CODESPRING
OUR CODE YOUR SUCCESS
20 years

Codespring mentorprogram
Enterprise szoftverfejlesztési módszerek (MP II)

Let's win the race together!

Konfiguráció, nemzetköziesítés és lokalizáció

Java SE kiegészítések – 1. rész

Simon Károly
simon.karoly@codespring.ro

Properties

- Általános szabály, hogy a különböző konfigurációs jellemzőket (amelyek változhatnak) ne építsük be a kódunkba (hardcoding), mivel ez minden változás esetén az újrafordítás kényszeréhez vezetne.
- *Példa:* hálózati kliens-szerver alkalmazás esetén a cím vagy port változása ne eredményezze a kód újrafordítását.
- A paramétereket konfigurációs állományokban tároljuk.
- **`Properties`** osztály: egy alkalmazás esetében előforduló perzisztens tulajdonságok kezelésére alkalmas.
- Tulajdonképpen egy hasító tábla alapú szótár implementáció, a **`Hashtable`** osztály leszármazottja. A tulajdonságokat azonosító-érték párok formájában állományokban tárolhatjuk, a következő módon:

```
identifier1=value1
identifier2=value2
```

Properties

- Az állományok tipikusan **`.properties`** kiterjesztésűek. A **`Properties`** osztály megfelelő metódusaival (**`load`** és **`store`**) beolvashatjuk, vagy lementhetjük (módosíthatjuk) ezeket a tulajdonságokat. Ezen kívül az osztály lehetőséget biztosít arra, hogy lekérjünk, vagy módosítsunk egy adott azonosítóval rendelkező tulajdonságot:

```
public String getProperty(String key);
public Object setProperty(String key, String value);
```

- A második metódus az illető azonosítónak megfelelő előző értéket téríti vissza, ha volt ilyen, egyébként **`null`** értéket ad.

Properties


```
public class PropertyProvider {

    private static Properties properties;

    static {
        properties = new Properties();
        try (InputStream is = PropertyProvider.class.getResourceAsStream("/blogconfig.properties")) {
            if (is != null) {
                properties.load(is);
            }
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    public static String getProperty(final String key) {
        return properties.getProperty(key);
    }
}
```

blogconfig.properties tartalma lehet például:

```
encoding=UTF8
default_language=en
default_region=US
```

Megjegyzés: állományok olvasása

- Állományokat olvashatunk egyenesen a fizikai file-rendszerünkről. Ezt tegyük, amikor egy külső felhasználó beadhat állományt, mint bemenet (pl. egy Swing felületen betölthet egy teljes path-et).

```
...
InputStream is = new FileInputStream(new File("pathOnPhysicalMachine"));
// read from is
...
```

- Továbbra olvashatunk állományokat a **JVM file-rendszeréről**. Ezen belül minden **`CLASSPATH`**-en levő mappa/JAR a *gyökér*be kerül. Ezt tegyük, amikor mi szállítjuk a betöltendő állományt az alkalmazásunkkal (pl. a **`Properties`** állományunk).

```
...
InputStream is = CurrentClass.class.getResourceAsStream("/pathOnJvmClasspath");
// read from is
...
```

Komplexebb konfigurációs állományok

- A `Properties` állományok életképesek egyszerű konfigurációk esetén, de növekvő alkalmazások esetén a következő hiányosságok lépnek fel:
  - Karakterláncokon kívüli típusok támogatása–számok, `boolean` vagy `enum`ok
  - Egymásba ágyazott objektumok
  - Validáció (pl. karakterlánc maximális hossza)
  - Alapértelmezett értékek megadása
- Komplexebb leíró nyelvezetekért:
  - *XML* állományokat olvashatunk a beépített **JAXB** (*Java Architecture for XML Binding*) API segítségével
  - A *JSON* formátum a Jakarta EE 8-as verziójától natívan támogatott a **JSON-B** API által
  - Alternatívan használhatunk *külső könyvtárakat*
    - A **Jackson** könyvtár alapból támogatja a *JSON* formátumot, opcionális kiterjesztésekkel *XML* és *YAML*-t is. A Spring alapértelmezetten ezt használja.
    - További példák: Gson, SnakeYAML

JSON konfiguráció példa Jacksonnal

- Komplex konfigurációs mechanizmusok esetén definiálhatunk POJO-kat a szükséges változóknak, majd a konfigurációs állományt azon osztály egy példányába olvassuk.

Példa: `conf-jackson`

- a konfigurációs állomány tartalmazhat különböző típusokat és egymásba ágyazott objektumokat

`src/main/resources/application.json`:

```
{
  "jdbc": {
    "createTables": true,
    "driverClass": "org.h2.Driver",
    "url": "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"
  },
  "connectionPool": {
    "poolSize": 4
  }
}
```

JSON konfiguráció példa Jacksonnal

- Térképezett konfigurációs POJO-k

```
public class MainConfiguration {
    JdbcConfiguration jdbc = new JdbcConfiguration();
    ConnectionPoolConfiguration connectionPool = new ConnectionPoolConfiguration();
    ...
}

public class JdbcConfiguration {
    // alapértelmezett beállítás adható meg
    // a Jackson csak akkor írja fölül, ha a JSON állományban meg van adva
    Boolean createTables = false;
    String driverClass;
    String url;
    ...
}

public class ConnectionPoolConfiguration {
    Integer poolSize = 1;
    ...
}
```

Jackson: `ObjectMapper`

- A Jackson által nyújtott **`ObjectMapper`** osztály képes JSON-ból felépíteni egy megadott osztály példányát. A JSON forrás megadható több különböző formátumban: karakterlánc, állomány, stream, külső URL, stb. Megadható egy konkrét osztály, vagy lekérhető egy általános hasítótábla is.

```
ObjectMapper objectMapper = new ObjectMapper();
MyBeanClass myBean = objectMapper.readValue(myJsonStream, MyBeanClass.class);
```

- A Jackson előbb absztrakt fát épít fel a JSON állományból, melyet reflection segítségével megpróbál a megadott oszály egy példányára alakítani. A második lépést használhatjuk általános objektum-átalakításra is.

```
Map<String, Object> intermediate = objectMapper.readValue(myJsonStream, MyBeanClass.class);
MyBeanClass myBean = objectMapper.convertValue(intermediate, MyBeanClass.class);
```

- Fordítva is képes átalakítani egy POJO-t JSON karakterlánccá.

```
// általános stream kimenet
objectMapper.writeValue(myBean, outputStream);
// string kimenet
String jsonString = writeValueAsString(myBean);
```

- Jackson annotációkkal *opcionálisan* személyre szabhatjuk a (de)szerializáció folyamatát: **`@JsonProperty`**, **`@JsonIgnore`**, **`@JsonDeserializer`**, stb.

JSON konfiguráció példa Jacksonnal

- Jackson függőség szükséges a Gradle állományba:

```
compile group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.10.0'
```

- `ObjectMapper` használata az állomány beolvasására:

```
Object Mapper objectMapper = new ObjectMapper();
InputStream inputStream = ConfigurationFactory.class.getResourceAsStream("/application.json");
MainConfiguration mainConfiguration = objectMapper.readValue(inputStream, MainConfiguration.class);
```

- factory használata az objektum elérhetőségéért:

```
// ... mainConfiguration felépítése
public static JdbcConfiguration getJdbcConfiguration() {
  return mainConfiguration.getJdbcConfiguration();
}
public static ConnectionPoolConfiguration getConnectionPoolConfiguration() {
  return mainConfiguration.getConnectionPoolConfiguration();
}
```

- konfiguráció használata a connection pool felépítésénél:

```
JdbcConfiguration jdbcConfiguration = ConfigurationFactory.getJdbcConfiguration();
Class.forName(jdbcConfiguration.getDriverClass());
Connection connection = DriverManager.getConnection(jdbcConfiguration.getUrl());
```

Gyakorlatok

1. Készítsünk egy `.properties` állományt, amelyben konfiguráljuk a következőket:
   - JDBC driver osztály neve
   - JDBC elérési URL
   - JDBC felhasználónév és jelszó
   - **Otthoni:** A *pool* mérete a korábbi gyakorlathoz.
2. Készítsük el a `PropertyProvider` osztályt, amely `CLASSPATH`-ről olvassa az új állományt, majd használjuk az értékeit a megfelelő helyeken.

2. rész

Profilozás

Profilok

- A fejlesztési ciklus alatt kialakult alkalmazás több környezetben működőképes kell legyen, különböző beállítások segítségével:
  - `development`
    - fejlesztők alkalmazzák az aktív fejlesztői munka közben
    - lokális adatbázis használható, ez lehet egy kisebb erőforrásokat igénylő (pl. in-memory) rendszer
    - nem szükséges egy nagy méretű connection pool, mivel nincs nagy erőforrás-használat
    - kisebb erőforrás-használat a preferált, mivel a fejlesztői számítógépek nem élnek fel a produkciós szerverek hardver-kapacitásával
    - részletes (*debug-szintű*) loggolási beállítás
  - `test` vagy `staging` környezet
    - szimulálja a végső környezetet egy belső infrastruktúrán
    - integrációs tesztekre használható CI rendszerben
    - nem feltétlenül produkciós beállítások–pl. nincsenek `https` certificate-ek, lefoglalt domain név, stb.
  - `production`
    - a végső környezet, ahol a stabilitás a kulcs
    - adatbázis, webszerver és API-szerver potenciálisan különböző hostokon
    - produkciós adatbázis rendszer

Profilok

- Az alkalmazás buildelhető és futtatható kell legyen a különböző profilok szerint.
- A következő aspektusok **változhatnak** profil szerint:
  - A properties (vagy más formátumokban megadott) nem kompilált beállítások–pl. különböző állományokat tölthetünk be a profil beállítás szerint.
  - A függőségek–egyes függőségek erősen hozzáfűződnek a profilhoz.
  - Erőforrás-állományok és source setek
- A *Maven* és a *Spring* natívan támogatja a profilok mintáját.
- Egyszerűen implementálható kihasználva az oprendszerszintű *környezeti változó*kat, vagy Java esetén JVM *rendszerváltozó*kkal vagy Gradle *property*k segítségével.

Profilok példa

Példa: `conf-properties-profiles`

- Szétválasztunk 2 profilt egy Gradle alkalmazásban:
  - **`dev`** profil (*alapértelmezett*):
    - H2 in-memory adatbázis a lokális gépen
    - táblákat létrehozzuk indításkor
    - kisebb connection pool
  - **`prod`** profil:
    - MySQL produkciós adatbázis külső hoston
    - a táblák megfelelő létét feltételezzük (hogy ne törlődjön korábbi adat)
    - nagyobb connection pool
    - **`docker-compose`** segítségével könnyebben használható
- A profilt eldöntjük a Gradle-nek megadott **`profile`** property-vel, aszerint különböző build szkriptet töltünk be:
  - **`gradle run`** vagy **`gradle -Pprofile=dev run`** - dev profil
  - **`gradle -Pprofile=prod run`** - prod profil

Profilok példa

- `build.gradle`:

```
// ... közös beállítások az elején...

// kiválasztjuk a profilt a Gradle projekt property alapján
// alapértelmezetten "dev"
def profile = properties.profile ?: 'dev'
logger.quiet "Detected profile: ${profile}"

// profil-specifikus beállításokat betöltünk a specifikus gradle állományból
apply from: "build-${profile}.gradle"
```

- `build-dev.gradle`:

```
dependencies {
    // h2 in-memory adatbázis
    runtime group: 'com.h2database', name: 'h2', version: '1.4.200'
}
```

Profilok példa

- `build-prod.gradle`:

```
dependencies {
    // mysql connector a h2 helyett
    runtime group: 'mysql', name: 'mysql-connector-java', version: '5.1.48'
}

application {
    // beállítjuk a futáskor használt prod profilt,
    // így a megfelelő application-prod.properties lesz betöltve
    applicationDefaultJvmArgs = ['-Dprofile=prod']

    // a generált indító-szkriptek neve
    applicationName = "${project.name}-dist"
}

distributions {
    main {
        // átállítjuk a distribution nevét,
        // így külön install, zip és tar jön létre a prod profilnak
        baseName = "${project.name}-dist"
    }
}
```

Profilok példa

- src/main/resources/application.properties:

```
jdbc_create_tables=true
jdbc_driver_class=org.h2.Driver
jdbc_db_url=jdbc:h2:mem:test;DB_CLOSE_DELAY=-1
jdbc_pool_size=1
```

- src/main/resources/application-prod.properties:

```
jdbc_create_tables=false
jdbc_driver_class= com.mysql.jdbc.Driver
jdbc_db_url=jdbc:mysql://externalmysqlhost:3306/blog
jdbc_pool_size=8
```

Profilok példa: `PropertyProvider`


```
public class PropertyProvider {
    // ...
    static {
        String propertiesResourceName = buildPropertiesResourceName();
        InputStream inputStream = PropertyProvider.class.getResourceAsStream(propertiesResourceName);
        properties.load(inputStream);
        // ...
    }

    private static String buildPropertiesResourceName() {
        StringBuilder sb = new StringBuilder();
        sb.append('/').append(PROP_FILE_NAME);

        String profile = System.getProperty("profile");
        LOG.info("Determined profile: {}", profile);
        if (profile != null && !profile.isEmpty()) {
            sb.append('-').append(profile);
        }
        return sb.append(".properties").toString();
    }

    public static String getProperty(final String key) {
        return properties.getProperty(key);
    }
}
```

3. rész

Nemzetköziesítés

i18n, L10n, g11n

- A számítógépes alkalmazásoknál általában, de főként a grafikus felhasználói felületek esetében, általános követelmény a nemzetköziesítés és lokalizáció.
- A nemzetköziesítés többnyelvűséget jelent: a különböző anyanyelvű felhasználók könnyen (a program módosítása/újrafordítása nélkül) állíthassák át a felület nyelvét.
- Fontos továbbá a különböző helyi, a felhasználó országában megszokott megjelenítési módok, konvenciók betartása, például numerikus adatok, dátumok, pénzösszegek megjelenítésének esetén. A már nemzetköziesített, több nyelvet támogató programok esetében a lokalizáció folyamata felelős az adatok megfelelő megjelenítéséért.
- Az angol nyelvű szakterminológiában a két folyamat neve **internationalization**, illetve **localization**, és gyakran használják az i18n és L10n rövidítéseket, ahol a számok a megnevezések első és utolsó betűi közötti karakterek számát jelölik. Mivel a két műveletet általában együtt alkalmazzák, a szakirodalomban néhol találkozhatunk a **globalization** (g11n) kifejezéssel is, a két művelet együttes megnevezéseként.

Locale

- ISO Language Code (ISO 639), ISO Country Code (ISO 3166)

```
Locale l1 = new Locale ("en", "US");
Locale l2 = new Locale ("en", "GB");
```

- Támogatott **`Locale`**-ek:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Locale;

public class LocaleExample {
    private static final Logger LOG = LoggerFactory.getLogger(LocaleExample.class);

    public static void main(String[] args) {
        Locale[] list = Locale.getAvailableLocales();
        for (Locale l:list) {
            LOG.info("Supported locale: {}-{}", l.getLanguage(), l.getCountry());
        }
    }
}
```

Locale

- Alapértelmezett érték megadása: a **`user.language`** és **`user.region`** tulajdonságok beállításával, vagy a **`Locale`** osztály **`setDefault()`** metódusának segítségével.
- A **`user.region`** és **`user.language`** rendszertulajdonságok a VM indulásakor lesznek beállítva, ha változtatni akarjuk ezeket, akkor a VM argumentumaként kell megadnunk az értékeket, annak indításakor. Az értékek futási idejű változtatása (**`System.setProperties`** metódus segítségével nem vonja maga után a **`defaultLocale`** érték módosítását).
- *Következmény:* ha futási időben akarjuk változtatni az alapértelmezett nyelvet/régiót a **`Locale`** osztály **`setDefault`** metódusát alkalmazhatjuk.

ResourceBundle


```
import java.util.Locale;
import java.util.ResourceBundle;

public class LocaleExample {
    public static void main (String[] args) {

        // custom locale
        Locale currentLocale = new Locale("hu", "HU");
        // or default locale
        Locale currentLocale = Locale.getDefault();

        // load resource bundle
        ResourceBundle messages = ResourceBundle.getBundle("MessageBundle", currentLocale);

        System.out.println(messages.getString("greetings"));
        System.out.println(messages.getString("inquiry"));
        System.out.println(messages.getString("farewell"));
    }
}
```

MessageBundle

- MessageBundle.properties

```
greetings = Hello.
farewell = Goodbye.
inquiry = How are you?
```

- MessageBundle_de_DE.properties

```
greetings = Hallo.
farewell = Tschüß.
inquiry = Wie geht's?
```

- MessageBundle_en_US.properties:

```
greetings = Hello.
farewell = Goodbye.
inquiry = How are you?
```

- MessageBundle_fr_FR.properties:

```
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

MessageBundle

- Ha a megadott értékek nem felelnek meg az érvényes régió és nyelv kódoknak, a fordító nem jelez hibát, mivel a problémát megoldja a `ResourceBundle` osztály keresési stratégiája.
- A gyártómetódus először megpróbálja betölteni a megadott `Locale`-nak megfelelő erőforrás állományt. Amennyiben ez nem sikerül megpróbálja betölteni az alapértelmezett régiónak és nyelvnek megfelelő állományt.
- Megjegyezhetjük, hogy a `getBundle` metódus második paramétere el is hagyható, és ilyenkor is az alapértelmezett `Locale`-nak megfelelő állományt próbálja meg betölteni a rendszer.
- Ha az alapértelmezett `Locale`-nak megfelelő állomány sem létezik, a rendszer továbblépik a keresésben, és megpróbálja betölteni az alapértelmezettként megadott erőforrás állományt.
- Megjegyezhetjük, hogy ugyanezt az eredményt elérhetjük úgy is, hogy a `Locale` konstruktorának üres karakterláncokat adunk meg paraméterként. Ilyen esetben mindig az alapértelmezettként megadott állomány, és nem az alapértelmezett `Locale`-nak megfelelő állomány kerül betöltésre.
- Az erőforrások betöltésekor probléma csak akkor léphet fel, ha az alapértelmezett állomány sem található. Ebben az esetben `MissingResourceException` típusú futási idejű kivételt kapunk.

MessageBundle

- Oda kell figyelnünk az erőforrás állományok elhelyezésére, illetve az alap állománynév megadásakor az útvonal helyes megadására.
- Az erőforrás állományokat, az osztályokhoz hasonlóan, a rendszer az osztálybetöltő (classloader) segítségével tölti be (a `ClassLoader.getResource` metódus által). Ennek megfelelően a projekthez tartozó csomagok hierarchiájában keresi ezeket. Amennyiben nem használunk csomagokat, az állományoknak az osztályállományokkal egy könyvtárban kell lenniük. Ha a betöltést végző osztály valamilyen csomagban található, akkor az állományokat a projekt gyökérkönyvtárában kell elhelyeznünk.
- Általában az erőforrás állományokat egy projekten belül külön könyvtárakba csoportosítjuk. Ebben az esetben az alap állománynév megadásakor az állományt tartalmazó csomagot is meg kell határoznunk. Tételezzük fel, hogy példánk esetében az erőforrás állományokat a projekten belül egy külön "res" könyvtárban tároljuk. Ebben az esetben a program megfelelő sora a következőképpen módosul:

```
java messages = ResourceBundle.getBundle("res.MessageBundle",
currentLocale);
```

MessageBundle

- A rendszer a megadott `"res.MessageBundle"` karakterláncból felépíti a megfelelő elérési útvonalat (a `"."` karaktereket `"/"` karakterekkel helyettesítve, és az állománynevet a `Locale`-nak megfelelő utótagokkal, illetve a `.properties` kiterjesztéssel kiegészítve).
- Megjegyzendő, hogy a keresés ebben az esetben is a csomaghierarchián belül, a classpath gyökerétől kiindulva történik. Amennyiben abszolút elérési útvonalat szeretnénk megadni (bár ez ritkán szükséges, előfordulhat, ha az állományok a projekten kívül találhatóak), akkor egy, a célnak megfelelő osztálybetöltőt (pl. `URLClassLoader`) alkalmazhatunk (a `getBundle` metódus háromparaméteres változatának segítségével adhatunk át a metódusnak egy erre mutató referenciát).

Number format, currency and dates

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
public class NumberFormatDemo {
    private static final Logger LOG = LoggerFactory.getLogger(NumberFormatDemo.class);

    public static void displayNumber(Locale currentLocale) {
        Integer intExample = 123456;
        Double doubleExample = 345987.246;

        // locale-specific formatter
        NumberFormat numberFormatter = NumberFormat.getNumberInstance(currentLocale);
        LOG.info("Integer  in locale {} is {}", currentLocale, numberFormatter.format(intExample));
        LOG.info("Double   in locale {} is {}", currentLocale, numberFormatter.format(doubleExample));
    }

    public static void displayCurrency(Locale currentLocale) {
        Double currency = 9876543.21;

        // locale-specific currency formatter
        NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance(currentLocale);
        LOG.info("Currency in locale {} is {}", currentLocale, currencyFormatter.format(currency));
    }

    public static void displayPercent(Locale currentLocale) {
        Double percent = 0.75;

        // locale-specific percent formatter
        NumberFormat percentFormatter = NumberFormat.getPercentInstance(currentLocale);
        LOG.info("Percent  in locale {} is {}", currentLocale, percentFormatter.format(percent));
    }
    ...
```

Number format, currency and dates

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
    ...
    public static void displayDatetime(Locale currentLocale) {
        Date dateTime = new Date();

        // locale-specific datetime formatter
        DateFormat datetimeFormatter = DateFormat.getDateTimeInstance(
            DateFormat.LONG, DateFormat.LONG, currentLocale);
        LOG.info("Datetime in locale {} is {}", currentLocale, datetimeFormatter.format(dateTime));
    }

    public static void main (String[] args) {
        Locale[] locales = {
            new Locale ("fr","FR"),
            new Locale ("de","DE"),
            new Locale ("en","US"),
        };

        for (Locale locale : locales) {
            LOG.info("");
            displayNumber(locale);
            displayCurrency(locale);
            displayPercent(locale);
            displayDatetime(locale);
        }
    }
}
```

Number format, currency and dates

Kimenet:

```
Integer  in locale fr_FR is 123 456
Double   in locale fr_FR is 345 987,246
Currency in locale fr_FR is 9 876 543,21 €
Percent  in locale fr_FR is 75 %
Datetime in locale fr_FR is 11 novembre 2018 18:36:45 EET

Integer  in locale de_DE is 123.456
Double   in locale de_DE is 345.987,246
Currency in locale de_DE is 9.876.543,21 €
Percent  in locale de_DE is 75%
Datetime in locale de_DE is 11. November 2018 18:36:45 OEZ

Integer  in locale en_US is 123,456
Double   in locale en_US is 345,987.246
Currency in locale en_US is $9,876,543.21
Percent  in locale en_US is 75%
Datetime in locale en_US is November 11, 2018 6:36:45 PM EET
```

Példa: BiblioSpring i18n

CODESPRING
OUR CODE YOUR SUCCESS
20 years

```
public class LabelProvider {

    private static final Logger LOG = LoggerFactory.getLogger(LabelProvider.class);
    private static final String BUNDLE_NAME = "BSSwingLabels";

    private static Locale locale = Locale.getDefault();
    private static ResourceBundle resourceBundle =
            ResourceBundle.getBundle(BUNDLE_NAME, locale);

    public static String getLabel(String key) {
        try {
            return resourceBundle.getString(key);
        } catch (MissingResourceException e) {
            LOG.warn("Key '{}' missing for locale {}", key, locale);
            return '!' + key + '!';
        }
    }

    public static void setLocale (Locale newLocale) {
        locale = newLocale;
        resourceBundle = ResourceBundle.getBundle(BUNDLE_NAME, locale);
    }
}
```

Példa: BiblioSpring i18n

CODESPRING
OUR CODE YOUR SUCCESS
20 years

- BSSwingLabels.properties

```
MainFrame.loanButton=Loans
MainFrame.reservationButton=Reservations
MainFrame.title=BiblioSpring Admin Interface
MainFrame.titleButton=Title Management
MainFrame.userButton=User Managemenet
```

- BSSwingLabels_hu_HU.properties

```
MainFrame.loanButton=Kölcsönzés
MainFrame.reservationButton=Foglalás
MainFrame.title=BiblioSpring Admin felület
MainFrame.titleButton=Címek menedzsmentje
MainFrame.userButton=Felhasználók menedzsmentje
```

Példa: BiblioSpring i18n


```
public class MainFrame extends JFrame {
    ...
    private void initializeGUI() {
        ...
        bookButton = new JButton (LabelProvider.getLabel ("MainFrame.titleButton"));
        userButton = new JButton (LabelProvider.getLabel ("MainFrame.userButton"));
        reservationButton = new JButton (LabelProvider.getLabel ("MainFrame.reservationButton"));
        loanButton = new JButton (LabelProvider.getLabel ("MainFrame.loanButton"));
        ...
    }
    ...
}
```