Umgebungsvariablen. Mit `setProperty()` können Konfigurationsparameter sogar neu gesetzt bzw. abgeändert werden – bei Umgebungsvariablen ist ein Setzen allerdings nicht möglich. Die vorhandenen Property-Werte kann man sich mit

```
System.getProperties.list(System.out)
```

auf die Konsole ausgeben lassen. Allgemeine Informationen über die von `System` verwendete Klasse `Properties` finden Sie in Abschnitt 4.2.3.

Weitere wichtige Methoden der Klasse `System` sind die Methoden `currentTimeMillis()` zum Ermitteln des aktuellen Zeitstempels (siehe auch Abschnitt 4.2.4) und `arraycopy()` zum Kopieren von Arraybereichen. Da Millisekunden zum Messen von Ausführungszeiten inzwischen zu grobkörnig sind, gibt es seit Java 5 auch noch die Methode `nanoTime()`, die einen Zeitstempel in der Auflösung von Nanosekunden liefert (freilich nur, wenn die Ablaufplattform dies unterstützt). Die Methode `arraycopy()` hatten wir bereits in Abschnitt 3.2.2 in der Anwendung gesehen.

Auskunft über den maximal möglichen, den aktuell verfügbaren und den freien Speicherplatz der *Java Virtual Machine* erhält man über die Klasse `Runtime` mithilfe der Methoden `maxMemory()`, `totalMemory()` und `freeMemory()`. Mit der Methode `gc()` kann der *Garbage Collector* explizit angestartet werden. Alle diese Methoden sind (anders als bei `System`) Instanzenmethoden. Mit der statischen Methode `getRuntime()` muss man sich deshalb zunächst eine `Runtime`-Instanz besorgen.

Außerdem ermöglicht die Klasse `Runtime` das Ausführen von Betriebssystemkommandos mithilfe der Methode `exec()`. Diese Methode wartet nicht auf die Beendigung des neuen Prozesses, liefert aber eine `Process`-Instanz zurück. Mittels deren `waitFor()`-Methode kann auf die Beendigung des Prozesses gewartet werden. Zusätzlich ermöglicht die `Process`-Instanz den Zugriff auf Eingabe-, Ausgabe- und Fehlerströme des neuen Prozesses sowie auf dessen Rückgabecode. So können eventuelle Ausgaben des Betriebssystems vom Java-Programm empfangen und ausgewertet werden.

Wichtig ist außerdem die `Runtime`-Methode `addShutdownHook()`. Über diese Methode kann eine `Thread`-Instanz registriert werden, die beim ordnungsgemäßen Beenden der *Java Virtual Machine* ausgeführt wird. So wird in unserem *jDashboard*-Beispielprogramm bereits in der `main()`-Methode ein solcher Shutdown-Hook gesetzt, um später beim Herunterfahren des Systems unter anderem den aktuellen Zustand des Dashboard in eine XML-Datei zu sichern. Beim Neustart kann dann mithilfe dieser Datei der vorige Zustand wiederhergestellt werden:

```
Thread shutdownHook = new Thread(new Runnable() {
  public void run() {
    if (dashboard.opened)
      dashboard.saveXmlFile(manifest);
    dashboard.stopWidgets();
    disposeGlobalHotkeySupport();
  }
});
Runtime.getRuntime().addShutdownHook(shutdownHook);
```

4.2 Das Package java.util

Neben `java.lang` ist das Package `java.util` das wohl meist verwendete Package in Java. Einige seiner Klassen möchte ich im Folgenden vorstellen.

4.2.1 Arrays und Kollektionen

Arrays hatten wir bereits im Abschnitt 3.2.2 als Java-Sprachelelement kennengelernt. Die Java-Sprache definiert für Arrays zwar einige elementare Funktionen wie den Zugriff auf Array-Elemente, komplexere Operationen wie z. B. das Sortieren von Arrayelementen bleiben jedoch außen vor. Zwar können die Methoden der Klasse `Object` auch auf Arrays angewandt werden, jedoch reicht deren Funktionalität in vielen Fällen nicht aus. So ergibt beispielsweise

```
String[] array = new String[] { "Hello", "World" };
System.out.println(array);
```

die Ausgabe

```
[Ljava.lang.String;@3e25a5
```

also die interne Typdarstellung von `String[]` mitsamt der Objektidentifikation. Möchte man jedoch den Inhalt des Arrays ausgedruckt bekommen, sind andere Mittel notwendig.

So stellt das Package `java.util` die Klasse `Arrays` zur Verfügung, welche die fehlenden Operationen in Form statischer Methoden bereitstellt. Insbesondere werden folgende Methoden angeboten:

- `asList()` für die Umwandlung des Arrays in eine Liste (siehe unten).
- `binarySearch()` für die schnelle Suche nach einem gegebenen Array-Element.
- `copyOf()` erzeugt eine neue Kopie eines Arrays.
- `equals()` vergleicht zwei Arrays Element für Element.
- `fill()` füllt ein Array mit einem vorgegebenen Wert.
- `hashCode()` berechnet den Hash-Code eines Arrays aus seinen Elementen (siehe nächster Abschnitt).
- `sort()` sortiert ein Array entsprechend vorgegebenen Kriterien. Die Array-Elemente müssen dabei entweder primitiv sein oder aber das Interface `Comparable` implementieren.
- `toString()` wandelt ein Array in eine Zeichenkette um, wobei der Inhalt des Arrays angezeigt wird. Alternativ kann man auch die Methode `deepToString()` anwenden, um auch den Inhalt von Unter-Arrays anzuzeigen.

So ergibt denn

```
String[] array = new String[] { "Hello", "World" };
System.out.println(Arrays.toString(array));
```

die Ausgabe

```
[Hello, World]
```

Kollektionen

In Abschnitt 3.2.2 hatten wir bereits darauf hingewiesen, dass Arrays in vielen Fällen nicht flexibel genug sind. Schon von Anfang an stellte Java mit der Klasse `Vector` eine Speicherstruktur zur Verfügung, die dynamisch mit der Anzahl der hinzugefügten Elemente mitwachsen kann. Seit der Version 1.2 gibt es ein ganzes Arsenal solcher dynamischer Speicherstrukturen, die alle unter dem Interface `Collection` zusammengefasst werden. In der Regel sollte man die Unterklassen von `Collection` anstelle von `Vector` benutzen, da sie performanter sind. Nur wenn die jeweilige Speicherstruktur thread-sicher sein muss (siehe Abschnitt 5.1), ist der Einsatz der synchronisierten Klasse `Vector` sinnvoll.

Das Interface `Collection` erweitert seinerseits das Interface `Iterable`, dessen einzige Methode `iterator()` eine `Iterator`-Instanz bereitstellen kann, mit der alle Elemente der `Iterable`-Instanz durchlaufen werden können. Zu diesem Zweck besitzt `Iterator` die Methoden `hasNext()` und `next()`, so dass er leicht in `while`- oder `for`-Schleifen angewendet werden kann (siehe auch Abschnitt 3.8.4). In diesem Abschnitt hatten wir bereits darauf hingewiesen, dass bei den derzeit vorliegenden konkreten `Collection`-

Implementierungen eine Modifikation der `Collection` innerhalb einer `Iterator`-Schleife nicht zulässig ist. Die einzige Ausnahme ist das Entfernen von Elementen mittels der `remove()`-Methode des Iterators.

Das Interface `Collection` stellt einige allgemeine Methoden für Kollektionen zur Verfügung:

- Mit `add()` werden Elemente hinzugefügt, mit `remove()` wieder entfernt.
- Mit `addAll()` bzw. `removeAll()` können alle Elemente einer anderen Kollektion hinzugefügt bzw. entfernt werden.
- Mit `retainAll()` werden alle Elemente entfernt, die nicht in der als Argument angegebenen Kollektion enthalten sind.
- Die Methoden `contains()` bzw. `containsAll()` stellen fest, ob ein Element bzw. alle Elemente einer als Argument angegebenen Kollektion enthalten sind.
- Die Methode `size()` liefert die Größe der Kollektion; der Aufruf `isEmpty()` ist gleichbedeutend zu `size() == 0`.
- Mit der Methode `clear()` kann der Inhalt der Kollektion gelöscht werden.
- Mit der Methode `toArray()` kann eine Kollektion in ein Array umgewandelt werden.

Welche konkreten Kollektionen werden nun von dem Package `java.util` bereitgestellt? Im Wesentlichen gibt es hier drei verschiedene Gruppen:

- Listen, repräsentiert durch das Interface `List`
- Warteschlangen, repräsentiert durch das Interface `Queue`
- Mengen, repräsentiert durch das Interface `Set`

Listen

Bei Listen sind alle Elemente in einer genau festgelegten Reihenfolge angeordnet – Duplikate sind durchaus möglich. Deshalb besteht auch die Möglichkeit mit `get()` und `set()` gezielt auf Elemente zuzugreifen oder unter Angabe eines Index Elemente an genau bezeichneten Positionen mit `add()` hinzuzufügen bzw. mit `remove()` zu entfernen. Auch kann der Index eines Elements mit `indexOf()` bzw. `lastIndexOf()` festgestellt werden. Konkrete Implementierungen dieses Interface sind die `ArrayList` und die `LinkedList`. Erstere eignet sich insbesondere, wenn sich der Inhalt der Liste nicht allzu häufig ändert. Letztere verträgt häufige Änderungen besser, benötigt aber mehr Speicherplatz und das wahlfreie Abrufen einzelner Elemente mit `get()` dauert länger.

Als Beispiel zeige ich hier die Definition der Liste der Widgets aus dem *jDashboard*-Beispiel:

```
private List<Widget> widgets = new ArrayList<Widget>();
```

Im Konstruktor der `ArrayList` ist keine Anfangskapazität gesetzt. Deshalb wird der Standardwert 10 verwendet. Die Liste ist aber nicht auf diese Größe beschränkt, sondern kann sich bei Bedarf vergrößern. Ein automatisches Verkleinern der Speicherkapazität erfolgt bei diesem Listentyp nicht.

Beim Start des Dashboard wird die Konfigurationsdatei gelesen und entsprechend den Angaben in dieser Datei werden neue Widgets erzeugt und der Liste der Widgets hinzugefügt:

```
Widget widget = new Widget(...);
widgets.add(widget);
```

Beim Beenden des Dashboard soll jedem Widget die Möglichkeit gegeben werden, noch Aufräumarbeiten durchzuführen. Wir durchlaufen deshalb die Liste der Widgets in einer Schleife. Intern wird dabei allerdings ein Iterator verwendet (siehe Abschnitt 3.8.4):

```
private void stopWidgets() {
  for (Widget widget : widgets)
    try {
      widget.getApp().stop(widget);
    } catch (DashboardException e) {
      // Fehler beim Shutdown ignorieren
    }
}
```

Die `LinkedList` stellt außerdem noch eine konkrete Implementierung des Interface `Deque` (*double ended queue*) dar, einer Erweiterung des Interface `Queue`. Während die `Queue` mit den Methoden `add()`, `offer()`, `poll()`, `remove()`, `peek()` und `element()` nur *First-in-first-out*-Operationen (FIFO) zulässt, realisiert die `Deque` zusätzlich mit den Methoden `push()`, `pop()`, `peekLast()` u. a. auch noch einen Kellerspeicher. Außerdem wird mit der Methode `descendingIterator()` ein rückwärts laufender Iterator bereitgestellt.

Mengen

Für die durch das Interface `Set` beschriebenen Mengen gilt, dass alle darin enthaltenen Elemente verschieden sein müssen: Für alle Elemente `x` und `y` einer Menge muss gelten `x.equals(y) == false`. Das Hinzufügen eines Elements, das diese Bedingung verletzen würde, bleibt wirkungslos. Damit haben die folgenden Methoden direkte mengenalgebraische Entsprechungen:

- `set1.addAll(set2)` ergibt die Vereinigung der Mengen `set1` und `set2`. Das Ergebnis wird in `set1` gespeichert.
- `set1.retainAll(set2)` ergibt den Durchschnitt der Mengen `set1` und `set2`. Das Ergebnis wird in `set1` gespeichert.
- `set1.removeAll(set2)` subtrahiert die Menge `set2` von der Menge `set1`. Das Ergebnis wird in `set1` gespeichert.
- `set1.containsAll(set2)` prüft, ob `set2` eine Untermenge von `set1` ist.

Konkrete Implementierungen für dieses Interface liegen in Form der Klassen `HashSet`, `LinkedHashSet` und `TreeSet` vor. Während der `TreeSet` über einen binären Baum realisiert ist (und deshalb nur mit Objekten funktioniert, die das Interface `Comparable` implementieren) und für die gängigen Operationen eine Zugriffszeit von *log(n)* hat, sind `HashSet` und `LinkedHashSet` mithilfe von Hash-Tabellen (siehe nächster Abschnitt) realisiert. Bei den beiden Letzteren sind die Zugriffzeiten konstant, allerdings kann der Verwaltungsaufwand bei häufigen Änderungen höher sein. Iteriert man über einen `HashSet`, so erhält man die Elemente der Menge in einer nicht festgelegten Reihenfolge. Beim `LinkedHashSet` dagegen entspricht die Iterationsfolge der Reihenfolge beim Hinzufügen der Elemente – allerdings ist hier der Verwaltungsaufwand etwas höher.

Gemeinsam ist allen Kollektionen, dass sie keine primitiven Datentypen speichern können. Ist das trotzdem nötig, müssen diese Werte ge-»boxed«, also in Objekte eingepackt werden (siehe Abschnitt 4.1). Seit Java 5 erledigt allerdings der Compiler diese Arbeit. So kann man nun durchaus schreiben:

```
List<Integer> list = new ArrayList<Integer>();
list.add(512);
```

Man sollte sich allerdings darüber klar sein, dass der Compiler aus der zweiten Zeile ein

```
list.add(new Integer(512));
```

macht. Der Mehraufwand ist durch die Objekterzeugung erheblich und man sollte sich lieber überlegen, ob man die gewünschte Funktion nicht auch mit einem Array erreichen kann.

Wie wir gesehen haben, besitzen auch die Kollektionen keine Methoden für das Sortieren, die binäre Suche und andere allgemeine Funktionen. Ähnlich wie bei den Arrays werden diese Funktionen in Form von statischen Methoden der Klasse `Collections` bereitgestellt. Neben `sort()` und `binarySearch()` findet man hier eine Anzahl nützlicher Funktionen, darunter auch die Methoden `emptyList()` und `emptySet()`, die (unveränderliche) leere Listen und Mengen abliefern. Da diese Objekte statisch definiert sind, wird bei deren Verwendung der *Garbage Collector* geschont.

Beispiel: die Klasse PreferenceChoice

Bei Benutzerpräferenzen kommt es mitunter vor, dass ein Wert aus einer Liste möglicher Werte ausgewählt werden kann. Um sowohl den Wert als auch die Auswahlliste selbst innerhalb eines einzigen Objekts zu einem Editor transportieren zu können, benötigen wir eine besondere Datenstruktur in Form der Klasse `PreferenceChoice`. Diese Klasse ist als Unterklasse von `ArrayList` angelegt. So kann auf die Liste der Auswahlmöglichkeiten direkt mit den Methoden der Klasse `ArrayList` zugegriffen werden. Zusätzlich wird noch das Feld `value` für den ausgewählten Wert benötigt.

Der Konstruktor dieser Klasse akzeptiert den aktuellen Wert und eine weitere Zeichenkette. Die Zeichenkette enthält alle möglichen Werte jeweils durch Kommata getrennt. Ein `StringTokenizer` wird benutzt, um diese Zeichenkette in Einzelwerte zu zerlegen. Im Konstruktor des `StringTokenizer` wird im zweiten Parameter ein Komma als Trennzeichen übergeben. In der `while`-Schleife wird jeweils ein einzelner Wert mithilfe der `add()`-Methode der `ArrayList` in die Liste aufgenommen.

Beim Setzen des Werts `value` wird zusätzlich noch mit der `ArrayList`-Methode `contains()` überprüft, ob der Wert in der Liste der möglichen Werte vorkommt. Falls nicht, wird der Wert nicht gesetzt.

```
package com.bdaum.dashboard.internal;

import java.util.ArrayList;
import java.util.StringTokenizer;

/**
 * Diese Klasse speichert einen Wert und eine Enumeration von
 * Auswahlmöglichkeiten für den Wert
 */
public class PreferenceChoice extends ArrayList<String> {

  private String value;

  /**
   * Konstruktor
   * @param value - Wert
   * @param items - Auswahlmöglichkeiten, durch Kommata getrennt
   */
  public PreferenceChoice(String value, String items) {
    // Auswahlmöglichkeiten in Einzelwerte zerlegen
    // und zur Liste hinzufügen
    StringTokenizer st = new StringTokenizer(items, ",");
    while (st.hasMoreTokens())
      add(st.nextToken());
    setValue(value);
  }
```

```
  @Override
  public String toString() {
    return value;
  }

  /**
   * Überprüft den übergebenen Wert gegen die Auswahlmöglichkeiten und
   * setzt den Wert neu
   * @param value - neuer Wert
   */
  public void setValue(String value) {
    if (contains(value))
      this.value = value;
  }
}
```

4.2.2 Zuordnungstabellen

Eng verwandt mit den Kollektionen sind die Zuordnungstabellen, die unter dem Interface `Map` zusammengefasst sind. Zuordnungstabellen gestatten den Zugriff auf Tabellenelemente über einen Schlüssel. Das Interface `Map` definiert insbesondere die Methoden:

- `put()` zum Eintragen von Werten unter einem Schlüsselbegriff. Ist bereits ein Wert unter dem betreffenden Schlüssel eingetragen, wird der alte Wert überschrieben.
- `get()` zum Zugriff auf Werte mittels eines Schlüssels. Ist kein Wert unter dem Schlüssel gespeichert, wird `null` zurückgegeben.
- `remove()` zum Entfernen von Werten anhand eines Schlüssels. Dabei wird der gelöschte Wert zurückgegeben, sofern vorhanden.

Daneben gibt es noch einige Abfragemethoden wie `containsKey()`, `containsValue()` oder `isEmpty()`, um auf das Vorhandensein eines Schlüssels, eines Werts oder überhaupt eines Eintrags zu testen. Außerdem kann man die Schlüssel- und Wertemengen mit `keySet()` bzw. `values()` als `Set`-Instanzen abfordern, so dass man auch über die Menge aller Schlüssel bzw. Werte iterieren kann. Die Menge aller Einträge (vom Typ `Map.Entry`) erhält man mit der Methode `entrySet()`. Mit `putAll()`können die Einträge einer anderen `Map`-Instanz in das aktuelle `Map`-Objekt übernommen werden.

Die wichtigsten `Map`-Implementierungen sind `HashMap` und `TreeMap`. Wie schon bei `TreeSet` werden bei der `TreeMap` die Schlüsselbegriffe in einem binären Baum angeordnet (und müssen deshalb das Interface `Comparable` implementieren). Die Zugriffszeit auf `Map`-Elemente beträgt *log(n)*. Dafür sind diese Maps recht änderungsfreundlich.

Die HashMap funktioniert dagegen so, dass für jeden Schlüssel mithilfe der Methode hashCode() des Schlüsselobjekts ein Index berechnet wird. Unter diesem Index wird dann der Map.Entry-Eintrag abgelegt. Beim Zugriff wird auch wieder nur der HashCode des Schlüsselobjekts errechnet, darüber direkt auf den Map.Entry-Eintrag zugegriffen, der Schlüsselwert mithilfe der equals()-Methode überprüft und der Wert entnommen. Freilich kann es beim Abspeichern vorkommen, dass mehrere Schlüssel den gleichen Indexwert ergeben. Dann werden an dem betreffenden Platz mehrere Map.Entry-Einträge in einer Liste zusammengefasst. Beim Lesezugriff muss umgekehrt der richtige Map.Entry-Eintrag durch einen Schlüsselvergleich herausgefunden werden. Damit solche Mehrfacheinträge nicht zu oft vorkommen, ist es nötig, die HashMap nicht zu klein anzulegen. Auf der anderen Seite sollte sie auch nicht zu groß angelegt werden, denn sonst werden die Methoden keySet(), values() und entrySet() zu langsam. HashMaps wachsen automatisch, wenn der Platz zu eng wird. Dies wird vom Ladefaktor (*loadFactor*) bestimmt, der im Konstruktor angegeben werden kann – der Vorgabewert ist 0.75. Übersteigt die Anzahl der Elemente die Gesamtkapazität der HashMap multipliziert mit dem Ladefaktor, so wird die Kapazität automatisch auf etwa das Doppelte vergrößert und die gesamte HashMap reorganisiert. Es ist eine ganz gute Praxis, die Anfangsgröße der HashMap im Konstruktor so zu wählen, dass diese Reorganisation praktisch nie vorkommt: Dazu setzt man die Anfangsgröße auf *(erwartete Anzahl)/Ladefaktor*. Werden Elemente aus der HashMap gelöscht, wird die Kapazität nicht verringert.

Eine Sonderform der HashMap ist die LinkedHashMap, die dafür sorgt, dass die Reihenfolge der Elemente bei der Iteration mit der Reihenfolge beim Hinzufügen übereinstimmt.

Beispiel

In dem in Kapitel 10 implementierten Server für den Nachrichtenaustausch zwischen *jDashboard*-Benutzern wird darüber Buch geführt, wann welches Themengebiet zum letzten Mal geändert wurde. Dies ist ein typischer Anwendungsfall für eine HashMap, die z. B. so definiert wird:

```
private Map<String, Long> modifications = new HashMap<String, Long>();
```

Beim Abspeichern oder Löschen einer Nachricht wird dann einfach die Methode setLastModification() unter Angabe des Themengebiets aufgerufen. Diese Methode speichert unter dem Namen des Themengebiets einen Zeitstempel als Long-Instanz in der HashMap:

```
private void setLastModification(String topic) {
  modifications.put(topic, System.currentTimeMillis());
}
```

Will ein Client wissen, wann ein bestimmtes Themengebiet zum letzten Mal geändert wurde, genügt der Aufruf der Methode `getLastModification()`:

```
public long getLastModification(String topic) {
  return modifications.get(topic);
}
```

Auch hier wurde das *Auto-Boxing* angewandt, welches zunächst den Zeitstempel, der im Format `long` vorlag, in ein Objekt des Typs `Long` umwandelt und dann in der `HashMap` speichert. Bei der Rückgabe des Werts erfolgt dann umgekehrt das *Unboxing* von `Long` nach `long`.

Wie wir gesehen haben, spielen beim Schlüsselobjekt einer `HashMap` die Methoden `hashCode()` und `equals()` eine wichtige Rolle. Damit `HashMap`s richtig funktionieren, müssen die beiden Methoden zueinander passen. Zwei Objekte, die von `equals()` als gleich erkannt werden, müssen auch den gleichen Hash-Code liefern. Wenn also bei der Definition einer Klasse die von der Superklasse (z. B. `Object`) geerbten Methoden `hashCode()` bzw. `equals()` überschrieben werden, müssen deshalb in der Regel beide Methoden überschrieben werden.

Beispiel: die Klasse NamedPreferenceList

Wie schon bei der Klasse `PreferenceChoice` im vorigen Abschnitt geht es auch bei der Klasse `NamedPreferenceList` um das Editieren von Präferenzen. Hier geht es allerdings darum, eine ganze Liste benannter Werte darzustellen. Ein konkretes Beispiel ist beim Internetradio die Liste der Radiostationen. Jeder Eintrag besteht aus einem Stationsnamen und einem URL. Eine solche Datenstruktur lässt sich mühelos mithilfe einer `Map` abbilden, wir haben hier aber noch besondere Anforderungen: Die Liste der Einträge hat eine festgelegte Reihenfolge und wir möchten wissen, welche Bedeutung die Werte der Einträge haben. Ist der `Map`-Schlüssel ein Stationsname oder eine Automarke? Handelt es sich bei dem Wert um einen URL oder um eine Stadt? Diese Metadaten wollen wir zusätzlich in unsere Datenstruktur mit aufnehmen.

Die erste Anforderung – die festgelegte Reihenfolge – lässt sich durch die Verwendung einer `LinkedHashMap` erfüllen. Wir bauen deshalb unsere Klasse als Unterklasse von `LinkedHashMap` auf. Zusätzlich werden jedoch auch noch die Metadaten im `String`-Array `description` gespeichert. Im Konstruktor nehmen wir an, dass die Metadaten in der Form *NamensÜberschrift=WertÜberschrift* übergeben werden. Wir zerlegen diese Zeichenkette, indem wir die Position des Gleichheitszeichens feststellen und dann die Zeichenketten vor und nach dem Gleichheitszeichen an das `String`-Array zuweisen.

Die Wertangabe `values` im Konstruktor enthält die einzelnen Einträge jeweils durch Semikolon getrennt. Ein `StringTokenizer` wird benutzt, um diese Zeichenkette in Einzelwerte zu zerlegen. Im Konstruktor des `StringTokenizer` wird deshalb im zweiten Parameter ein Semikolon als Trennzeichen übergeben. In der `while`-Schleife wird

jeweils ein Einzelwert mithilfe der Methode `nextToken()` vom `StringTokenizer` erhalten. Dieser Einzelwert wird dann ähnlich wie die Metadaten analysiert und in einen Namens- und einen Wertteil zerlegt. Die Kombination aus Name und Wert wird dann mit der `LinkedHashMap`-Methode `put()` abgespeichert.

Natürlich wollen wir die Liste der Einträge auch wieder in der serialisierten Form als Zeichenkette zurückhaben. Das geschieht durch Überschreiben der schon in der Klasse `Object` definierten Methode `toString()`. Hier legen wir zunächst eine Instanz der Klasse `StringBuilder` an, in welcher die neue Zeichenkette Schritt für Schritt aufgebaut wird. Das ist viel effizienter, als direkt einzelne Zeichenketten miteinander zu verknüpfen. Mit der `for`-Schleife iterieren wir dann über alle Werte der `Set`-Instanz, die wir von der `LinkedHashMap`-Methode `keySet()` in der Variablen `key` erhalten. Diese Menge enthält alle in der `LinkedHashMap` eingetragenen Schlüsselwerte. Mit dem Schlüsselwert können wir uns dann von der `LinkedHashMap` mit der Methode `get()` den zugehörigen Wert holen. Schlüssel, Wert und Trennzeichen fügen wir mit der Methode `append()` der `StringBuilder`-Instanz hinzu. Diese Methode liefert die `StringBuilder`-Instanz wieder als Ergebnis ab, weshalb sie in der gezeigten Weise aneinandergekettet werden kann.

```
package com.bdaum.dashboard.internal;
import java.util.LinkedHashMap;
import java.util.StringTokenizer;

/**
 * Diese Klasse speichert Werte unter einem Namen. Für Namen und Werte
 * können Spaltenköpfe gespeichert werden.
 */
class NamedPreferenceList extends LinkedHashMap<String, String> {

  private String[] description = new String[2];

  /**
   * Konstruktor
   * @param values - Namenswertepaare in der Form name1=wert1;name2=wert2;...
   * @param description - Spaltenköpfe in der Form name=wert
   */
  public NamedPreferenceList(String values, String description) {
    // Spaltenköpfe aufbereiten
    int p = description.indexOf('=');
    if (p > 0) {
      this.description[0] = description.substring(0, p);
      this.description[1] = description.substring(p + 1);
    } else {
      this.description[0] = description;
      this.description[1] = "";
    }
```

```
    // Wertangabe in Einzelwerte zerlegen und in Map speichern
    StringTokenizer st = new StringTokenizer(values, ";");
    while (st.hasMoreTokens()) {
      String token = st.nextToken();
      p = token.indexOf('=');
      if (p > 0)
        put(token.substring(0, p), token.substring(p + 1));
      else
        put(token, "");
    }
  }

  @Override
  public String toString() {
    // Werte wieder zu einer Zeichenkette zusammensetzen
    StringBuilder sb = new StringBuilder();
    for (String key : keySet()) {
      if (sb.length() > 0)
        sb.append(';');
      sb.append(key).append("=").append(get(key));
    }
    return sb.toString();
  }

  /**
   * Lieferte die beiden Spaltenüberschriften
   * @return - Spaltenüberschriften
   */
  public String[] getDescription() {
    return description;
  }
}
```

4.2.3 Eigenschaftstabellen

Eine Sonderform einer Zuordnungstabelle ist die Klasse `Properties`, die verwendet wird, um alle Arten von Konfigurationsdaten zu verwalten. Dazu gehören z. B. Benutzerpräferenzen, aber auch sprachabhängige Textkonstanten. Die Klasse `Properties` verfügt über spezielle Methoden zum Setzen (`setProperty()`) und zur Abfrage (`getProperty()`) von Eigenschaften, wobei jede Eigenschaft eindeutig über einen Schlüssel identifiziert wird. Außerdem gibt es `load()`-Methoden für das Laden einer `Properties`-Tabelle von einem Eingabestrom und umgekehrt `store()`-Methoden für das Schreiben auf einen Ausgabestrom.

Eigenschaftstabellen werden traditionell in `.properties`-Dateien gespeichert. Das Format ist recht einfach: Schlüssel und Wert werden durch ein Gleichheitszeichen getrennt. Der umgekehrte Schrägstrich `\` kann als Fluchtsymbol verwendet werden,

falls Steuerzeichen im Schlüssel oder Wert als normale Zeichen behandelt werden sollen. Ein \ am Ende einer Zeile zeigt an, dass die nächste Zeile eine Fortsetzungszeile ist. Hier ist als Beispiel die application.properties-Datei unseres Taschenrechnerbeispiels:

```
about=Calculator - (c) 2007 Berthold Daum
displayColor=#00C000
```

Alternativ kann seit Java 5.0 auch ein XML-Format benutzt werden. Das Laden and Abspeichern erfolgt dann mit den Methoden loadFromXml() bzw. storeToXml(). Das Format wird durch die DTD http://java.sun.com/dtd/properties.dtd beschrieben. Unsere Beispieldatei würde im XML-Format folgendermaßen aussehen:

```
<properties version="1.0">
  <element key="about"> Calculator - (c) 2007 Berthold Daum </element>
  <element key=" displayColor ">#00C000</element>
</properties>
```

Anders als die im vorherigen Abschnitt gezeigte Klasse NamedPreferenceList bewahren Properties nicht die Reihenfolge der Einträge, denn sie basieren intern auf der einfachen HashMap, nicht auf der LinkedHashMap.

4.2.4 Datum und Zeit

Ein weiterer wichtiger Funktionsbereich des Package java.util ist die Datumsverarbeitung und die Bereitstellung eines Zeitgebers.

Kalenderfunktionen

An den vielen @deprecated-Einträgen (veraltet) im API der Datumsklassen kann man sehen, dass Datumsverarbeitung kein einfaches Feld ist. Schon seit mehreren Jahrtausenden mühen sich Mathematiker, Physiker, Historiker und Astronomen mit den Problemen von Datum und Zeit ab. Auch bei Java dauerte es einige Zeit – die Sommerzeitumstellung lief erst seit Java 5 richtig. Doch schauen wir uns die Datumsklassen und -typen der Reihe nach an.

In den Datumsklassen aus java.util wie Date oder den Unterklassen von Calendar werden intern Zeit und Datum durch einen long-Wert dargestellt. Dieser Wert repräsentiert die Anzahl der Millisekunden seit dem 1. Januar 1970, 0:00. In dieser Form liefert auch die Methode System.currentMillis() (siehe Abschnitt 4.1.2) den aktuellen Zeitstempel ab. Den aktuellen Zeitstempel in Objektform erhält man auch, wenn Date- oder Calendar-Konstruktoren ohne die Datums- oder Zeitangaben aufgerufen werden, z. B.:

```
Date now = new Date();
```

Die Millisekunden bekommt man dann von einer `Date`-Instanz mit `getTime()`. Die meisten anderen Methoden von `Date` sind jedoch inzwischen als `@deprecated` markiert. Will man Datumsarithmetik betreiben oder ist man z. B. am Wochentag oder Monat interessiert, so verwendet man besser eine Unterklasse von `Calendar`. Davon gibt es in Java 6 gleich drei: den `GregorianCalendar`, der in der westlichen Welt seit dem 15. Oktober 1582 gilt, den `BuddhistCalendar` und den `JapaneseImperialCalendar`.

Von `Calendar`-Objekten erhalten Sie die Millisekunden über die Methode `getTimeInMillis()`. Als `Date`-Objekt wird die Zeit von der Methode `getTime()` abgeliefert. Mithilfe der Methode `get()` erhält man unter Angabe eines Schlüssels folgende Werte:

Ära	`ERA`
	ergibt beim Julianischen Kalender die Werte `GregorianCalendar.AD` (n. u. Z.) oder `GregorianCalendar.BC` (v. u. Z.).
Jahr	`YEAR`
Monat	`MONTH` **Achtung:** `JANUARY == 0`
	Für Mondkalender gibt es den zusätzlichen Wert `UNDECIMBER` (12).
Wochennummer	`WEEK_OF_YEAR`
	Zählung beginnt bei 1.
Woche des Monats	`WEEK_OF_MONTH` Zählung beginnt bei 1.
Tag	`DAY_OF_MONTH` oder `DATE`
Tag im Jahr	`DAY_OF_YEAR`
Wochentag	`DAY_OF_WEEK`
	Zählung beginnt bei 1 (`SUNDAY`).
vor-/nachmittags	`AM_PM`
Stunden auf 12-Stunden-Basis	`HOUR`
Stunden auf 24-Stunden-Basis	`HOUR_OF_DAY`
Minute	`MINUTE`
Sekunde	`SECOND`
Millisekunde	`MILLISECOND`
Zeitzone	`ZONE_OFFSET`
	Zeitverschiebung gegen GMT in Millisekunden
Sommerzeitverschiebung	`DST_OFFSET`
	Zeitverschiebung gegen Winterzeit in Millisekunden

Schön ist, dass man jedes dieser Felder getrennt modifizieren kann: Mit `set()` kann ein neuer Wert gesetzt, mit `add()` ein Wert addiert oder mit `clear()` ein Feld zurückgesetzt werden. Der interne Zeitstempel wird dabei entsprechend angepasst.

Beim `GregorianCalendar` kann zusätzlich mit der Methode `isLeapYear()` abgefragt werden, ob das Datum in einem Schaltjahr liegt. Normalerweise benutzt die Klasse `GregorianCalendar` für Datumswerte vor dem 15. Oktober 1582 die Algorithmen des Julianischen Kalenders. Da der Gregorianische Kalender jedoch nicht gleichzeitig in allen Ländern eingeführt wurde, besteht die Möglichkeit, den Einführungszeitpunkt mit der Methode `setGregorianChange()` zu setzen. Wie gesagt, Datum und Zeit sind kein einfaches Feld ...

Alle `Calendar`-Klassen arbeiten mit Zeitzonen und eignen sich deshalb auch für zeitzonenüberspannende Anwendungen. Wird bei der Erzeugung eines `Calendar`-Objekts die Zeitzone nicht mit angegeben, so wird als Zeitzone die der Ablaufplattform eingetragen.

Probleme bei der Datumsverarbeitung entstehen insbesondere dann, wenn man direkt mit den von der Klasse `System` gelieferten Millisekunden rechnet. So könnte man mit

```
long millis = System.getCurrentMillis();
int millisOfDay = (int) (millis % (24*60*60*1000));
int hour = millisOfDay / (60*60*1000);
```

die aktuelle Stunde berechnen, hätte allerdings im Sommer dann die falsche Zeit, denn die Zeitverschiebung durch die Sommerzeit wird auf diese Weise nicht berücksichtigt. Richtig (und auch bequemer) geht's mit:

```
GregorianCalendar now = new GregorianCalendar();
int hour = now.get(GregorianCalendar.HOUR_OF_DAY);
```

Während die `toString()`-Methode bei `Date`-Objekten eine Zeichenkette vom Format

```
Fri Jan 19 21:25:09 CET 2007
```

liefert, erhält man bei `Calendar`-Objekten eine Auflistung sämtlicher Felder. Für die Anzeige an Endbenutzer sind deshalb spezielle Aufbereitungsschritte nötig. Dazu steht im Package `java.text` die Klasse `SimpleDateFormat` zur Verfügung. Um Datums- und Zeitangaben aufzubereiten, muss zunächst eine `SimpleDateFormat`-Instanz unter Angabe einer Maske spezifiziert werden. Dann können mithilfe dieser Instanz `Date`-Objekte formatiert werden. So ergibt:

```
java.util.Date date = new java.util.Date();
java.text.SimpleDateFormat df =
        new java.text.SimpleDateFormat("EEEEEEEEEE, 'den' d.MMMMMMMMM yyyy");
String s = df.format(date);
```

die Zeichenkette

```
Freitag, den 19.Januar 2007
```

Die einzelnen Maskenzeichen sind in der Dokumentation der Klasse `SimpleDateFormat` ausführlich erklärt. Die Anzahl ihrer Wiederholungen legt die maximal mögliche Ausgabelänge für den jeweiligen Wert fest. Textkonstanten innerhalb der Maske müssen in Hochkommata eingeschlossen werden, wenn sie Maskenzeichen enthalten. Da in der Regel das gleiche Datumsformat immer wieder verwendet wird, empfiehlt sich die Definition der `SimpleDateFormat`-Instanz als Konstante (`static final`).

Der Zeitgeber

Die Klasse `Timer` wird vom Package `java.util` als Zeitgeber bereitgestellt. Damit ist es möglich, einmalige Ereignisse oder periodisch wiederkehrende Ereignisse auszulösen. Die Klasse `Timer` stellt dazu mehrere Varianten der Methode `schedule()` zur Verfügung. Im folgenden Beispiel aus einer frühen Version der *jDashboard*-Uhr sehen wir, wie beim Start der Uhr ein neuer Zeitgeber eingerichtet und dann gestartet wird. Die Methode `schedule()` startet nach einer ersten Verzögerung von 1000 Millisekunden den als anonyme innere Klasse angegebenen `TimerTask` und startet ihn dann nach jeweils 1000 Millisekunden erneut. In dessen `run()`-Methode werden dann die auszuführenden Operationen nebenläufig (siehe Abschnitt 5.1) ausgeführt. `Timer`-Instanzen sind in der Lage, mehrere `TimerTask`s zu verwalten.

```
private Timer timer;

@Override
public JComponent start(Widget widget) throws DashboardException {
  final ClockPanel clockPanel = (ClockPanel) super.start(widget);
  timer = new Timer();
  timer.schedule(new TimerTask() {
    @Override
    public void run() {
      clockPanel.display(System.currentTimeMillis());
    }
  }, 1000, 1000);
  return clockPanel;
}
```

Sowohl `Timer` als auch `TimerTask` besitzen eine `cancel()`-Methode, mit welcher der Zeitgeber gestoppt werden kann. Wir nutzen das in der `stop()`-Methode des Uhren-Widget:

```
@Override
  public void stop(Widget widget) throws DashboardException {
    if (timer != null)
      timer.cancel();
    super.stop(widget);
  }
```

Doch leider hat der `Timer` für Anwendungen wie unsere Uhr einen kleinen Schönheitsfehler. Wird die Systemzeit zurückgestellt, so bleibt die Uhr erst einmal für einige Zeit stehen – so lange, bis wieder die vorgemerkte Schaltzeit erreicht ist. In Abschnitt 5.1.1 zeigen wir, wie die oben gezeigte Timer-Lösung durch eine eigene robustere Lösung ersetzt werden kann.

4.2.5 Reguläre Ausdrücke

Schon in Abschnitt 4.1.1 hatte ich die `String`-Methoden `replaceAll()` und `replaceFirst()` erwähnt, mit denen komplexe Umformungen von Zeichenketten vorgenommen werden können. Diese Methoden beruhen auf der Klasse `Pattern` aus dem Package `java.util.regex`, das mit Java 1.4 eingeführt wurde. »regex« steht für *regular expression* (regulärer Ausdruck). Reguläre Ausdrücke beruhen auf der Theorie der regulären Mengen, die von dem Mathematiker *Stephen Cole Kleene* Mitte der fünfziger Jahre entwickelt wurde. Während reguläre Ausdrücke z. B. für *Perl*-Programmierer zum täglichen Brot gehören, sind Kenntnisse über diese Algebra bei Java-Programmierern eher spärlich. Deshalb vorweg eine kurze Auffrischung.

Die Grundlagen der Algebra der regulären Mengen sind recht einfach. Wir setzen zunächst ein endliches Alphabet Σ voraus. Das können alle möglichen Symbole sein, aber für unseren Zweck genügt die Menge der Unicode-Zeichen. Die Zeichen bzw. Zeichenfolgen aus diesem Alphabet notieren wir hier mit Kleinbuchstaben, Mengen mit Großbuchstaben. Außer diesen Zeichen benötigen wir noch zwei spezielle Konstanten: die leere Menge `{}` und die leere Zeichenkette ε. Drei Operationen genügen uns, um reguläre Mengen zu konstruieren:

Notation	Definition	Beispiel
`RS`	Verkettung. `RS` bezeichnet die Menge { `ab` | mit `a` in `R` und `b` in `S` }, also die Menge aller Verkettungen aus je einem Element aus der Menge `R` und der Menge `S`.	`{"ab", "c"}{"x", "yz"}` `= {"abx", "abyz", "cx", "cyz"}`
`R | S`	Die klassische Mengenvereinigung	`{"ab", "c"} | {"x", "yz"}` `= {"ab", "c"", "x", "yz"}`
`R*`	Die Kleenesche Stern-Operation, auch Powerset genannt. `R*` bezeichnet die kleinstmögliche Obermenge von `R`, die ε enthält und unter der Verkettungsoperation geschlossen ist (d.h., Verkettungen zwischen Elementen aus `R` erzeugen keine neuen Elemente in `R`). Wenn `R*` nicht nur aus der leeren Zeichenkette besteht, ist `R*` immer unendlich.	{"ab", "c"}* = { , "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcab", "abcc", "cabab", "cabc", "ccab", "ccc", ... }

Ausdrücke, die aus den oben definierten Konstanten, Symbolen des Alphabets Σ und den hier definierten Operationen gebildet werden, heißen reguläre Ausdrücke. Jede reguläre Menge kann aus dem gegebenen Alphabet Σ mit einer endlichen Zahl der oben definierten Operationen erzeugt werden.

Schauen wir uns nun die Umsetzung dieser Theorie, die im Laufe der Jahre noch verfeinert wurde, in Java an. Die Methode `matches()` erlaubt im ersten Parameter die Angabe eines regulären Musters, das auf die im zweiten Parameter übergebene Zeichenkette angewandt wird. So ergibt beispielsweise der Ausdruck

```
Pattern.matches("a*b", "aaaaab");
```

den Wert `true`. Dieses Muster überprüft, ob der zweite Parameter aus einer beliebigen Anzahl von `a` gefolgt von einem `b` besteht.

Im Muster können die meisten Zeichen direkt angegeben werden, einigen Sonderzeichen muss jedoch das Fluchtsymbol `\` vorangestellt werden. `\\` steht für das Fluchtsymbol selbst. Dabei ist zu beachten, dass das Zeichen `\`, wenn in einem `String`-Literal angegeben, verdoppelt werden muss, da es auch dort als Fluchtsymbol gilt: Der Ausdruck

```
myString.replaceFirst("\\\\","/")
```

bedeutet also: Ersetze das erste `\` durch ein `/`.

Aus diesen Zeichen können sogenannte Zeichenklassen gebildet werden. So bezeichnet z. B. die Zeichenklasse `[abc]` ein Zeichen, das entweder `a`, `b` oder `c` sein darf. Für die Konstruktion solcher Zeichenklassen stehen eine Reihe von Operatoren zur Verfügung wie `^` (Negation), – (Bereich) oder `&&` (Durchschnitt). So bedeutet z. B. der Ausdruck

```
[a-z&&[^i-k]]
```

alle Zeichen von `a` bis `z`, aber nicht die Zeichen von `i` bis `k`.

Eine ganze Reihe von Zeichenklassen sind bereits vordefiniert. So steht z. B. `\d` für eine Ziffer (0–9), `\p{Space}` für alle Leerzeichen, `\p{Lower}` für alle Kleinbuchstaben usw.

Hinter Zeichen und Zeichenklassen können Quantifizierungsoperatoren angegeben werden, welche bestimmen, wie oft sich ein solches Zeichen wiederholen darf. Dabei wird unterschieden zwischen gierigen (*greedy*), zögerlichen (*reluctant*) und besitzergreifenden (*possessive*) Operatoren. Die bekanntesten (gierigen) Operatoren sind `?` (einmal oder keinmal), `*` (beliebig oft) und `+` (mindestens einmal). Gierige Operatoren verbrauchen so viel wie möglich Zeichen von der zu untersuchenden Zeichenkette, wobei sie darauf achten, dass der Rest der Zeichenkette möglichst noch auf den Rest des Musters passt. Zögerliche Operationen, denen ein `?` nachgestellt wird, verbrauchen so wenig wie möglich Zeichen. Besitzergreifende Operationen, denen ein `+` nachgestellt wird, verbrauchen so viel wie möglich Zeichen, wobei sie sich nicht darum kümmern, ob der Rest noch passt.

Durch die Definition von geklammerten Gruppen können Teilzeichenketten aufgezeichnet (*captured*) werden. Die Zeichenkette, die mit dem Ausdruck in der Klammer übereinstimmt, gilt als *captured* und lässt sich in der Folge mit dem Ausdruck `\n` abrufen. Dabei gibt *n* die Nummer der Klammergruppe an, wobei die Zählung bei 1 beginnt. So akzeptiert z. B. das Muster

```
(\p{Lower}*)\1\1
```

alle Zeichenfolgen aus Kleinbuchstaben, die sich exakt dreimal wiederholen, z. B. `hallohallohallo`. Diese Capture-Gruppen können auch Daten zwischen dem ersten und zweiten Parameter der `String`-Methoden `replaceAll()` und `replaceFirst()` transportieren. Allerdings wird in der Ersetzungszeichenkette eine variable Gruppe nicht mit `\` gekennzeichnet, sondern mit `$`. So erzeugt die Anweisung

```
System.out.println("Das ist 'schick'".replaceFirst("'([a-z]*)'", "-$1-"));
```

die Ausgabe

```
Das ist -schick-
```

Die Syntax für reguläre Ausdrücke ist ziemlich prägnant in der Javadoc der Klasse `Pattern` beschrieben. Ist Ihnen diese Information zu knapp, finden Sie dort Verweise auf zusätzliche Literatur. Auch in der Perl- und PHP-Community gibt es einige gute Tutorien über reguläre Ausdrücke.

Technisch arbeitet die Klasse `Pattern` so, dass der Suchausdruck zunächst in einen endlichen Automaten umgewandelt wird, der dann die zu durchsuchende Zeichenkette verarbeitet. Da die Transformation in einen solchen Automaten einige Rüstzeiten benötigt, empfiehlt es sich, bei wiederholter Verwendung des gleichen Musters dieses zunächst in einen Automaten zu kompilieren und dann das Kompilat auf die zu durchsuchenden Zeichenketten anzusetzen:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
m = p.matcher("aaaaaaaaaab");
boolean b = m.matches();
```