1 Java i internacionalizacija (I18n)

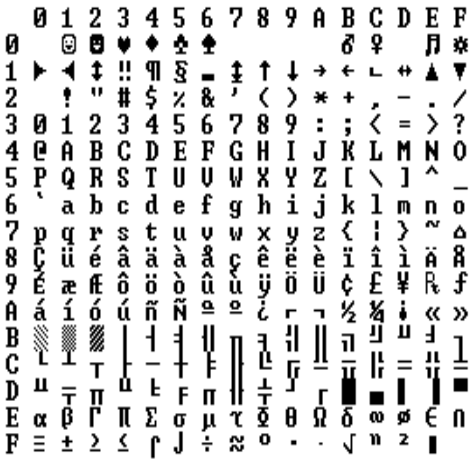
Datumi, vremena, valute, čak i decimalni brojevi, različito se formatiraju i prikazuju u različitim delovima sveta. Zbog toga je potreban jednostavan način za podešavanje menija, natpisa na dugmadima, poruka o greškama i prečica sa tastature u različitim jezicima. Promene u interfejsu aplikacije projektovane tako da podržava internacionalizaciju treba da budu pokrenute na osnovu informacija koje ona dobija od računara na kome se izvršava. Javin SDK nudi alate za internacionalizaciju aplikacija u smislu lokalizovanja prikaza datuma i vremena, brojeva i teksta, grafičkih interfejsa.

Java je prvi programski jezik od početka projektovan tako da podrži internacionalizaciju. To je bilo moguće zato što podržava Unicode u radu sa stringovima. Da bi se razumela internacionalizacija, mora se prvo razumeti šta je to Unicode.

1.1 Unicode i kodiranje znakova

U doba kada su se pojavili UNIX i jezik C, prikaz znakova je bio vrlo jednostavan. U računarstvu je bila bitna samo engleska abeceda, za koju je postojao ASCII kod, te se svaki znak engleske abecede predstavljao brojem između 32 i 127. Tako je kod za razmak bio 32, za slovo *A* 65 itd. Postojeći kodovi znakova mogli su se bez problema smestiti u 7 bitova. Pošto je većina tadašnjih računara koristila osmobitne reči, ne samo da se u njih mogao smestiti čitav znak, već je postojao još i bit viška, koji se mogao upotrebiti za razne svrhe. Kodovi ispod 32 korišćeni su za tzv. kontrolne znake (npr. kod 12 je služio za izbacivanje papira iz štampača i uzimanje novog, a 13 za prelazak u novi red). Ovaj sistem je funkcionisao pod uslovom da se koristio engleski jezik.

Pošto u bajtu ima mesta za osam bitova, mnogi su došli na ideju da slobodne kodove od 128 do 255 koriste u sopstvene svrhe. Problem je nastao zato što je mnogo ljudi došlo tu ideju istovremeno. IBM-PC je koristio tzv. OEM skup znakova u kome se našlo mesta za određena akcentovana slova iz evropskih jezika i znakove za crtanje linija:



Ove linije mogle su se koristiti za iscrtavanje okvira i crta na ekranu računara. Čim su PC računari osvojili tržište izvan SAD, počeli su da se pojavljuju različiti skupovi OEM znakova, u kojima se gornjih 128 znakova koristilo za različite namene. Često se dešavalo da u nekim jezicima postoji više varijanti korišćenja gornjih 128 kodova; npr. dokumenti na ruskom jeziku nisu mogli pouzdano da se razmenjuju.

Da bi se rešio problem različitog kodiranja viših 128 kodova, OEM je standardizovan. Po tom standardu, u donjih 128 kodova zadržan je ASCII kod, ali je u zavisnosti od lokacije standardizovano šta će se raditi sa gornjih 128. Ti standardizovani kodovi nazvani su kodne strane (code page). Tako je npr. za ćirilicu korišćena kodna strana 855, a za našu latinicu 852. U kodnim stranama svi znaci izvan engleske abecede kodirani su vrednostima većim od 128. Međutim, bilo je potpuno nemoguće na istom računaru prikazivati npr. ćirilični i latinični tekst, jer su korišćene posebne kodne strane sa različitim tumačenjima viših kodova.

U Aziji je stanje sa kodiranjem bilo još haotičnije jer azijski jezici imaju na hiljade znakova koji nikako ne mogu da se smeste u osam bitova. Azijski problem se rešavao konfuznim sistemom nazvanim DBCS (double byte character set) u kome je za kodiranje nekih znakova korišćen jedan bajt, a za druge dva. String se lako tumačio unapred, ali nikako unazad. Programerima se preporučivalo da ne koriste naredbe s++ i s– za kretanje kroz stringove, već da umesto toga pozivaju Windowsove funkcije AnsiNext i AnsiPrev koje su nekako izlazile na kraj sa celom zbrkom.

Ipak, većina programera se i dalje pretvarala da je jedan bajt znak, i dogod se nije pojavila potreba da se string prenosi sa jednog računara na drugi, ili da program "progovori" nekim drugim jezikom, takav pristup se mogao tolerisati. Međutim, kada se pojavio Internet, prenošenje stringova postalo je svakodnevica, a problemi sa kodiranjem znakova više se nisu mogli olako zanemarivati.

Standard Unicode uveden je sa ciljem da se ustanovi jedinstven skup znakova koji će podržavati sve postojeće sisteme pisanja. Često se može čuti pogrešno

tumačenje da je Unicode 16-bitni kod za znakove, tj. način predstavljanja znakova u kome svaki znak zauzima dva bajta, pa prema tome Unicode može da kodira najviše 65.536 znakova. To nije tačno.

Dosad smo pretpostavljali da znak odgovara nekoj kombinaciji bitova koji se smeštaju u memoriju ili na disk, npr. u ASCII kodu slovo A se čuvalo kao bajt 0100 0001. U standardu Unicode svaki znak ima sopstvenu kodnu tačku (code point). Kodna tačka je u suštini teorijski pojam, jer Unicode ne kaže ništa o tome kako će ona biti sačuvana u memoriji ili na disku. Kodna tačka se piše ovako: U+0639, gde U+ znači "Unicode", a brojevi su heksadecimalni. Kodna tačka slova *A* je U+0041, a našeg slova *Č* je U+014C. Unicode bilo kog znaka može da se pronađe pomoću Windowsovog programa charmap ili na adresi www.unicode.org. Ne postoji nikakvo ograničenje broja znakova koje Unicode može da predstavi; broj znakova koji su već predstavljeni odavno je prešao 65.536, što znači da ne može svaki Unicode znak da se kodira sa dva bajta, kao što se često pogrešno veruje.

Na primer, string *Java* bi u Unicode standardu odgovarao sledećoj kombinaciji kodnih tačaka:

U+004A U+0061 U+0076 U+0061

Za čuvanje Unicode kodnih tačaka koriste se različiti sistemi kodiranja. Prva ideja za Unicode kodiranje, koja je i dovela do uobičajene zablude o dva bajta za Unicode, bila je da se brojevi u kodnim tačkama predstave sa po dva bajta. Tako bi string *Java* postao:

00 4A 00 61 00 76 00 61 00

Međutim, isti string mogao bi da se predstavi i kao:

4A 00 61 00 76 00 61 00

Tehnički, oba pristupa su izvodljiva, pa se u ranim implementacijama Unicode standarda moglo birati da li će Unicode kodne tačke biti predstavljene na high-endian ili low-endian način[1], u zavisnosti od toga koji je bio brži na konkretnom procesoru. Tako su se odmah pojavila dva načina da se sačuva Unicode, što nije bilo od koristi u rešavanju opšte zbrke sa kodiranjima. Da bi se ova dva pristupa Unicode kodiranju razlikovala, uveden je tzv. Unicode Byte Order Mark, tj. oznaka metoda prikaza (FE FF za little-endian, odnosno FF FE za big-endian). Izvesno vreme činilo se da bi to moglo da bude prihvatljivo, ali su programeri, a naročito oni čiji je maternji jezik engleski, počeli da se žale. Naročito su im smetale brojne nule u kodu engleskog teksta, u kome se retko koriste kodne tačke iznad U+00FF. Zbog toga je većina programera rešila da zanemari Unicode nekoliko godina, a u međuvremenu se stanje dodatno pogoršavalo.

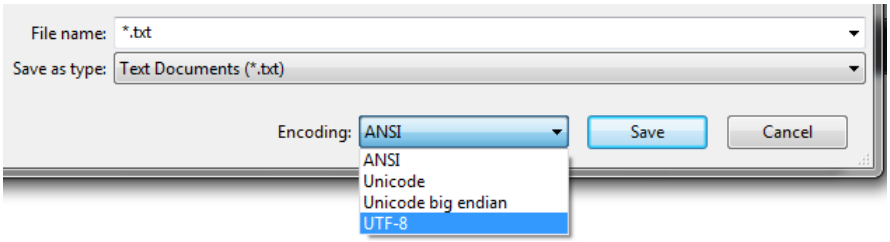
Rešenje problema predstavljanja Unicode kodnih tačaka dobijeno je uvođenjem kodiranja UTF-8. U sistemu UTF-8, sve kodne tačke od 0 do 127 čuvaju se kao jedan bajt, dok se kodne tačke iznad 128 čuvaju u 2, 3, ili čak 6 bajtova.

[1] Pretpostavimo da radimo sa dva bajta; oni se u memoriji mogu sačuvati tako da prvi bajt sadrži bitove veće težine (to je tzv. big-endian metod), ili bitove manje težine (tzv. little-endian method). Različiti procesori koriste različit pristup; npr. SPARC koristi big-endian, a Pentium little-endian metod. U Javi se sve vrednosti čuvaju kao big-endian, nezavisno od procesora, što čini Javine fajlove platformski nezavisnim.

```
 Hex Min  Hex Max  Byte Sequence in Binary
00000000 0000007F 0vvvvvvv
00000080 000007FF 110vvvvv 10vvvvvv
00000800 0000FFFF 1110vvvv 10vvvvvv 10vvvvvv
00010000 001FFFFF 11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
00200000 03FFFFFF 111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
04000000 7FFFFFFF 1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
```

Koristan sporedni efekat ovog sistema jeste da tekst na engleskom kodiran kao UTF-8 izgleda isto kao u ASCII kodu, pa se u engleskom razlika između ova dva kodiranja ne primećuje. Na primer, string *Java*, čije su kodne tačke U+004A U+0061 U+0076 U+0061, biće sačuvan kao 4A 61 76 61 (po jedan bajt za svako slovo), što je isto kao u ASCII, ANSI, ili bilo kom postojećem OEM kodiranju. Međutim, ako upotrebite srpsku azbuku, biće potrebna po dva bajta za predstavljanje jedne kodne tačke (to je i razlog zbog koga ako npr. u SMS poruci koristite "naša" slova, u nju staje mnogo manje od uobičajenih 160 znakova).

Dosad smo već pomenuli tri načina za Unicode kodiranje. Standardni načini čuvanja u obliku dva bajta zovu se UTF-16 (zato što imaju 16 bitova) ili prosto Unicode, ali i dalje treba znati da li je reč o high-endian ili low-endian rasporedu. Treći je popularan standard UTF-8 koji ima tu srećnu okolnost da radi neprimetno i u slučaju engleskog teksta i starih programa koji nemaju pojma da na svetu postoji još nešto osim ASCII koda. Postoji još nekoliko manje rasprostranjenih načina za Unicode kodiranje (npr. UTF-7 ili UTF-32) kojima se zbog ograničenog prostora i značaja ovde nećemo baviti.



Unicode kodne tačke mogu se predstaviti i u bilo kom postojećem sistemu kodiranja znakova. Tako bi se Unicode string mogao kodirati ASCII, OEM srpskim ili hebrejskim ANSI kodiranjem, doduše sa jednim nedostatkom: neki znakovi možda se ne bi prikazali. Ako u kodiranju koje se koristi ne postoji ekvivalent za Unicode kodnu tačku, obično se umesto tog znaka pojavljuje znak pitanja ili romb. Postoje na stotine uobičajenih kodiranja koja ispravno kodiraju samo neke kodne tačke, a sve druge prikazuju kao znakove pitanja. Za tekstove na engleskom popularno kodiranje je ISO-8859-1 (Latin-1 ili ANSI). Pokušate li međutim da kodirate srpski ili grčki u tim kodiranjima, dobićete mnoštvo znakova pitanja. Sva UTF kodiranja (8, 16 i 32) imaju dobru osobinu da sve Unicode kodne tačke čuvaju ispravno.

U svetlu svega onoga što je dosad rečeno o kodiranjima, treba imati u vidu da je string za koji ne znamo kako je kodiran u suštini beskoristan. Više se ne možemo pretvarati da je to samo običan ASCII kod. Da bi se neki string pravilno

protumačio i prikazao, neophodno je znati kako je kodiran: ako programer ne zna da li je korišćeno kodiranje UTF-8, ASCII ili ISO 8859-1, nije u stanju da ispravno prikaže string niti da zaključi gde se završava.

Postoje standardni načini za čuvanje informacije o kodiranju određenog stringa. Na primer, u zaglavljima e-poruka treba da se nađe string:

Content-Type: text/plain; charset="UTF-8"

Na Web stranama to je meta tag:

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

Ovaj tag treba da bude prvi u odeljku <head> jer čim čitač Weba naiđe na njega, prestaje da tumači stranu i započinje ponovo da je parsira koristeći naznačeno kodiranje. Ako ne naiđe na oznaku Content-Type, čitač Weba pokušava da "pogodi" kodiranje strane u zavisnosti od učestanosti pojavljivanja određenih bajtova u tipičnom tekstu u uobičajenim kodiranjima različitih jezika. Ovaj pristup je toliko uspešan da čak i strane za koje nije naveden tag content-type obično izgledaju sasvim dobro.

1.2 Lokalne oznake (lokaliteti)

Da bi se neka aplikacija zaista prilagodila za međunarodno tržište, nije dovoljno prevesti samo natpise u korisničkom interfejsu; postoje mnoge druge razlike, npr. u formatiranju brojeva ili datuma. Na primer, broj u engleskom prikazu 123,456.78 u nemačkom jeziku se formatira kao 123.456,78, tj. decimalni zarez i tačka u ova dva jezika imaju obrnuto značenje. Isti je slučaj i sa datumima; npr. englesko formatiranje datuma 3/22/61 na nemačkom je 22.03.1961. Ponekad se čak dešava da je u dve zemlje u kojima se govori istim jezikom (npr. SAD i Velika Britanija) potrebno načiniti izvesne izmene u prikazu da bi korisnici iz obe zemlje bili zadovoljni.

Za kontrolu formatiranja koristi se klasa `java.util.Locale`. Ona opisuje:

- jezik
- lokaciju (opciono)
- jezičku varijantu (opciono)

Na primer, za SAD treba koristiti lokalitet sa podešavanjima:

language=English, location=US

U Nemačkoj se koristi lokalitet sa vrednostima:

language=German, location=GERMANY

Švajcarska ima četiri zvanična jezika (nemački, francuski, italijanski i retoromanski). Švajcarac iz nemačkog govornog područja bi koristio:

language=German, location=SWITZERLAND

Ovaj lokalitet priprema ispis na isti način kao nemački, ali se valuta ispisuje u švajcarskim francima, a ne u evrima. Ako se navede samo jezik, na primer:

language=German

onda lokalitet ne registruje razlike u valutama zemalja. Java koristi kodove jezika i mesta koje je definisala Međunarodna organizacija za standardizaciju;

jezik se izražava kodom od dva mala slova, a kod zemlje sa dva velika slova. Potpuna lista kodova jezika nalazi se na adresi http://www.loc.gov/standards/iso639-2/php/code_list.php, a lista kodova država na adresi

http://www.iso.org/iso/country_codes/iso_3166_code_lists.htm.

Da bi se definisao lokalitet, u konstruktoru klase `Locale` zadaju se jezik, kod zemlje i varijanta jezika (ako postoji):

```
Locale nemacki = new Locale("de");
Locale nemackiNemacki = new Locale("de", "DE");
Locale svajcarskiNemacki = new Locale("de", "CH");
```

Java nudi brojne predefinisane objekte lokaliteta, npr.

```
Locale.CHINA
Locale.GERMANY
Locale.UK
Locale.US
```

Na raspolaganju su i predefinisani objekti lokaliteta koji određuju jezik bez obzira na lokaciju:

```
Locale.ENGLISH
Locale.GERMAN
Locale.SIMPLIFIED_CHINESE
```

Osim konstruisanja novog objekta `Locale` ili korišćenja nekog predefinisanog, postoje još dva načina za dobijanje objekta lokaliteta. Statički metod `Locale.getDefault()` vraća podrazumevanu vrednost lokaliteta u zavisnosti od podešavanja operativnog sistema. Pozivom metode `Locale.setDefault()` ta vrednost se može izmeniti, ali se promena odražava samo na program, a ne i na operativni sistem. Takođe, svi alati koji zavise od lokaliteta mogu da vrate niz lokaliteta koji podržavaju, npr. sledeća funcija vraća sve lokalitete koje klasa `DateFormat` podržava:

```
Locale[] podrzaniLokaliteti = DateFormat.getAvailableLocales();
```

Objekat klase `Locale` sam po sebi nije naročito koristan, ali služi za prosleđivanje metodama koje rade sa tekstom koji se prikazuje korisnicima na različitim mestima, kao što ćemo videti u narednim primerima.

1.3 Kodiranje znakova i Java

Često se dešava da operativni sistemi i čitači ne podržavaju Unicode. Zato postoji sloj za prevođenje skupa znakova i fontova između računara i Javine virtuelne mašine koja radi na njoj. Primitivni tip u `char` u Javi koristi big-endian UTF-16 kodiranje za predstavljanje kodnih tačaka iz opsega U+0000 do U+FFFF. Svaka instanca Javine virtuelne mašine koristi podrazumevano kodiranje znakova koje se određuje tokom njenog pokretanja, a zavisi od lokaliteta i kodiranja operativnog sistema na kome se virtuelna mašina izvršava. Trenutno u Javi nema nikakve veze između lokaliteta, tj. klase `Locale` i kodiranja znakova (npr. ako se odabere kineski lokalitet, ne postoji Javin metod koji će reći da treba da se upotrebi kodiranje Big5).

Java nudi skup filtara za tokove koji premošćuju jaz između Unicode teksta i kodiranja koje koristi lokalni operativni sistem. Sve te klase izvedene su iz apstraktnih klasa `Reader` i `Writer`; na primer, klasa `InputStreamReader` pretvara ulazni tok sa bajtovima u određenom kodiranju u čitač sa Unicode znakovima. Slično, klasa `OutputStreamWriter` pretvara niz Unicode znakova u tok bajtova određenog kodiranja. Podrazumevana vrednost kodiranja u Windowsu je ISO 8859-1 (ANSI), ali se različito kodiranje može zadati u konstruktoru, npr.

```
InputStreamReader in = new InputStreamReader(
        new FileInputStream("file.dat"), "UTF8");
```

Na isti način, kada se podaci smeštaju u tekstualnu datoteku, treba uzeti u obzir lokalno kodiranje znakova tako da korisnici mogu da je otvore u svojim aplikacijama. U konstruktoru klase `FileWriter` treba navesti zapis u željenom kodu:

```
out = new FileWriter(filename, "ISO-8859-1");
```

Spisak svih šema kodiranja koje Java podržava može se pronaći na adresi http://docs.oracle.com/javase/1.3/docs/guide/intl/encoding.doc.html

1.4 Lokalizacija brojeva i valuta

Način na koji se formatiraju decimalni brojevi, tj. njihova lokalizacija zavisi od zemlje i jezika. Javin paket `java.text` sadrži klase koje obrađuju brojne vrednosti. Da bi se za određeni lokalitet pripremili brojevi za ispis, potrebno je izvršiti sledeće korake:

1. doći do objekta lokaliteta, na neki od načina opisanih u odeljku 1.2
2. upotrebiti metode faktorisanja klase `NumberFormat` da bi se dobio objekat za formatiranje
3. iskoristiti taj objekat za prikaz i obradu brojnih vrednosti

Metode faktorisanja su tri statičke metode klase `NumberFormat` sa argumentom `Locale`[2]:

```
NumberFormat.getNumberInstance(Locale)
NumberFormat.getCurrencyInstance(Locale)
NumberFormat.getPercentInstance(Locale)
```

Ove metode vraćaju objekte tipa `NumberFormat` koji mogu da formatiraju brojeve, valutu i procente, redom. Sledeći primer ilustruje ispis nemačke valute:

```
Locale lok = new Locale("de", "DE");
NumberFormat valuta = NumberFormat.getCurrencyInstance(lok);
Double iznos = 123456.78;
System.out.println(valuta.format(iznos)); //ispisuje 123.456,78
DM
```

Da bi se učitao broj zadat ili unet korišćenjem određenog lokaliteta, koriste se metode za obradu klase `NumberFormat`. Sledeći primeru ilustruje kako se lokalizuje brojna vrednost koju korisnik unosi u grafičkom interfejsu, preko polja za tekst, a uz oslanjanje na podrazumevani lokalitet:

[2]Postoje i verzije metoda bez argumenta; one rade sa podrazumevanim lokalitetom.

```
TextField inputField;
NumberFormat format = NumberFormat.getNumberInstance();
Number input = format.parse(inputField.getText().trim());
double x = input.doubleValue();
```

Povratni tip metode `parse()` je apstraktna klasa `Number`; da bi se od nje stiglo do prostog tipa `double`, treba upotrebiti metod `doubleValue()`. Ako tekst iz koga se parsira broj nije u odgovarajućem formatu, npr. na početku sadrži razmake, metod `parse()` baca izuzetak tipa `ParseException`. Da bi se to izbeglo, treba pozvati metod `trim()` za stringove. Znakovi koji slede iza broja u stringu se zanemaruju i u tom slučaju nema izuzetka.

Za formatiranje vrednosti valute može da se koristi statički metod `NumberFormat.getCurrencyInstance()`. Međutim, taj metod je prilično nefleksibilan, jer vraća formater za samo jednu valutu. Pretpostavimo da pripremate izveštaj u kome neki iznosi treba da budu prikazani u dolarima, a drugi u evrima. Kada biste koristili dva formatera:

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

izveštaj bi izgledao čudno jer bi neke vrednosti bile prikazane kao $100,000 a druge kao 100.000€ (obratite pažnju da evro koristi decimalnu tačku, a ne zarez, i da se pojavljuje iza cifre). Za takve slučajeve bolje je koristiti statički metod `Currency.getInstance()`, kao u sledećem primeru:

```
NumberFormat.euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);

euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Identifikatori valuta su definisani standardom ISO 4217.

1.5 Lokalizacija datuma i vremena

Format prikaza datuma i vremena razlikuje se od jezika do jezika. Osim redosleda dana, meseca i godine, treba voditi računa i o prikazu meseca u odgovarajućem jeziku i o vremenskim zonama. Način na koji se ovo radi veoma je sličan korišćenju klase `NumberFormat` u sprezi sa objektom `Locale` što je opisano u prethodnom odeljku, samo što se u ovom slučaju koristi klasa `DateFormat`. Dakle, prvo je potrebno dobiti instancu lokaliteta, koja se zatim prosleđuje metodama faktorisanja klase `DateFormat`:

```
DateFormat.getDateInstance(dateStyle, locale)
DateFormat.getTimeInstance(timeStyle, locale)
DateFormat.getDateTimeInstance(dateStyle, timeStyle, locale)
```

Sve metode imaju parametar stila koji može biti jedna od sledećih konstanti:

`DateFormat.DEFAULT`

`DateFormat.FULL` (Wednesday, September 15, 2004 8:51:03 PM PDT za US lokalitet)

`DateFormat.LONG` (September 15, 2004 8:51:03 PM PDT za US lokalitet)

`DateFormat.MEDIUM` (Sep 15, 2004 8:51:03 PM za US lokalitet)

`DateFormat.SHORT` (9/15/04 8:51 PM za US lokalitet)

Da bi se datum formatirao za trenutni lokalitet, može se upotrebiti statički metod:

```
String datum = DateFormat.getDateInstance().format(myDate);
```

Da bi se datum formatirao za različit lokalitet, on se zadaje u pozivu metode `getDateInstance()`:

```
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);
```

Klasa `DateFormat` može se koristiti i za parsiranje datuma iz stringa:

```
Date datum = df.parse(mojString);
```

1.6 Internacionalizacija teksta

U svakoj aplikaciji koja se internacionalizuje postojaće veliki broj poruka, dugmadi i sl. čije natpise treba prevesti na različite jezike. Da bi se to obavilo na najbolji način, potrebno je definisati stringove poruka na spoljašnjim lokacijama koje se zovu resursi. Na taj način poruke će moći da se prevode bez menjanja izvornog koda aplikacije.

U Javi treba koristiti datoteke svojstava (tj. datoteke sa ekstenzijom .properties) u koje se smeštaju stringovi i odgovarajući prevodi. Svi natpisi koje treba internacionalizovati smeštaju se u datoteku svojstava, npr. *stringovi.properties*. To je jednostavna tekstualna datoteka sa po jednim parom ključ/vrednost po redu.

Lokalizacija počinje kreiranjem skupa resursa (resource bundle). Svaki skup se sastoji od datoteke svojstava (property file) koja opisuje lokalizovane stavke (poruke, natpise i sl). Potrebno je obezbediti verzije za sve lokale koje treba podržati. Prilikom kreiranja resursa koristi se specijalno imenovanje. Na primer, resursi određeni za Italiju trebalo bi da se nađu u datoteci *stringovi_it_IT*, dok se oni za sve zemlje u kojima se govori italijanski (npr. za Švajcarsku) smeštaju u datoteku *stringovi_it*. Po pravilu se koristi šema imenovanja *imeDatoteke_jezik_zemlja* za sve resurse specifične za zemlju, odnosno *imeDatoteke_jezik* za sve resurse specifične za jezik.

Skup resursa učitava se naredbom:

```
ResourceBundle tekuciResursi = ResourceBundle.getBundle(imeDatoteke,
lokalitet);
```

Metod `getBundle()` pokušava da učita resurs koji odgovara tekućem lokalitetu po jeziku, zemlji i varijanti jezika. Ako u tome ne uspe, iz pretrage izbacuje varijantu jezika, zemlju, i na kraju jezik. Isti princip se primenjuje i u traženju podrazumevanog lokaliteta; ako pretraga ne nađe ništa, konsultuje se podrazumevana datoteka sa resursima. Ako se ni ona ne nađe, metod baca izuzetak `MissingResourceException`.

Kada metod `getBundle()` pronađe resurs, recimo, *imeDatoteke_it_IT*, i dalje će nastaviti da traži *imeDatoteke_it* i *imeDatoteke*. Ako ti resursi postoje, oni će postati roditeljski resursi za *imeDatoteke_it_IT* u hijerarhiji resursa. Kasnije, prilikom traženja resursa, roditelji će se tražiti ako pretraga tekućeg resursa ne uspe. Odnosno, ako se određeni resurs ne nađe u *imeDatoteke_it_IT*, onda će se tražiti i u resursima *imeDatoteke_it* i *imeDatoteke*.

Tipična datoteka svojstava za meni u italijanskom lokalitetu, tj. datoteka *menu_it.properties* mogla bi da izgleda ovako:

fileMenu=File
editMenu=Modifica
viewMenu=Visualizza
toolsMenu=Strumenti
helpMenu=Aiuto

Zatim se datoteka svojstava imenuje na način opisan u prethodnom odeljku, npr. *menu.properties*, *menu_it.properties*, *menu_it_CH.properties*. Resurs se učitava naredbom

```
ResourceBundle bundle = ResourceBundle.getBundle("menu", lokalitet);
```

Određeni string se pronalazi pomoću naredbe:

```
String viewMenuLabel = bundle.getString("viewMenu");
```

U mnoga razvojna okruženja za Javu ugrađen je mehanizam za internacionalizaciju; na primer, uputstvo za NetBeans IDE se nalazi na adresi http://netbeans.org/kb/docs/java/gui-automatic-i18n.html.