# Jenseits von Babel

Anwender wollen Menüs und Meldungen von Programmen gern in ihrer Landessprache haben. Dieser Coffee-Shop zeigt, wie einfach es ist, internationalisierte Programme in Java ohne zusätzlichen Aufwand zu lokalisieren und an andere Sprachen anzupassen. Bernhard Bablok


**Wer** Open-Source-Programme schreibt, verwendet als Sprache für Ausgaben, GUI-Elemente und Kommentare meist erst mal Englisch – sowohl für das Programm als auch für Meldungen und Beschriftungen. Das hat verschiedenste Gründe. Entweder herrscht großer Zeitdruck oder das Programm ändert seinen Status fließend vom Prototyp zu einer Produktionsversion. Endanwender wollen aber Software in ihrer eigenen Sprache. Das fällt naturgemäß bei erfolgreichen Programmen besonders auf – die Nutzer in den nicht-englischsprachigen Ländern drängen dann den Entwickler, seine Software nachträglich zu internationalisieren.

Internationalization, kurz I18n wegen der 18 Buchstaben zwischen dem I und dem n, bezeichnet ein Design, bei dem die Anwendung an verschiedene Sprachen und lokale Gegebenheiten (etwa das Datumsformat) angepasst werden kann, ohne dass Programmänderungen notwendig sind. Mit Localization, analog mit L10n abgekürzt, bezeichnet man den konkreten Übersetzungsprozess für eine Sprache samt Anpassung an lokale Besonderheiten.

## Einfache Mechanismen

Internationalisierte Programme sollen ohne großen Aufwand und vor allem ohne Neukompilierung zusätzliche Sprachen unterstützen und spezialisierte Daten, etwa Adressen, Datumsangaben und Geldbeträge, in den landesüblichen Formaten ausgeben. Textelemente, beispielsweise Beschriftungen von GUI-Elementen oder Meldungen, sind deshalb außerhalb des Programms gespeichert; das Programm lädt sie dynamisch. Java unterstützt I18n und L10n durch eine Reihe von Klassen und zwei grundlegende Mechanismen.

## Zentrale Klassen des I18n-API

Zum Ersten werden Textelemente nicht hart kodiert, sondern über Schlüssel angesprochen. Die Suche nach dem richtigen – oder besser dem am besten geeigneten – Wert zum Schlüssel übernimmt eine Java-Klasse. Zum Zweiten hat Java spezielle Formatierungsklassen, die eingebautes Wissen über Datums- und Währungsformate besitzen.

Bei beiden Mechanismen ist es ausreichend, die aktuelle Sprache und das Land zu definieren. Viel Arbeit fällt hier nicht an, denn die Java-Runtime holt sich Defaultwerte vom Betriebssystem. Sollen dagegen auf einem deutschsprachigen Rechner die Meldungen in Englisch erscheinen, kann der Entwickler das explizit festlegen. Nützlich ist so ein Verhalten für Serverprozesse, die entweder immer in einer festen Sprache mit dem Client kommunizieren oder je nach Client anders lokalisierte Meldungen ausgeben.

Die Lokalisierung steuern Instanzen der Klasse »java.util.Locale«. Sie haben selbst keine eigene Funktionalität, sondern dienen als Marker für andere Klassen. Der Aufruf des Konstruktors erfolgt mit dem Sprachcode (ISO-Sprachcode gemäß [1]) und Länderkennung [2]. Typische Konstruktoraufrufe sehen also wie folgt aus:

```
Locale a = new Locale("en","US");
Locale b = new Locale("en","UK");
Locale c = new Locale("en");
```

Letzteres spezifiziert nur die Sprache, nicht die Region und ist immer dann ausreichend, wenn es keine Unterschiede in Texten gibt und das Programm keine Währungen, Daten und Zahlen darstellen muss. Ein dritter Parameter wäre eine applikationsspezifische Variable, die aber in vielen Fällen nicht unbedingt nötig ist. ▶

Die Klasse »java.util.ResourceBundle« dient dazu, Texte über Schlüssel anzusprechen. Der Konstruktor erhält als Parameter den Namen – er verweist auf eine Klasse – sowie optional eine Instanz der Klasse »Locale«. Die »getString(key)«-Methode holt dann den zugehörigen Wert.

Für die Darstellung von Meldungen, Zahlen und Daten sind die unterschiedlichen Format-Klassen im Package »java.text« zuständig, vor allem die Klassen »DateFormat«, »MessageFormat« und »NumberFormat«. Die »Collator«-Klasse – ebenfalls in »java.text« zu finden – sortiert Strings gemäß den lokalen Gegebenheiten, was insbesondere in Sprachen mit Sonderzeichen wie im Deutschen wichtig ist.

## Ein Beispiel

Die Internationalisierungskonzepte lassen sich am besten an einem kleinen Beispiel erklären, das die wesentlichen Klassen nutzt. Die in **Listing 1** abgedruckte I18n-fähige Klasse hat vier Ausgabemethoden, die die einzelnen oben genannten Aspekte demonstrieren. Die »main«-Methode (Zeilen 65 bis 78) ruft alle vier Methoden zweimal auf, einmal für das Default-Locale und einmal spezifisch für das amerikanische Locale. Das Default-Locale setzt der Konstruktor in der Zeile 44. Die interne »setLocale()«-Methode speichert das Locale ab und erzeugt auch die richtige Instanz des Resource Bundle.

Die »printStrings()«-Methode zeigt den Zugriff auf feste Strings über ein Resource Bundle. Typischer Anwendungsfall sind Beschriftungen in GUIs, etwa ein Button mit „Search“ beziehungsweise „Suche“. Eingebürgerte englische Begriffe wie „OK“ bedürfen natürlich keiner Übersetzung.

Strings mit variablen Bestandteilen könnte man auch über diese Methode verarbeiten, hat dabei aber das Problem, dass die Satzstellung und der Ort der variablen Größe je nach Sprache unterschiedlich sein können. Hierfür ist die »MessageFormat«-Klasse besser geeignet (siehe Zeilen 109 bis 116). Sie verwendet ein Template mit Variablen (Zeile 2 in den **Listings 2 und 3**). Die Instanz der »MessageFormat«-Klasse konfiguriert man mit dem gewünschten Locale, setzt das Template und übergibt der »format()«-Methode einen Array mit Objekten, die den Typen der Variablen im Template entsprechen.

Natürlich ist dies aufwändig und kostet Performance. Deshalb sollten wirklich nur Meldungen an den Endbenutzer lokalisiert werden. Für ein Kommunikationsprotokoll zwischen Client und Server ist es in der Regel genauso unnötig wie für die Protokollierung in Logfiles. Ein weiteres Problem ist der Umgang mit Texten, die – je nachdem, ob die variablen Größen im Singular oder Plural auftreten – unterschiedlich sind. Im Beispiel ist dies nicht berücksichtigt, um auch hier sauber zu sein, ist es erforderlich, ein »ChoiceFormat« zu verwenden.

## Datum und Zahl

Daten und Zahlenwerte ausgeben ist hingegen recht einfach. Die »DateFormat«- und »NumberFormat«-Klassen verfügen über statische Factory-Methoden, die eine korrekt lokalisierte Instanz zurückliefern (Zeilen 96 bis 101 und 124 bis 129). Im Normalfall, also ohne Nutzung eines expliziten Locale, sind auch die Factory-Methoden ohne Argumente anwendbar.

Die Ausgabe des Beispielprogramms mit verschiedenen Locales zeigt die **Abbildung 1**. Das Default-Locale kann man unter Linux entweder über die Umge-

**Listing 1: »I18nBsp.java«**

```
022 import java.util.*;
023 import java.text.*;
024
032 public class I18nBsp{
033
034   private Locale iLocale;
035   private ResourceBundle iBundle;
036
037   ////////////////////////////////////////
038
043   public I18nBsp() {
044     setLocale(Locale.getDefault());
045   }
046
047   ////////////////////////////////////////
048
053   private void setLocale(Locale locale) {
054     iLocale = locale;
055     System.out.println("\n--- Locale: " +
    iLocale.toString() + "---");
056     iBundle = ResourceBundle.getBundle
    ("I18nBspMsgs",iLocale);
057   }
058
059   ////////////////////////////////////////
060
065   public static void main(String[] args) {
066     I18nBsp bsp = new I18nBsp();
067
068     bsp.printStrings();
069     bsp.printDate();
070     bsp.printMessage();
071     bsp.printCurrency();
072
073     bsp.setLocale(new Locale("en","US"));
074     bsp.printStrings();
075     bsp.printDate();
076     bsp.printMessage();
077     bsp.printCurrency();
078   }
079
080   ////////////////////////////////////////
081
086   private void printStrings() {
087     System.out.println("\tString:   " +
    iBundle.getString("hello"));
088   }
089
090   ////////////////////////////////////////
091
096   private void printDate() {
097     Date now = new Date();
098     DateFormat df = DateFormat.
099       getDateInstance(DateFormat.DEFAULT,
    iLocale);
100     System.out.println("\tDate:     " +
    df.format(now));
101   }
102
103   ////////////////////////////////////////
104
109   private void printMessage() {
110     String template =
    iBundle.getString("dirInfo");
111     MessageFormat mf = new
    MessageFormat("");
112     mf.setLocale(iLocale);
113     mf.applyPattern(template);
114     System.out.println("\tMessage:  " +
115       mf.format(new Object[] {new
    Integer(5)}));
116   }
117
118   ////////////////////////////////////////
119
124   private void printCurrency() {
125     Double value = new Double(12345678.99);
126     NumberFormat cf = NumberFormat.
127       getCurrencyInstance(iLocale);
128     System.out.println("\tCurrency: " +
    cf.format(value));
129   }
130 }
```

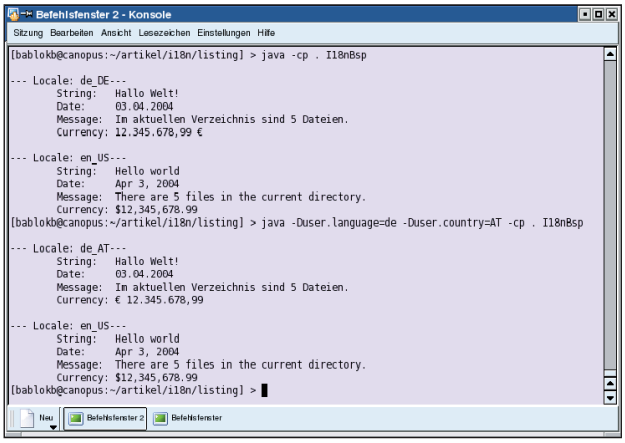


**Abbildung 1:** Ausgabe des Beispielprogramms mit verschiedenen Lokalisierungen.

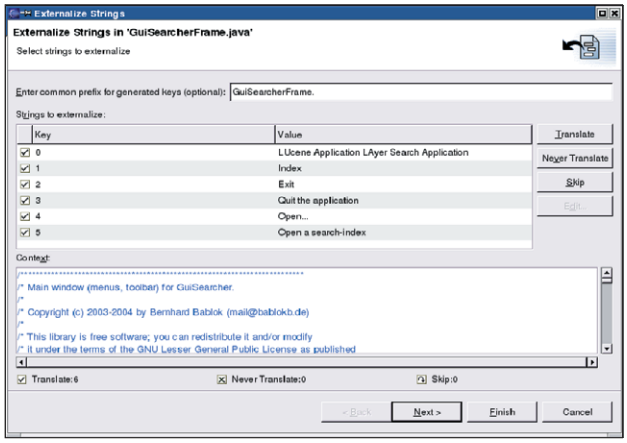


**Abbildung 2:** Der Eclipse-Wizard hilft beim nachträglichen Internationalisieren.

bungsvariable »LANG« setzen oder – wie im Screenshot zu sehen – über entsprechende Properties. Wie man sieht, unterscheiden sich die Österreicher von den Deutschen nur darin, dass sie das Euro-Zeichen vor den Betrag setzen. Bei den Schweizern sieht es wieder etwas anders aus.

## Die Magie dahinter

Unklar ist bisher noch, wie Java die richtigen Texte findet. In Zeile 56 (Listing 1) instanziert die Klasse ein Resource Bundle mit dem Namen »I18nBspMsgs«. Die »getString()«-Methode sucht jetzt bei einem deutschen Default-Local die folgenden Klassen:

```
I18nBspMsgs_de_DE.class
I18nBspMsgs_de.class
I18nBspMsgs.class
```

Bei dem Locale »en-UK« sind es die Klassen:

```
I18nBspMsgs_en_UK.class
I18nBspMsgs_en.class
I18nBspMsgs.class
```

Für Strings reicht es aus, Java-Properties-Dateien mit demselben Namen vorzuhalten. Wer auch andere Objekte, beispielsweise Icons, lokalisieren möchte, muss aber entsprechende Klassen implementieren.

Die Listings 2 und 3 zeigen die beiden Dateien »I18nBspMsgs.properties« und »I18nBspMsgs_en_US.properties«. Hier dient die deutsche Version als Fallback-Alternative, das ist jedoch willkürlich gewählt und im Allgemeinen unüblich. Das Beispiel [3] funktioniert nur, wenn die Properties-Dateien mit den lokalisierten Texten im aktuellen Verzeichnis liegen – für produktive Programme keine akzeptable Lösung. Typischerweise sind die Dateien in Jar-Archiven verpackt.

## Verpackung in Jars

Dabei sind mehrere Szenarien denkbar. Bei kleineren Anwendungen kann es durchaus sinnvoll sein, alle Sprachvarianten in das einzige Applikationsarchiv zu integrieren. Kommt dann eine weitere Sprache hinzu, ist eine neue Version zu bauen. Bei Open-Source-Projekten ist dies sicherlich die unkomplizierteste Lösung des Problems.

Alternativ dazu kann ein Installationsprogramm die gewünschte Sprachversion abfragen und die entsprechende Jar-Datei installieren. Auch mehrere Sprachversionen parallel sind damit möglich. Im letzteren Fall müssen aber alle Sprach-Jars im »CLASSPATH« liegen. Das erste Argument zu »getBundle()« sollte ein voll qualifizierter Klassenname sein. Dies stellt sicher, dass die Runtime die Ressourcen innerhalb der Jar-Archive auch findet. Aus Kompatibilitätsgründen zu früheren Versionen sind jedoch auch Pfadangaben mit »/« statt mit ».« erlaubt.

## Nacharbeiten mit Eclipse

Wie eingangs erwähnt ist es durchaus weit verbreitet, Programme nachträglich zu internationalisieren. Zum Glück gibt es dafür Unterstützung durch Tools. In der freien IDE Eclipse stößt der Menüpunkt »Source | Externalize Strings« einen Wizard an, der die meiste Arbeit erledigt (Abbildung 2). Wenn im Package-Explorer ein Verzeichnis beziehungsweise ein Package ausgewählt ist, hilft der Menüpunkt »Source | Find Strings to Externalize« weiter. Über einen Auswahldialog lassen sich dann alle Quelldateien bearbeiten.

## finally{}

Java-Programme sind einfach zu internationalisieren. Steht die Infrastruktur, genügt es, für neue Sprachen und Regionen nur einfache Textdateien zu übersetzen. Die Erfahrung zeigt, dass dann die Hemmschwelle zur Mitarbeit gerade bei Open-Source-Projekten sehr niedrig ist. So können auch Nur-Nutzer von Programmen sich an deren Weiterentwicklung beteiligen. *(uwo)* ■

### Infos

[1] ISO-Sprachcodes: [http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt]
[2] ISO-Ländercodes: [http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html]
[3] Listings dieses Coffee-Shops: [ftp.linux-magazin.de/pub/listings/2004/07]

**Listing 2: »I18nBspMsgs.properties«**

```
1 hello = Hallo Welt!
2 dirInfo = Im aktuellen Verzeichnis sind {0,number,
  integer} Dateien.
```

**Listing 3: »I18nBspMsgs_en_US.properties«**

```
1 hello = Hello world
2 dirInfo = There are {0,number,integer} files in the
  current directory.
```