2013. Július

Informatikai Navigátor

Érdekes Java Programozói könyvtárak

Gondolatok a szoftverek használatáról és fejlesztéséről

9. szám

Tartalomjegyzék

Főszerkesztő: Nyiri Imre (imre.nyiri@gmail.com)

1. JAXB - A Java és XML összekötése

Az XML adatok és a Java objektumok közötti leképzés fontos eszköze egy korszerű fejlesztői környezetnek. Sok olyan Java API létezik, amelyek XML-t kezelnek, de itt most az a kérdés, hogy egy XML-ben tárolt adatszerkezetet hogyan konvertálunk oda-vissza egy alkalmas Java objektumba. Ezt oldja meg a JDK 1.6 óta beépített részként működő *JAXB* könyvtár.

A JAXB[1] az XML adatok és a Java objektumok közötti leképzést valósítja meg, aminek nyilvánvaló előnye az, hogy az XML adatstruktúrákat Java objektumként tudjuk kezelni és fordítva. A Java sok helyen felhasználja ezt a beépített szolgáltatást, erre jó példa az *JAX-WS* webservice technológia XML↔Java leképzése. A leképzés azt jelenti, hogy a Java objektum adattagjait (vagy más néven a property-ket) leképezzük (ezt nevezzük *binding*-nak) az XML tagekre. Ezt a feladatot a JAXB kifinomultan és rugalmasan képes ellátni. A fejlesztést végezhetjük a Java osztályokból kiindulva, de lehetséges az is, hogy az XML sémák alapján generáljuk a befogadó objektumok osztályait. A következő egyszerű példa az alapszintű használatba vezet be.

Bevezető példa

A Java osztályok

Legyen a célunk az, hogy az 1-1. Programlistán látható *Employee* class objektumait XML formátumra alakítsuk (perzisztáljuk). Amint látjuk ez az osztály egy egyszerű *JavaBean* (más szavakkal *POJO*[2]), aminek van néhány adatmezője, köztük az egyik – az *Address* class – maga is egy osztály, amit az 1-2. Programlista mutat. Mindez igazán egyszerű, de most szeretnénk egy olyan XML modellt rendelni ehhez, amibe egy *Employee* példány elmenthető. A JAXB 2.0 óta mindez úgy valósítható meg, hogy a Java osztályt annotáljuk a JAXB-os „@”-okkal. Tekintettel arra, hogy az *Employee* class reprezentálja az egész objektum kezdetét, így az őt leíró XML gyökérelemének is ez felel meg. Ezt reprezentálja a *@XmlRootElement*, aminek paramétereiben még azt is megadtuk, hogy ez a gyökér XML tag az *org.ceg.test* XML névtérben lesz. Az *@XmlType* annotáció azt mondja meg, hogy ez a class egyben egy XSD-beli típusnév is lenne, amit láthatunk is, amennyiben XML sémát szeretnénk a későbbiekben legenerálni. A névteret itt is meg kell adni.

```
// 1-1. Programlista: Employee.java

package org.cs.test;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.➥
    XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(namespace = "org.ceg.test")
@XmlType(namespace = "org.ceg.test")
public class Employee
{
    @XmlElement
    Address address;

    @XmlElement(namespace = "org.ceg.test")
    String name;

    @XmlElement
    String sex;

    @XmlElement
    int kor;

    @XmlElement
    double suly;
}
```

Most az *Employee* mindegyik adattagját elláttuk *@XmlElement* jelzéssel, ami azt jelenti,

[1] JAXB=Java API for XML Binding
[2] POJO=Plain Old Java Object

hogy ezen mezők mindegyikének egy-egy XML tag-et szeretnénk megfeleltetni. A tag neve a bean property neve lesz, de egy esetleges *name annotáció attribútum* megadással explicit is megadhatjuk, hogy mi legyen az adott XML címke neve. Esetünkben most csak a *namespace* paramétert használjuk próbaként, hagyjuk, hogy az XML tag nevek automatikusan képződjenek a class adatmezők nevéből. Az *Address* class annotálása ezek után már érthető.

```
// 1-2. Programlista: Address.java

package org.cs.test;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlType(namespace = "org.ceg.test")
public class Address
{
    @XmlElement
    String street;

    @XmlElement
    int houseNumber;
}
```

Az eddigiekben odáig jutottunk, hogy az *Employee* és *Address* osztályokat elláttuk JAXB annotációkkal, ezzel megteremtettük a lehetőségét annak, hogy kipróbáljuk a JAXB Java Object→XML leképzési lehetőségét, amit *marshalling*nak nevezünk. Nézzük is meg a használatát!

Az XML előállítása (marshaller)

Az 1-3. Programlista a JAXB használatát szemlélteti, feladata az, hogy egy 15-24 sorok között előállított *e Employee* objektumot XML fájlba mentse. Ehhez először a 26. sorban egy *jaxbContext* objektum előállítása szükséges, amely az *Employee* class mentén ad egy kezelő kontextust, ezért volt szükséges ezt a *newInstance()* metódusban paraméterül átadni. A 27. sorban egy általános eljárással lekérünk a kontextustól egy *jaxbMarshaller* objektumot, ami a tényleges Java Object→XML átalakítást fogja végezni. A 28. sor nem kötelező lépés, de mi azt szeretnénk, hogy az előállított XML szépen formázva nézzen ki. A 30. sor egy *xmlFile* nevű *File* objektum, ahova majd az eredményül kapott XML-t mentjük. Végül a 30. sorban elvégezzük a feladatot, azaz az *e* objektumot perzisztáljuk az *xmlFile* helyre.

```
1  // 1-3. Programlista: TestJAXBMarshaller.java
2
3  package org.cs.test;
4
5  import java.io.File;
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Marshaller;
9
10 public class TestJAXBMarshaller
11 {
12     public static void main(String[] args)
13             throws JAXBException
14     {
15         Address a = new Address();
16         a.houseNumber = 12;
17         a.street = "Váci út";
18
19         Employee e = new Employee();
20         e.address = a;
21         e.kor = 32;
22         e.name = "Alma Ferenc";
23         e.sex = "férfi";
24         e.suly = 88;
25
26         JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);
27         Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

```
28          jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
29          File xmlFile = new File("/home/tanulas/xml/Employee.xml");
30          jaxbMarshaller.marshal(e, xmlFile);
31      }
32  }
```

A program lefuttatása után az 1-4. Programlista mutatja a legyártott *Employee.xml* fájl tartalmát. Az alábbiakat javasoljuk megfigyelni:

- A class nevek alapértelmezetten kisbetűvel kezdődnek.
- Az adatmező nevek teljesen megmaradnak.
- Az XML névtér minden olyan tag-hez bekerült, ahol ezt kértük a „@”-ok megadásánál.

```
// 1-4. Programlista: Employee.xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:employee xmlns:ns2="org.ceg.test">
    <address>
        <street>Váci út</street>
        <houseNumber>12</houseNumber>
    </address>
    <ns2:name>Alma Ferenc</ns2:name>
    <sex>férfi</sex>
    <kor>32</kor>
    <suly>88.0</suly>
</ns2:employee>
```

Az esetek jelentős részénél nem mindig fájlba szeretnénk tenni az előállított XML-t, ezért a *marshal()* metódusnak is több alakja van. Tipikus használat, hogy egy *Writer* interface-szel rendelkező objektumba pakoljuk az XML outputot, ilyen lehet például a *StringWriter* class, aminek a használatát a 1-5. Programlistáról tanulmányozhatjuk. Ekkor a *marshal()* metódus a *writer* objektumba teszi az eredményt, ahonnan azt *String* vagy *StringBuffer* objektumként is lekérhetjük.

```
// 1-5. Programlista: StringWriter

StringWriter writer = new StringWriter();
jaxbMarshaller.marshal(e, writer);

String       data       = writer.toString();
StringBuffer dataBuffer = writer.getBuffer();
```

Az XML visszaolvasása (unmarshaller)

Most végezzük el a fordított feladatot, azaz írjunk egy olyan JAXB alapú programot, ami az XML→Java Object beolvasást (*unmarshalling*) valósítja meg. Mindezt végezzük el a most előállított *Employee.xml* fájlra. Amennyiben nem lenne megfelelően „@”-olt (azaz szebben fogalmazva: annotált) befogadó Java class, akkor azt el kéne készítenünk, de ezek már itt vannak: *Employee*, *Address*. A feladat megoldását az 1-6. Programlista mutatja.

```
1   // 1-6. Programlista: TestJAXBUnmarshaller.java
2
3   package org.cs.test;
4
5   import java.io.File;
6   import javax.xml.bind.JAXBContext;
7   import javax.xml.bind.JAXBException;
8   import javax.xml.bind.Unmarshaller;
9
10  public class TestJAXBUnmarshaller
11  {
12      public static void main(String[] args)
13              throws JAXBException
14      {
15        JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);
16        Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
17        File xmlFile = new File("/home/tanulas/xml/Employee.xml");
18        Employee e = (Employee) jaxbUnmarshaller.unmarshal(xmlFile);
```

```
19          System.out.println( e.name );
20          System.out.println( e.address.street );
21      }
22  }
```

A 15. sorban a már ismert módon egy *JAXBContext* objektumot készítünk, amiben csak az *Employee* class játszik szerepet. Ezt azért emeltük most ki, mert általában több osztály is megadható lenne a gyártómetódusnak, erre majd a későbbiekben mutatunk példát. A 16. sorban természetesen most egy *Unmarshaller* objektumot kérünk le, hogy azt a 17. sorban megadott fájlra működtessük. Magát a beolvasást a 18. sor *unmarshal()* metódusa végzi, amelynek használatában nincs semmi különleges. Az utolsó 2 sor csak tesztelésként kiírja a képernyőre az XML fájlból létrehozott objektum 2 mezőjének az értékét. Az XML→Java Object átalakításnál is gyakori eset, amikor nem fájlból érkezik az input XML, ezért az *unmarshal()* metódusnak is sok alakja van. A gyakorlatban talán az egyik leghasznosabb esetet mutatja az 1-7. Programlista, ahol egy Java *String* objektumban van az XML, amit egy *StringReader* objektum közbeiktatásával már át is adhatunk a metódusnak.

```
// 1-7. Programlista: StringReader

String xmlString = beolvasFromXMLFile();
StringReader reader = new StringReader( ➥
    xmlString );
Employee em = (Employee) jaxbUnmarshaller.➥
    unmarshal( reader );
```

Az XML séma (XSD) előállítása

Alapvetően fontos az XML világban, hogy annak a sémáját is lehetőleg birtokoljuk, így annak előállításához a JAXB a *schemagen* parancsot biztosítja, ami a JDK része. Amennyiben a *java/javac* elérhető, úgy ez a parancs is rendelkezésre áll. Használata egyszerű, mindössze meg kell adnunk azt a Java forrásfájlt, amiben egy JAXB annotált class található. Az *Employee* class-ra esetünkben ez a parancs futtatható:

```
schemagen -cp /opt/netbeans/myprojects/TestProjects/➥
    JavaTest/src /opt/netbeans/myprojects/➥
    TestProjects/JavaTest/src/org/cs/test/Employee.➥
    java
```

A *-cp* kapcsolót azért adtuk meg, hogy a beágyazott *Address* osztályt is megtalálja a parancs, hiszen ez a séma része kell legyen. A parancs lefutása után az 1-8. Programlista által tartalmazott sémát kapjuk meg.

```
1   // 1-8. Programlista: Employee XML séma
2
3   <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
4   <xs:schema version="1.0" targetNamespace="org.ceg.test" xmlns:tns="org.ceg.test" xmlns:xs="http://www.w3.org➥
        /2001/XMLSchema">
5
6     <xs:element name="employee" type="tns:employee"/>
7
8     <xs:complexType name="employee">
9       <xs:sequence>
10        <xs:element name="address" type="tns:address" minOccurs="0"/>
11        <xs:element name="name" type="xs:string" form="qualified" minOccurs="0"/>
12        <xs:element name="sex" type="xs:string" minOccurs="0"/>
13        <xs:element name="kor" type="xs:int"/>
14        <xs:element name="suly" type="xs:double"/>
15      </xs:sequence>
16    </xs:complexType>
17
18    <xs:complexType name="address">
19      <xs:sequence>
20        <xs:element name="street" type="xs:string" minOccurs="0"/>
21        <xs:element name="houseNumber" type="xs:int"/>
22      </xs:sequence>
23    </xs:complexType>
24  </xs:schema>
```

Ránézésre tetszetős, az XSD típusok szépen leképződtek a megfelelő Java típusokról. A szerkezete is megfelelő. A 11. sor *form="qualified"* jelzése azt jelenti, hogy ez a tag a *targetNamespace*-ben (*org.ceg.test*) van. Az *address complexType* a várt módon, külön nevesítve megtalálható és a 10. sorban korrekt módon került használatra.

Java osztályok generálása XML sémából

A gyakorlatban az is sokszor előfordul, hogy az XSD már rendelkezésünkre áll. Nagyon sok munkát jelentene a megfelelő befogadó Java osztály létrehozása és JAXB „@”-ok elhelyezése, de szerencsére a JAXB erre is tartalmaz egy JDK parancsot, aminek a neve *xjc*. A sémák általában valamilyen szabvány részeként állnak rendelkezésünkre, például: *UBL* (Universal Business Language), *ebXML*, *OFX* (Open Financial Exchange), de az is lehet, hogy a rendszerek közötti kommunikáció egy sémával, platformfüggetlen módon volt csak definiálható, ami nagy előny. Az *Employee* class alapján generált séma felhasználásával generáljuk le a Java osztályokat, azaz tegyünk úgy mintha azok még nem léteznének:

```
xjc Employee.xsd
```

Az eredményt az 1-9., 1-10. és 1-11. Programlisták tartalmazzák.

```
1  // 1-9. Programlista: Az xjc által generált Address osztály
2
3  //
4  // This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference ➥
       Implementation, v2.2.4-2
5  // See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
6  // Any modifications to this file will be lost upon recompilation of the source schema.
7  // Generated on: 2013.03.19 at 06:49:04 PM GMT
8  //
9  package test.ceg.org;
10
11 import javax.xml.bind.annotation.XmlAccessType;
12 import javax.xml.bind.annotation.XmlAccessorType;
13 import javax.xml.bind.annotation.XmlType;
14
15 /**
16  * <p>Java class for address complex type.
17  *
18  * <p>The following schema fragment specifies the expected content contained
19  * within this class.
20  *
21  * <pre>
22  * &lt;complexType name="address">
23  *   &lt;complexContent>
24  *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
25  *       &lt;sequence>
26  *         &lt;element name="street" type="{http://www.w3.org/2001/XMLSchema}string" minOccurs➥
        ="0"/>
27  *         &lt;element name="houseNumber" type="{http://www.w3.org/2001/XMLSchema}int"/>
28  *       &lt;/sequence>
29  *     &lt;/restriction>
30  *   &lt;/complexContent>
31  * &lt;/complexType>
32  * </pre>
33  *
34  *
35  */
36 @XmlAccessorType(XmlAccessType.FIELD)
37 @XmlType(name = "address", propOrder =
38 {
```

```
39      "street",
40      "houseNumber"
41  })
42  public class Address
43  {
44
45      protected String street;
46      protected int houseNumber;
47
48      /**
49       * Gets the value of the street property.
50       *
51       * @return possible object is {@link String }
52       *
53       */
54      public String getStreet()
55      {
56          return street;
57      }
58
59      /**
60       * Sets the value of the street property.
61       *
62       * @param value allowed object is {@link String }
63       *
64       */
65      public void setStreet(String value)
66      {
67          this.street = value;
68      }
69
70      /**
71       * Gets the value of the houseNumber property.
72       *
73       */
74      public int getHouseNumber()
75      {
76          return houseNumber;
77      }
78
79      /**
80       * Sets the value of the houseNumber property.
81       *
82       */
83      public void setHouseNumber(int value)
84      {
85          this.houseNumber = value;
86      }
87  }
```

```
1   // 1-10. Programlista: Az xjc által generált Employee osztály
2
3   //
4   // This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference ➥
        Implementation, v2.2.4-2
5   // See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
6   // Any modifications to this file will be lost upon recompilation of the source schema.
7   // Generated on: 2013.03.19 at 06:49:04 PM GMT
8   //
9   package test.ceg.org;
10
11  import javax.xml.bind.annotation.XmlAccessType;
12  import javax.xml.bind.annotation.XmlAccessorType;
13  import javax.xml.bind.annotation.XmlElement;
14  import javax.xml.bind.annotation.XmlRootElement;
```

```
15 import javax.xml.bind.annotation.XmlType;
16
17 /**
18  * <p>Java class for employee complex type.
19  *
20  * <p>The following schema fragment specifies the expected content contained
21  * within this class.
22  *
23  * <pre>
24  * &lt;complexType name="employee">
25  *   &lt;complexContent>
26  *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
27  *       &lt;sequence>
28  *         &lt;element name="address" type="{org.ceg.test}address" minOccurs="0"/>
29  *         &lt;element name="name" type="{http://www.w3.org/2001/XMLSchema}string" minOccurs="0" form="qualified"/>
30  *         &lt;element name="sex" type="{http://www.w3.org/2001/XMLSchema}string" minOccurs="0"/>
31  *         &lt;element name="kor" type="{http://www.w3.org/2001/XMLSchema}int"/>
32  *         &lt;element name="suly" type="{http://www.w3.org/2001/XMLSchema}double"/>
33  *       &lt;/sequence>
34  *     &lt;/restriction>
35  *   &lt;/complexContent>
36  * &lt;/complexType>
37  * </pre>
38  *
39  *
40  */
41 @XmlRootElement(namespace = "org.ceg.test")
42 @XmlAccessorType(XmlAccessType.FIELD)
43 @XmlType(name = "employee", propOrder =
44 {
45     "address",
46     "name",
47     "sex",
48     "kor",
49     "suly"
50 })
51 public class Employee
52 {
53
54     protected Address address;
55     @XmlElement(namespace = "org.ceg.test")
56     protected String name;
57     protected String sex;
58     protected int kor;
59     protected double suly;
60
61     /**
62      * Gets the value of the address property.
63      *
64      * @return possible object is {@link Address }
65      *
66      */
67     public Address getAddress()
68     {
69         return address;
70     }
71
72     /**
73      * Sets the value of the address property.
74      *
75      * @param value allowed object is {@link Address }
76      *
77      */
```

```
78      public void setAddress(Address value)
79      {
80          this.address = value;
81      }
82
83      /**
84       * Gets the value of the name property.
85       *
86       * @return possible object is {@link String }
87       *
88       */
89      public String getName()
90      {
91          return name;
92      }
93
94      /**
95       * Sets the value of the name property.
96       *
97       * @param value allowed object is {@link String }
98       *
99       */
100     public void setName(String value)
101     {
102         this.name = value;
103     }
104
105     /**
106      * Gets the value of the sex property.
107      *
108      * @return possible object is {@link String }
109      *
110      */
111     public String getSex()
112     {
113         return sex;
114     }
115
116     /**
117      * Sets the value of the sex property.
118      *
119      * @param value allowed object is {@link String }
120      *
121      */
122     public void setSex(String value)
123     {
124         this.sex = value;
125     }
126
127     /**
128      * Gets the value of the kor property.
129      *
130      */
131     public int getKor()
132     {
133         return kor;
134     }
135
136     /**
137      * Sets the value of the kor property.
138      *
139      */
140     public void setKor(int value)
141     {
142         this.kor = value;
```

```
143     }
144
145     /**
146      * Gets the value of the suly property.
147      *
148      */
149     public double getSuly()
150     {
151         return suly;
152     }
153
154     /**
155      * Sets the value of the suly property.
156      *
157      */
158     public void setSuly(double value)
159     {
160         this.suly = value;
161     }
162 }
```

```
1  // 1-11. Programlista: Az xjc által generált ObjectFactory osztály
2
3  //
4  // This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference ➥
       Implementation, v2.2.4-2
5  // See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
6  // Any modifications to this file will be lost upon recompilation of the source schema.
7  // Generated on: 2013.03.19 at 06:49:04 PM GMT
8  //
9
10
11 package test.ceg.org;
12
13 import javax.xml.bind.JAXBElement;
14 import javax.xml.bind.annotation.XmlElementDecl;
15 import javax.xml.bind.annotation.XmlRegistry;
16 import javax.xml.namespace.QName;
17
18
19 /**
20  * This object contains factory methods for each
21  * Java content interface and Java element interface
22  * generated in the test.ceg.org package.
23  * <p>An ObjectFactory allows you to programatically
24  * construct new instances of the Java representation
25  * for XML content. The Java representation of XML
26  * content can consist of schema derived interfaces
27  * and classes representing the binding of schema
28  * type definitions, element declarations and model
29  * groups.  Factory methods for each of these are
30  * provided in this class.
31  *
32  */
33 @XmlRegistry
34 public class ObjectFactory {
35
36     private final static QName _Employee_QNAME = new QName("org.ceg.test", "employee");
37
38     /**
39      * Create a new ObjectFactory that can be used to create new instances of schema derived ➥
           classes for package: test.ceg.org
40      *
41      */
42     public ObjectFactory() {
```

```
43      }
44
45      /**
46       * Create an instance of {@link Employee }
47       *
48       */
49      public Employee createEmployee() {
50          return new Employee();
51      }
52
53      /**
54       * Create an instance of {@link Address }
55       *
56       */
57      public Address createAddress() {
58          return new Address();
59      }
60
61      /**
62       * Create an instance of {@link JAXBElement }{@code <}{@link Employee }{@code >}}
63       *
64       */
65      @XmlElementDecl(namespace = "org.ceg.test", name = "employee")
66      public JAXBElement<Employee> createEmployee(Employee value) {
67          return new JAXBElement<Employee>(_Employee_QNAME, Employee.class, null, value);
68      }
69
70  }
```

Szeretnénk kiemelni, hogy az *Employee* classhoz mi írtuk oda a *@XmlRootElement* annotációt, mert az *xjc* ezt valami miatt nem tette meg. Ettől eltekintve korrekt Java forrás jött létre, gyakorlatilag kézzel is ezt írtuk meg korábban. Az *Employee.xml* XML fájl beolvasása ezekkel a generált osztályokkal természetesen ugyanúgy működik, ha kipróbáljuk. Az *ObjectFactory* osztály gyártómetódusokat ad az *Employee* és *Address* objektumok előállítására, de ezek csak kényelmi funkciók.

Néhány apró változtatás

A példa befejezéseként teszünk néhány változtatást az eredeti 2 Java osztályunkon, majd megnézzük, hogy milyen sémát lehet velük generálni. A megváltoztatott 2 bean-t az 1-12. és 1-13. Programlisták mutatják.

```
// 1-12. Programlista: Employee.java

package org.cs.test.ttt;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.➥
    XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(
   namespace = "org.ceg.test",
   name = "Dolgozo"
)
@XmlType(namespace = "org.ceg.test")
public class Employee
{
    @XmlElement(name="cim",
     namespace = "http://org.cs.address")
    Address address;

    @XmlElement(name="nev",
     namespace = "org.ceg.test")
    String name;

    @XmlElement(name="nem")
    String sex;

    @XmlElement
    int kor;

    @XmlElement
    double suly;
}
```

```
// 1-13. Programlista: Address.java

package org.cs.test.ttt;

import javax.xml.bind.annotation.XmlElement;
```

```
import javax.xml.bind.annotation.XmlType;

@XmlType(
  name="cim",
  namespace = "org.ceg.test"
)
public class Address
{
    @XmlElement(name="utca")
    String street;

    @XmlElement(name="hazszam")
    int houseNumber;
}
```

A változtatások ezek voltak:

- Ahol nem volt magyar az adatmező neve, ott magyar *name* értékre állítottuk a generált címke nevet. Példa: street→utca.
- Az *address* mezőt egy minden eddigitől eltérő másik, *http://org.cs.address* nevű névtérbe tettük.

A *schemagen* parancs futtatása után most 2 XSD fájl keletkezett, amit az 1-14. és 1-15. Programlisták mutatnak.

```
1  // 1-14. Programlista: schema1.xsd
2
3  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
4  <xs:schema version="1.0" targetNamespace="org.ceg.test" xmlns:ns1="http://org.cs.address" xmlns:tns="org.ceg.➥
       test" xm
5
6    <xs:import namespace="http://org.cs.address" schemaLocation="schema2.xsd"/>
7
8    <xs:element name="Dolgozo" type="tns:employee"/>
9
10   <xs:complexType name="employee">
11     <xs:sequence>
12       <xs:element ref="ns1:cim" minOccurs="0"/>
13       <xs:element name="nev" type="xs:string" form="qualified" minOccurs="0"/>
14       <xs:element name="nem" type="xs:string" minOccurs="0"/>
15       <xs:element name="kor" type="xs:int"/>
16       <xs:element name="suly" type="xs:double"/>
17     </xs:sequence>
18   </xs:complexType>
19
20   <xs:complexType name="cim">
21     <xs:sequence>
22       <xs:element name="utca" type="xs:string" minOccurs="0"/>
23       <xs:element name="hazszam" type="xs:int"/>
24     </xs:sequence>
25   </xs:complexType>
26 </xs:schema>
```

```
1  // 1-15. Programlista: schema2.xsd
2
3  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
4  <xs:schema version="1.0" targetNamespace="http://org.cs.address" xmlns:ns1="org.ceg.test" xmlns:xs="http://www.➥
       w3.org
5
6  ␣␣<xs:import␣namespace="org.ceg.test"␣schemaLocation="schema1.xsd"/>
7
8  ␣␣<xs:element␣name="cim"␣type="ns1:cim"/>
9  </xs:schema>
```

Mit tanulhatunk ebből? Az első, amit észrevehetünk, hogy az XML tag neveket megváltoztattuk, azaz nem automatikusan a Java bean property névből képződnek ott, ahol eredetileg angol mező nevek voltak. A másik lényeges és megjegyzésre érdemes momentum az, hogy a 2 névtér miatt 2 darab XSD keletkezett, amik egymást beimportálják. Ezzel lehetővé vált, hogy mindkét *targetNamespace* létezzen és a megfelelően adjuk meg a névtérbe tartozó XML séma-típust.

Az XML sémák fordítása

Az előző példában minden alapvető dolgot megtanultunk, ezért a továbbiakban csak a tudásunkat mélyítjük egy kicsit tovább. Az 1-16. Programlistán látható XML séma egy minden nap használt, produktív megoldás része. A következőkben megnézzük, hogy milyen Java kód gene-

rálható erre és az egyes típusokhoz milyen Java típusok tartoznak majd. A sémával kapcsolatosan a következőkre hívjuk fel a figyelmet:

- Van benne 4 darab *complexType*
- A *targetNamespace*: *urn:jaxbtest/xsd2java*
- Mindegyik XSD skalár típus megtalálható benne
- Van olyan, ahol a *maxOccurs="unbounded"*, azaz a címke akárhányszor ismétlődhet.
- Amikor nem adjuk meg egy sémarészhez a *minOccurs* és *maxOccurs* attribútumokat, akkor az elem pontosan 1 alkalommal kell, hogy előforduljon. A *minOccurs="0"* azt jelenti, hogy 0 darab is lehet 1 XML-ben ebből a tag-ból, míg a *maxOccurs="unbounded"* pedig azt engedi, hogy akármennyi. Ekkor itt egy Java tömb vagy *List* várható majd a generálás eredményeként.

Ennyi előzmény után futtassuk le a következő *xjc* parancsot, ami legyártja a Java class-okat:

```
xjc -p cs.org.jaxb.test jaxbTest.xsd
```

A *-p* kapcsolóval azt adtuk meg, hogy a forrás milyen Java csomagba kerüljön, így a generált kód az *cs/org/jaxb/test* könyvtárba került.

```
1  // 1-16. Programlista: jaxbTest.xsd
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:jaxbtest/➥
       xsd2java" xmlns:xs="http://www.w3.org/2001/XMLSchema">
5      <xs:element name="select" type="sch:selectType" xmlns:sch="urn:jaxbtest/xsd2java"/>
6      <xs:complexType name="dataType">
7          <xs:sequence  minOccurs="0" maxOccurs="unbounded">
8              <xs:element type="sch:rowType" name="row" xmlns:sch="urn:jaxbtest/xsd2java"/>
9          </xs:sequence>
10     </xs:complexType>
11     <xs:complexType name="rowType">
12         <xs:sequence>
13             <xs:element type="xs:string" name="testString"/>
14             <xs:element type="xs:integer" name="testInteger"/>
15             <xs:element type="xs:int" name="testInt"/>
16             <xs:element type="xs:long" name="testLong"/>
17             <xs:element type="xs:short" name="testShort"/>
18             <xs:element type="xs:decimal" name="testDecimal"/>
19             <xs:element type="xs:float" name="testFloat"/>
20             <xs:element type="xs:double" name="testDouble"/>
21             <xs:element type="xs:boolean" name="testBoolean"/>
22             <xs:element type="xs:byte" name="testByte"/>
23             <xs:element type="xs:QName" name="testQName"/>
24             <xs:element type="xs:dateTime" name="testDateTime"/>
25             <xs:element type="xs:base64Binary" name="testBase64Binary"/>
26             <xs:element type="xs:hexBinary" name="testHexBinary"/>
27             <xs:element type="xs:unsignedInt" name="testUnsignedInt"/>
28             <xs:element type="xs:unsignedShort" name="testUnsignedShort"/>
29             <xs:element type="xs:unsignedByte" name="testUnsignedByte2"/>
30             <xs:element type="xs:time" name="testTime"/>
31             <xs:element type="xs:date" name="testDate"/>
32             <xs:element type="xs:anySimpleType" name="testAnySimpleType"/>
33             <xs:element type="xs:duration" name="testDuration"/>
34         </xs:sequence>
35     </xs:complexType>
36     <xs:complexType name="controllType">
37         <xs:sequence>
38             <xs:element type="xs:string" name="theme"/>
39             <xs:element type="xs:dateTime" name="timestamp"/>
40             <xs:element type="xs:string" name="query"/>
41             <xs:element type="xs:string" name="table"/>
42             <xs:element type="xs:string" name="datasource"/>
43             <xs:element type="xs:long" name="rowcnt"/>
44             <xs:element type="xs:long" name="resultsetid"/>
45         </xs:sequence>
46     </xs:complexType>
47     <xs:complexType name="selectType">
48         <xs:sequence>
49             <xs:element type="sch:controllType" name="controll" xmlns:sch="urn:jaxbtest/xsd2java"/>
50             <xs:element type="sch:dataType" name="data" xmlns:sch="urn:jaxbtest/xsd2java"/>
51         </xs:sequence>
52     </xs:complexType>
53 </xs:schema>
```

```
1  // 1-17. Programlista: ControllType.java
2
3  //
4  // This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference ➥
       Implementation, v2.2.4-2
5  // See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
6  // Any modifications to this file will be lost upon recompilation of the source schema.
7  // Generated on: 2013.04.06 at 02:59:45 PM GMT
8  //
9
10
11 package cs.org.jaxb.test;
12
13 import javax.xml.bind.annotation.XmlAccessType;
14 import javax.xml.bind.annotation.XmlAccessorType;
15 import javax.xml.bind.annotation.XmlElement;
16 import javax.xml.bind.annotation.XmlSchemaType;
17 import javax.xml.bind.annotation.XmlType;
18 import javax.xml.datatype.XMLGregorianCalendar;
19
20
21 /**
22  * <p>Java class for controllType complex type.
23  *
24  * <p>The following schema fragment specifies the expected content contained within this class.
25  *
26  * <pre>
27  * &lt;complexType name="controllType">
28  *   &lt;complexContent>
29  *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
30  *       &lt;sequence>
31  *         &lt;element name="theme" type="{http://www.w3.org/2001/XMLSchema}string"/>
32  *         &lt;element name="timestamp" type="{http://www.w3.org/2001/XMLSchema}dateTime"/>
33  *         &lt;element name="query" type="{http://www.w3.org/2001/XMLSchema}string"/>
34  *         &lt;element name="table" type="{http://www.w3.org/2001/XMLSchema}string"/>
35  *         &lt;element name="datasource" type="{http://www.w3.org/2001/XMLSchema}string"/>
36  *         &lt;element name="rowcnt" type="{http://www.w3.org/2001/XMLSchema}long"/>
37  *         &lt;element name="resultsetid" type="{http://www.w3.org/2001/XMLSchema}long"/>
38  *       &lt;/sequence>
39  *     &lt;/restriction>
40  *   &lt;/complexContent>
41  * &lt;/complexType>
42  * </pre>
43  *
44  *
45  */
46 @XmlAccessorType(XmlAccessType.FIELD)
47 @XmlType(name = "controllType", propOrder = {
48     "theme",
49     "timestamp",
50     "query",
51     "table",
52     "datasource",
53     "rowcnt",
54     "resultsetid"
55 })
56 public class ControllType
57 {
58     @XmlElement(required = true)
59     protected String theme;
60     @XmlElement(required = true)
61     @XmlSchemaType(name = "dateTime")
62     protected XMLGregorianCalendar timestamp;
63     @XmlElement(required = true)
```

```
64      protected String query;
65      @XmlElement(required = true)
66      protected String table;
67      @XmlElement(required = true)
68      protected String datasource;
69      protected long rowcnt;
70      protected long resultsetid;
71  ...
72  setter és getter metódusok
73      /**
74       * Gets the value of the timestamp property.
75       *
76       * @return
77       *     possible object is
78       *     {@link XMLGregorianCalendar }
79       *
80       */
81      public XMLGregorianCalendar getTimestamp() {
82          return timestamp;
83      }
84
85      /**
86       * Sets the value of the timestamp property.
87       *
88       * @param value
89       *     allowed object is
90       *     {@link XMLGregorianCalendar }
91       *
92       */
93      public void setTimestamp(XMLGregorianCalendar value) {
94          this.timestamp = value;
95      }
96  }
```

```
1   // 1-18. Programlista: RowType.java
2
3   package cs.org.jaxb.test;
4
5   import java.math.BigDecimal;
6   import java.math.BigInteger;
7   import javax.xml.bind.annotation.XmlAccessType;
8   import javax.xml.bind.annotation.XmlAccessorType;
9   import javax.xml.bind.annotation.XmlElement;
10  import javax.xml.bind.annotation.XmlSchemaType;
11  import javax.xml.bind.annotation.XmlType;
12  import javax.xml.bind.annotation.adapters.HexBinaryAdapter;
13  import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
14  import javax.xml.datatype.Duration;
15  import javax.xml.datatype.XMLGregorianCalendar;
16  import javax.xml.namespace.QName;
17
18
19  /**
20   * <p>Java class for rowType complex type.
21   *
22   * <p>The following schema fragment specifies the expected content contained within this class.
23   *
24   * <pre>
25   * &lt;complexType name="rowType">
26   *   &lt;complexContent>
27   *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
28   *       &lt;sequence>
29   *         &lt;element name="testString" type="{http://www.w3.org/2001/XMLSchema}string"/>
30   *         &lt;element name="testInteger" type="{http://www.w3.org/2001/XMLSchema}integer"/>
31   *         &lt;element name="testInt" type="{http://www.w3.org/2001/XMLSchema}int"/>
```

```
32  *          &lt;element name="testLong" type="{http://www.w3.org/2001/XMLSchema}long"/>
33  *          &lt;element name="testShort" type="{http://www.w3.org/2001/XMLSchema}short"/>
34  *          &lt;element name="testDecimal" type="{http://www.w3.org/2001/XMLSchema}decimal"/>
35  *          &lt;element name="testFloat" type="{http://www.w3.org/2001/XMLSchema}float"/>
36  *          &lt;element name="testDouble" type="{http://www.w3.org/2001/XMLSchema}double"/>
37  *          &lt;element name="testBoolean" type="{http://www.w3.org/2001/XMLSchema}boolean"/>
38  *          &lt;element name="testByte" type="{http://www.w3.org/2001/XMLSchema}byte"/>
39  *          &lt;element name="testQName" type="{http://www.w3.org/2001/XMLSchema}QName"/>
40  *          &lt;element name="testDateTime" type="{http://www.w3.org/2001/XMLSchema}dateTime"/>
41  *          &lt;element name="testBase64Binary" type="{http://www.w3.org/2001/XMLSchema}➥
        base64Binary"/>
42  *          &lt;element name="testHexBinary" type="{http://www.w3.org/2001/XMLSchema}hexBinary"/>
43  *          &lt;element name="testUnsignedInt" type="{http://www.w3.org/2001/XMLSchema}➥
        unsignedInt"/>
44  *          &lt;element name="testUnsignedShort" type="{http://www.w3.org/2001/XMLSchema}➥
        unsignedShort"/>
45  *          &lt;element name="testUnsignedByte2" type="{http://www.w3.org/2001/XMLSchema}➥
        unsignedByte"/>
46  *          &lt;element name="testTime" type="{http://www.w3.org/2001/XMLSchema}time"/>
47  *          &lt;element name="testDate" type="{http://www.w3.org/2001/XMLSchema}date"/>
48  *          &lt;element name="testAnySimpleType" type="{http://www.w3.org/2001/XMLSchema}➥
        anySimpleType"/>
49  *          &lt;element name="testDuration" type="{http://www.w3.org/2001/XMLSchema}duration"/>
50  *        &lt;/sequence>
51  *      &lt;/restriction>
52  *    &lt;/complexContent>
53  *  &lt;/complexType>
54  *  </pre>
55  *
56  *
57  */
58 @XmlAccessorType(XmlAccessType.FIELD)
59 @XmlType(name = "rowType", propOrder = {
60     "testString",
61     "testInteger",
62     "testInt",
63     "testLong",
64     "testShort",
65     "testDecimal",
66     "testFloat",
67     "testDouble",
68     "testBoolean",
69     "testByte",
70     "testQName",
71     "testDateTime",
72     "testBase64Binary",
73     "testHexBinary",
74     "testUnsignedInt",
75     "testUnsignedShort",
76     "testUnsignedByte2",
77     "testTime",
78     "testDate",
79     "testAnySimpleType",
80     "testDuration"
81 })
82 public class RowType
83 {
84     @XmlElement(required = true)
85     protected String testString;
86     @XmlElement(required = true)
87     protected BigInteger testInteger;
88     protected int testInt;
89     protected long testLong;
90     protected short testShort;
91     @XmlElement(required = true)
```

```
92      protected BigDecimal testDecimal;
93      protected float testFloat;
94      protected double testDouble;
95      protected boolean testBoolean;
96      protected byte testByte;
97      @XmlElement(required = true)
98      protected QName testQName;
99      @XmlElement(required = true)
100     @XmlSchemaType(name = "dateTime")
101     protected XMLGregorianCalendar testDateTime;
102     @XmlElement(required = true)
103     protected byte[] testBase64Binary;
104     @XmlElement(required = true, type = String.class)
105     @XmlJavaTypeAdapter(HexBinaryAdapter.class)
106     @XmlSchemaType(name = "hexBinary")
107     protected byte[] testHexBinary;
108     @XmlSchemaType(name = "unsignedInt")
109     protected long testUnsignedInt;
110     @XmlSchemaType(name = "unsignedShort")
111     protected int testUnsignedShort;
112     @XmlSchemaType(name = "unsignedByte")
113     protected short testUnsignedByte2;
114     @XmlElement(required = true)
115     @XmlSchemaType(name = "time")
116     protected XMLGregorianCalendar testTime;
117     @XmlElement(required = true)
118     @XmlSchemaType(name = "date")
119     protected XMLGregorianCalendar testDate;
120     @XmlElement(required = true)
121     @XmlSchemaType(name = "anySimpleType")
122     protected Object testAnySimpleType;
123     @XmlElement(required = true)
124     protected Duration testDuration;
125 ...
126 setter és getter metódusok
127 ...
128 }
```

A létrehozott Java osztályok közül most csak a *ControllType* és *RowType* egy-egy részlete került be a fenti programlistákba.A további generált Java forrásfájlok ezek:

- *DataType.java* a *dataType complexType* részére
- *SelectType.java* a *selectType complexType* részére
- *ObjectFactory* kényelmi osztály

Figyeljük meg a *RowType* tanulmányozásával, hogy az egyes XML séma típusok milyen Java típusokra képződtek le. Az *@XmlElement required = true* attribútuma azt jelenti, hogy ez a mező kötelezően megjelenő az XML, ami azért van, mert itt nem adtuk meg a *minOccurs és maxOccurs* paramétereket, tehát az előfordulás értéke 1. További érdekesség, hogy az *@XmlType* annotációnál létrejött a *propOrder*-t is, ami felsorolásként tartalmazza azt a tag sorrendet, amit az érvényes XML-nek prezentálnia kell. Ez nem véletlen, hiszen ez az XSD *xs:sequence* részénél felsorolt elemeket jelenti.

Dinamikus XML feldolgozás

Gyakori feladat, hogy különféle XML szövegek érkeznek, amiket a típusuktól függően kell feldolgoznunk. Kommunikációs, integrációs környezetekben például egy sorba (Queue) érkezhetnek ezek az XML üzenetek, amiket más-más módon kell kezelnünk. Ebben a pontban azt mutatjuk

meg, hogy az ilyen típusú feladatokat milyen módon tudjuk kezelni JAXB környezetben. Példaként tekintsük az 1-19. és 1-21. Programlistákat, amikhez tartozó XSD-ket is láthatjuk (1-20. és 1-22. Programlisták).

```
// 1-19. Programlista: ClassFirst.java

package org.cs.jaxb.test;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.➥
    XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(namespace = "org.cs.jaxb.test"➥
    )
@XmlType(namespace = "org.cs.jaxb.test")
public class ClassFirst
{
    @XmlElement
    String name;

    @XmlElement
    int value;
}
```

```
// 1-20. Programlista: ClassFirst.xsd

<?xml version="1.0" encoding="UTF-8" ➥
    standalone="yes"?>
<xs:schema version="1.0" targetNamespace="org.➥
    cs.jaxb.test" xmlns:tns="org.cs.jaxb.test"➥
     xmlns:xs="http://www.w3.org/

  <xs:element name="classFirst" type="➥
       tns:classFirst"/>

  <xs:complexType name="classFirst">
    <xs:sequence>
      <xs:element name="name" type="xs:string"➥
           minOccurs="0"/>
      <xs:element name="value" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```
// 1-21. Programlista: ClassSecond.java

package org.cs.jaxb.test;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.➥
    XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(namespace = "org.cs.jaxb.test"➥
    )
@XmlType(namespace = "org.cs.jaxb.test")
public class ClassSecond
{
    @XmlElement
    String description;

    @XmlElement
    String title;

    @XmlElement
    int pages;
}
```

```
// 1-22. Programlista: ClassSecond.xsd

<?xml version="1.0" encoding="UTF-8" ➥
    standalone="yes"?>
<xs:schema version="1.0" targetNamespace="org.➥
    cs.jaxb.test" xmlns:tns="org.cs.jaxb.test"➥
     xmlns:xs="http://www.w3.org/

  <xs:element name="classSecond" type="➥
       tns:classSecond"/>

  <xs:complexType name="classSecond">
    <xs:sequence>
      <xs:element name="description" type="➥
           xs:string" minOccurs="0"/>
      <xs:element name="title" type="xs:string➥
           " minOccurs="0"/>
      <xs:element name="pages" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

A *GenXML* osztály (1-23. Programlista) segítségével legyártottuk a *ClassFirst.xml* és *ClassSecond.xml* XML fájlokat, ezek lesznek a minta üzenetek, amiket különféle módon kell feldolgozni. A 15-22 sorok között létrehozzunk egy-egy *ClassFirst* és *ClassSecond* objektumot. A 27. sor nagyon fontos, mert ez egy olyan *Class[]* tömböt hoz létre, ami azokat az osztályokat sorolja fel, amiket majd a JAXB kontextusban kezelni, ismerni akarunk. A 29. sor ennek megfelelően hoz létre egy *JAXBContext* objektumot, amitől a következő sorban egy *Marshaller* objektumot igénylünk. A 32-33 sorokban mindkét előzetesen létrehozott objektumunkat XML fájlba perzisztáljuk. Most, hogy van 2 különböző XML üzenetünk, elérkeztünk oda, hogy be tudjuk mutatni az azok egyedi kezelését megvalósító *XMLProcessor* osztályt (1-24. Programlista). Nézzük meg röviden a működését! A 14-23 sorok jelentése már bizonyára mindenkinek világos, ezért a 25. sorban lévő *unmarshal()* metódushívásra hívnánk fel először a figyelmet, ahol a beolvasott XML egy általános *Object* sta-

tikus típusú objektumba kerül. A trükk igazából csak ennyi, mert utána az egyes elágazásokban az *instanceof* utasításra épülő döntés már segít a konkrét esetek szétválasztásában és feldolgozásában.

```
1  // 1-23. Programlista: GenXML.java
2
3  package org.cs.jaxb.test;
4
5  import java.io.File;
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Marshaller;
9  import test.ceg.org.Employee;
10
11 public class GenXML
12 {
13     public static void main(String[] args) throws JAXBException
14     {
15         ClassFirst cf = new ClassFirst();
16         cf.name = "Nyiri Imre";
17         cf.value = 5;
18
19         ClassSecond cs = new ClassSecond();
20         cs.description = "Ez egy jó lecke volt.";
21         cs.pages = 100;
22         cs.title = "Informatikai Navigátor";
23
24         File xmlfile1 = new File("/home/tanulas/xml/ClassFirst.xml");
25         File xmlfile2 = new File("/home/tanulas/xml/ClassSecond.xml");
26
27         Class[] classes = new Class[] {ClassFirst.class, ClassSecond.class};
28
29         JAXBContext jaxbContext = JAXBContext.newInstance( classes );
30         Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
31         jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
32         jaxbMarshaller.marshal( cf, xmlfile1 );
33         jaxbMarshaller.marshal( cs, xmlfile2 );
34     }
35 }
```

```
1  // 1-24. Programlista: XMLProcessor.java
2
3  package org.cs.jaxb.test;
4
5  import java.io.File;
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Unmarshaller;
9
10 public class XMLProcessor
11 {
12     public static void main(String[] args) throws JAXBException
13     {
14         Class[] classes = new Class[]
15         {
16             org.cs.jaxb.test.ClassFirst.class,
17             org.cs.jaxb.test.ClassSecond.class
18         };
19
20         JAXBContext jaxbContext = JAXBContext.newInstance( classes );
21         Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
22
23         File xmlfile = new File( "/home/tanulas/xml/ClassSecond.xml" );
```

```
24
25          Object xmlObject = jaxbUnmarshaller.unmarshal( xmlfile );
26
27          if ( xmlObject instanceof ClassFirst )
28          {
29              ClassFirst cf = (ClassFirst)xmlObject;
30              System.out.println( cf.name );
31              System.out.println( cf.value );
32          }
33          else if ( xmlObject instanceof ClassSecond )
34          {
35              ClassSecond cf = (ClassSecond)xmlObject;
36              System.out.println( cf.description );
37              System.out.println( cf.pages );
38              System.out.println( cf.title );
39          }
40          else
41          {
42              System.out.println("Ismeretlen XML!");
43          }
44      }
45  }
```

Érdekes XML séma lehetőségek

Ebben a pontban az 1-16. Programlista sémáját néhány helyen megváltoztatjuk és tanulmányozzuk, hogy mindez milyen hatással van a generált Java osztályokra. Ez a módszer azért lesz tanulságos, mert azután mi is tudunk így annotált class-okat készíteni, azaz sokat tanulhatunk a JAXB „@”-okról.

Ismétlődések és XML tag számosságok

Eddig nem mutattuk meg, hogy milyen lett a generált *DataType* class, pedig lehet belőle tanulni, emiatt nézzük is meg! Amit meg kell figyelnünk, az a *protected List<RowType> row* sor (45. sor), ugyanis a *row* tag 0 vagy több ismétlődéssel lehet a séma szerint, amit ez a konstrukció jól leképez.

```
1   // 1-25. Programlista: DataType.java
2
3   //
4   // This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference ➥
        Implementation, v2.2.4-2
5   // See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
6   // Any modifications to this file will be lost upon recompilation of the source schema.
7   // Generated on: 2013.04.06 at 04:22:16 PM GMT
8   //
9
10
11  package cs.org.jaxb.test;
12
13  import java.util.ArrayList;
14  import java.util.List;
15  import javax.xml.bind.annotation.XmlAccessType;
16  import javax.xml.bind.annotation.XmlAccessorType;
17  import javax.xml.bind.annotation.XmlType;
18
19
20  /**
21   * <p>Java class for dataType complex type.
22   *
23   * <p>The following schema fragment specifies the expected content contained within this class.
24   *
25   * <pre>
```

```
26   * &lt;complexType name="dataType"&gt;
27   *   &lt;complexContent&gt;
28   *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType"&gt;
29   *       &lt;sequence maxOccurs="unbounded" minOccurs="0"&gt;
30   *         &lt;element name="row" type="{urn:jaxbtest/xsd2java}rowType"/&gt;
31   *       &lt;/sequence&gt;
32   *     &lt;/restriction&gt;
33   *   &lt;/complexContent&gt;
34   * &lt;/complexType&gt;
35   * </pre>
36   *
37   *
38   */
39  @XmlAccessorType(XmlAccessType.FIELD)
40  @XmlType(name = "dataType", propOrder = {
41      "row"
42  })
43  public class DataType {
44
45      protected List<RowType> row;
46
47      public List<RowType> getRow() {
48          if (row == null) {
49              row = new ArrayList<RowType>();
50          }
51          return this.row;
52      }
53  }
```

Mi történik, ha a *maxOccurs* részt töröljük és csak a *minOccurs="0"* deklarációt hagyjuk meg? Ekkor a generált változó – ahogy vártuk is – nem tömb lesz (*protected RowType row*):

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "dataType", propOrder = {
    "row"
})
public class DataType {

    protected RowType row;

    public RowType getRow() {
        return row;
    }

    public void setRow(RowType value) {
        this.row = value;
    }
}
```

Amennyiben a *minOccurs* részt is elhagyjuk, akkor a dataType sémarészlet így fog kinézni:

```
<xs:complexType name="dataType">
   <xs:sequence>
     <xs:element type="sch:rowType" name="row"➥
          xmlns:sch="urn:jaxbtest/xsd2java"/>
  </xs:sequence>
</xs:complexType>
```

Az erre generált *DataType* class pedig ismét egy új elemmel fog bővülni, azaz a *row* adattagnál megjelenik a *required = true* jelzés, ami azért van, mert a row előfordulása most pontosan 1, azaz ebben az értelemben egy kötelezően előforduló része az XML-nek:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "dataType", propOrder = {
    "row"
})
public class DataType {

    @XmlElement(required = true)
    protected RowType row;

    public RowType getRow() {
        return row;
    }

    public void setRow(RowType value) {
        this.row = value;
    }
}
```

Elem sorrend

Egy nyelvtan lényeges része, hogy a szavai milyen sorrendben következhetnek egymás után.

Az XML-ben is lényeges, hogy az egyes címkéknek mi a sorrendje, ezért az XML séma ezzel a kérdéskörrel részletesen foglalkozik. Nézzük például a *selectType* séma típust:

```
...
<xs:complexType name="selectType">
  <xs:sequence>
      <xs:element type="sch:controllType" name➥
          ="controll" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
      <xs:element type="sch:dataType" name="➥
          data" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
  </xs:sequence>
</xs:complexType>
...
```

Itt az *xs:sequence* egy szigorú sorrendet ír le, azaz a *controll* és *data* címkéknek a felsorolás sorrendjében kell az XML-ben is előfordulniuk. Ennek megfelelően a generált Java class is ezt írja elő, ami a *propOrder* résznél való felsorolásban tükröződik vissza.

```
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "selectType", propOrder = {
    "controll",
    "data"
})
public class SelectType {

    @XmlElement(required = true)
    protected ControllType controll;
    @XmlElement(required = true)
    protected DataType data;
...
}
```

Amennyiben az *xs:sequence* helyett *xs:all* megadást adunk a sémában, akkor ez azt jelenti, hogy ezeknek az elemeknek továbbra is elő kell fordulniuk, de most már mindegy a sorrendjük:

```
...
<xs:complexType name="selectType">
  <xs:all>
      <xs:element type="sch:controllType" name➥
          ="controll" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
      <xs:element type="sch:dataType" name="➥
          data" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
  </xs:all>
</xs:complexType>
...
```

Ez a generált Java osztályon is korrekt módon látszik, azaz a *propOrder* nem rendelkezik a sorrendről:

```
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "selectType", propOrder = {

})
public class SelectType {

    @XmlElement(required = true)
    protected ControllType controll;
    @XmlElement(required = true)
    protected DataType data;
...
```

Elem attribútum

Legyen az a feladatunk, hogy a *selectType* elemnek egy *lang* nevű attribútumot is tervezünk, ekkor a séma így módosul:

```
...
<xs:complexType name="selectType">
 <xs:sequence>
      <xs:element type="sch:controllType" name➥
          ="controll" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
      <xs:element type="sch:dataType" name="➥
          data" xmlns:sch="urn:jaxbtest/➥
          xsd2java"/>
  </xs:sequence>
  <xs:attribute name="lang" type="xs:string"/>
</xs:complexType>
...
```

A lefordított Java kód pedig így fogja ezt JAXB annotáció szinten tükrözni:

```
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "selectType", propOrder = {
    "controll",
    "data"
})
public class SelectType {

    @XmlElement(required = true)
    protected ControllType controll;
    @XmlElement(required = true)
    protected DataType data;
    @XmlAttribute(name = "lang")
    protected String lang;
...
```

A simpleType kódja

Egészítsük ki a *selectType-ot* egy *simpleType* típussal. A sémarészlet így változik:

```
...
   <xs:complexType name="selectType">
        <xs:sequence>
            <xs:element type="sch:controllType➥
                " name="controll" xmlns:sch="➥
                urn:jaxbtest/xsd2java"/>
            <xs:element type="sch:dataType" ➥
                name="data" xmlns:sch="urn:➥
                jaxbtest/xsd2java"/>
            <xs:element type="sch:NumberType" ➥
                name="number" xmlns:sch="urn:➥
                jaxbtest/xsd2java"/>
        </xs:sequence>
        <xs:attribute name="lang" type="xs:➥
            string"/>
    </xs:complexType>

    <xs:simpleType name="NumberType">
      <xs:restriction base="xs:string">
        <xs:pattern value="d{5}" />
      </xs:restriction>
    </xs:simpleType>
...
```

A létrehozott Java kód:

```
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "selectType", propOrder = {
    "controll",
    "data",
    "number"
})
public class SelectType {

    @XmlElement(required = true)
    protected ControllType controll;
    @XmlElement(required = true)
    protected DataType data;
    @XmlElement(required = true)
    protected String number;
    @XmlAttribute(name = "lang")
    protected String lang;
...
```

Az XML séma egy elég kiterjedt szabvány, ezért most a kísérleteket abbahagyjuk, de ezzel a módszerrel bármilyen XSD konstrukcióról fel lehet deríteni, hogy milyen JAXB annotált osztály tartozik hozzá.

A JAXB annotációk áttekintése

A most következő pontban rendszerezetten áttekintjük a JAXB annotációkat. Az előzőekben inkább az XML séma oldaláról kísérletezve derítettük ki ezeket a lehetőségeket, most pedig a dokumentáció alapján fogjuk áttekinteni, kiegészíteni a már eddig megismert tudást. A JAXB alapfeladata, hogy bármely Java osztály objektumát XML szöveggé alakítson (szerializáció), illetve fordítva, bármely XML szövegből egy őt reprezentálni képes objektumot állítson elő (deszerializáció). Ehhez alapvetően a *javax.xml.bind.JAXBElement* class használatos, ami képes bármilyen Java objektumot becsomagolni. A szerializálandó osztályt pedig a *javax.xml.bind.annotation.XmlRootElement* annotációval kell megjelölni.

Annotáció nélkül...

Alaphasználat során nem is kell „@”-okat elhelyezni az osztályra, nézzük ugyanis az ismert 2 osztályunkat ebben a formában! Láthatjuk, hogy most nincs semmilyen jelölés sem az *Address*, sem az *Employee* osztályon. Ugyanakkor fontos megjegyezni, hogy az objektumok állapotát mindig a publikus jellemzőik (adattagjaik) jelentik, így az szerializációba bevonni kívánt adatmezőknek vagy publikusnak kell lenniük, vagy biztosítani kell a megfelelő getter metódusokat.

Ez lesz a mostani példában használt *Employee*:

```
// Employee.java "@" nélkül

package org.cs.test;

public class Employee
{
    Address address;
    String name;
    String sex;
    int kor;
    double suly;

    public Address getAddress()
    {
        return address;
    }

    public void setAddress(Address address)
    {
        this.address = address;
    }
...
}
```

És a beágyazott *Address* class:

```
// Address.java "@" nélkül

package org.cs.test;

public class Address
{
    String street;
    int houseNumber;

    public String getStreet()
    {
        return street;
    }

    public void setStreet(String street)
    {
        this.street = street;
    }
...
}
```

Az *Employee* egy példányának XML szerializációját mutatja a 1-26. Programlista. Nincs annotáció, de mégis tudunk olyan kódot írni, ami a 26-32 sorok közötti e *Employee* objektumot XML-be alakítja. A 34-36 sorok más ismerősek. A 38. sor jelenti az újdonságot, ahol létrehozunk egy *Employee* class-ra vonatkozó *JAXBElement* példányt, ahol a konstruktor ezeket a paramétereket kapja:

- A gyökérelem neve az XML eredményben. Ezt egy minősített *QName* osztály objektum segítségével adhattuk meg.
- A szerializálandó class neve (*Employee*)
- A konkrét szerializálandó objektum neve (*e*)

A további sorok már egyértelműek, a 41. sorban a képernyőre ki is íratjuk a kapott XML-t. Megjegyezzük, hogy ez a módszer azért hasznos, mert általában nincsenek a Java class-ok JAXB szerint annotálva, de mégis gyorsan hozzájuthatunk az objektum egy XML reprezentációjához, ami sok helyen hasznos lehet.

```
1  // 1-26. Programlista: TestNoAnnotation.java
2
3  package org.cs.test;
4
5  import java.io.File;
6  import java.io.StringReader;
7  import java.io.StringWriter;
8  import javax.xml.bind.JAXBContext;
9  import javax.xml.bind.JAXBElement;
10 import javax.xml.bind.JAXBException;
11 import javax.xml.bind.Marshaller;
12 import javax.xml.bind.Unmarshaller;
13 import javax.xml.namespace.QName;
14
15 public class TestNoAnnotation
16 {
17     public static void main(String[] args)
18     {
19         System.out.println("Start...");
20
21         try
22         {
23             Address a = new Address();
24             a.houseNumber = 12;
25             a.street = "Váci út";
26
27             Employee e = new Employee();
28             e.address = a;
29             e.kor = 32;
30             e.name = "Alma Ferenc";
31             e.sex = "férfi";
32             e.suly = 88;
33
34             JAXBContext jc = JAXBContext.newInstance(org.cs.test.Employee.class);
```

```
35              Marshaller jaxbMarshaller = jc.createMarshaller();
36              jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
37
38              JAXBElement<Employee> root = new JAXBElement<Employee>(new QName("employee"), ➥
                    Employee.class, e);
39              StringWriter sw = new StringWriter();
40              jaxbMarshaller.marshal(root, sw);
41              System.out.println( sw );
42          } catch (JAXBException e)
43          {
44              e.printStackTrace();
45          }
46      }
47  }
```

JAXB annotációk

Az annotációkat általában a class vagy az adattag szintjén adhatjuk meg és ezzel teljes rugalmassággal adhatjuk meg, hogy milyen XML outputot szeretnénk előállítani. Ezek a jelölések a *javax.xml.bind.annotation* csomagban kaptak helyet a Java SDK-ban. A rövid ismertetésben – ahogy eddig is – a Java Bean property és az adattag ugyanazt jelenti.

- *@XmlRootElement*: Amennyiben egy osztályt XML szerializálni szeretnénk, úgy a class deklaráció elé kell tenni, utalva arra, hogy ez az az osztály, amit XML szöveggé akarunk alakítani.

- *@XmlElement*: Egy XML tag és egy Java property összerendelése.

- *@XmlAttribute*: Amennyiben egy Java class adattagját ezzel annotálunk, úgy ezzel azt mondjuk meg, hogy ez egy XML attribútum értékét fogja tárolni, azaz azzal lesz mappelve. Az XML property – ahogy azt bizonyára sokan tudják – a címke egy jellemzője, azaz nem a nyitó és záró tag közötti érték.

- *@XmlType*: Amennyiben egy osztályt ezzel jelölünk meg, akkor az az XML sémában egy complex típus lesz. A példáinkban láthattuk a használatát. Kiemeljük, hogy például az XML címkék sorrendjét is ezen keresztül tudtuk megadni.

- *@XmlID*: Kizárólag csak egy *String* típusú adattag jelölhető meg ezzel, ami kulcsként fog szerepelni. Az *@XmlIDREF* segítségével hivatkozhatunk rá.

- *@XmlIDREF*: Kizárólag csak adattag jelölhető meg ezzel, amely tag maga is egy osztály, amit a szerializálás része. Amennyiben egy *@XmlID* azonosítóra hivatkozunk itt, akkor nem az egész típus, hanem csak a kulcsként kijelölt mező fog szerializálódni.

- *@XmlTransient*: Ezek egy explicite módon való deklarációt tudunk megadni egy bean property-re vonatkozólag, hogy azt nem szeretnénk az előállított XML szövegben látni.

- *@XmlValue*: Ez is csak adattagra adható meg és egy XML elem tartalmat reprezentál.

- *@XmlJavaTypeAdapter*: Megadhatunk egy Java class-t, ami pontosan megvalósít egy Java↔XML leképzést. Használatára a későbbiekben egy részletes példát is mutatunk.

- *@XmlElementWrapper*: Mindenütt használható, ahol a *@XmlElement* is. Hasz-

nálatára külön példát mutatunk.

- *@XmlAccessorType*: A legfelsőbb szintű osztályra annotálhatjuk ezt a lehetőséget, amivel azt lehet szabályozni, hogy általában milyen osztály adattagok kerüljenek a szerializációba. Alapértelmezésben a *public* tagok szerializálva lesznek. Viszonylag ritkán használjuk, de az *xjc* általában mindig legenerálja ezt az annotációt is.

Az adatmodell

Az annotációk segítségével nagy pontossággal tudjuk az XML és Java szerkezeteket, neveket (és névtereket) összerendelni (binding). Amikor egy Java class-t vagy XML sémát tervezünk, akkor az egy adatmodellezés. Az új elem az, hogy itt egyszerre 2 modellező nyelvben is gondolkodunk közben:

- XML séma (XML)
- Java osztályok és a közöttük lévő kapcsolatok (Java)

Ugyanakkor a 2 modell között tudatosan nem képezünk le mindent a másikra. Az *@XmlTransient* például pont azt jelentette, hogy az így megcímkézett adatmező nem része az XML séma alapú modellnek. Az *@XmlID* és *@XmlIDREF* használatára egy konkrét példa lehet egy olyan *MyPoint* class, aminek a következő 3 adattagja van:

```
// MyPoint.class

package org.cs.test;

@XmlRootElement
public class MyPoint
{
    public int x;
    public int y;

    @XmlID
    public String key;
}
```

Láthatjuk, hogy a *String* típusú és *key* nevű adatmezőt kulcsnak jelöltük. A szerializálandó osztály pedig legyen ez:

```
// MyPoint.class

package org.cs.test;

@XmlRootElement
public class Pontok
{
    @XmlElement
    public MyPoint pA;

    @XmlIDREF
    public MyPoint pB;
}
```

Ekkor a *pA* pont teljes egészében szerializálódik az output XML-ben, azonban a *pB* pontnak csak a *key* adattagja, mert ott ezt kértük, azaz elég a kulcs értéke az XML szövegbe.

A @XmlElementWrapper

A gyűjteményeknél fontos kérdés, hogy az elemeket csak magukban tüntetjük-e fel, vagy egy összefogó külső XML tag is utal az egész adatszerkezetre. Ennek könnyebb megértésére egy kis példa szolgálhat. Nézzük a szerializálandó *ClassThird* osztályt (1-27. Programlista), ahol a *@XmlElementWrapper* egyelőre megjegyzésbe került. A *GenThirdXML* (1-28. Programlista) osztályt futtassuk le, ez legyártja a szerializált XML-t, amit a 1-29. Programlistán tekinthetünk meg.

```
1  // 1-27. Programlista: ClassThird.java
2
3  package org.cs.jaxb.test;
4
5  import javax.xml.bind.annotation.XmlElement;
6  import javax.xml.bind.annotation.XmlElementWrapper;
7  import javax.xml.bind.annotation.XmlRootElement;
8  import javax.xml.bind.annotation.XmlType;
```

```
9
10 @XmlRootElement(namespace = "org.cs.jaxb.test")
11 @XmlType(namespace = "org.cs.jaxb.test")
12 public class ClassThird
13 {
14     @XmlElement
15     String name;
16
17     //@XmlElementWrapper(name="kategoriak")
18     @XmlElement
19     int[] value;
20 }
```

```
1  // 1-28. Programlista: GenThirdXML.java
2
3  package org.cs.jaxb.test;
4
5  import java.io.File;
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Marshaller;
9
10 public class GenThirdXML
11 {
12     public static void main(String[] args) throws JAXBException
13     {
14         ClassThird ct = new ClassThird();
15         ct.name = "Korcsoportok";
16
17         ct.value = new int[3];
18         ct.value[0] = 12;
19         ct.value[1] = 16;
20         ct.value[2] = 18;
21
22         File xmlFile = new File("/home/tanulas/xml/ClassThird.xml");
23
24         Class[] classes = new Class[] {ClassThird.class};
25
26         JAXBContext jaxbContext = JAXBContext.newInstance( classes );
27         Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
28         jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
29         jaxbMarshaller.marshal( ct, xmlFile );
30     }
31 }
```

Ezen az eredmény XML szövegen semmi meglepő nincs, mindent az eddig is ismert eszközökkel csináltunk. Az egyedüli jelenség, amit érdemes észrevenni az, hogy az *int[] value* adattag a *value* XML tag-ek sorozatára képződött le. Természetesen ezt nem csak tömbre, hanem listára vagy bármely más kollekció típusra is így használhatjuk.

```
// 1-29. Programlista

<?xml version="1.0" encoding="UTF-8" ➥
    standalone="yes"?>
<ns2:classThird xmlns:ns2="org.cs.jaxb.test">
    <name>Korcsoportok</name>
    <value>12</value>
    <value>16</value>
    <value>18</value>
</ns2:classThird>
```

Sok esetben praktikus vagy csak egyszerűen áttekinthetőbb, ha ezeket a *value* tag-eket egy őket befogadó XML címkébe ágyazzuk, ahogy azt az 1-30. Programlistán észrevehető *kategoriak* címke is teszi. Ezt úgy tudtuk megvalósítani, hogy a *ClassThird* class 17. sorából kivesszük a megjegyzést és hagyjuk működni a *@XmlElementWrapper* annotációt a *va-*

lue property-re, ahol azt is specifikáltuk, hogy a becsomagoló XML tag neve *kategoriak* legyen.

```
// 1-30. Programlista

<?xml version="1.0" encoding="UTF-8" ➥
    standalone="yes"?>
<ns2:classThird xmlns:ns2="org.cs.jaxb.test">
    <name>Korcsoportok</name>
    <kategoriak>
        <value>12</value>
        <value>16</value>
        <value>18</value>
    </kategoriak>
</ns2:classThird>
```

Adapter készítése

Korábbi programozási tapasztalataink alapján tudjuk, hogy adaptert mindig olyankor készítünk, amikor valamilyen 2 dolog nem illik össze, de azt mégis együtt szeretnénk használni. Erről többet is olvashatunk az Informatikai Navigátor 7. számának 10. cikkéből. Tekintsük a következő *Address* class változatot, aminek a neve *AddressWithMap*. Itt a cím adatelemeit egy *Map* adatszerkezetben tároljuk:

```
// AddressWithMap.java

package cs.org.jaxb.test.adapter;

import java.util.HashMap;
import java.util.Map;

public class AddressWithMap
{
    // keys: street, houseNumber
    public Map addressMap = new HashMap();
}
```

Ennek megfelelően az *Employee* class helyett is van egy másik hasonló, az *EmployeeOther*:

```
// EmployeeOther.java

package cs.org.jaxb.test.adapter;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.➥
    XmlRootElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.➥
    XmlJavaTypeAdapter;

@XmlRootElement(namespace = "org.ceg.test")
@XmlType(namespace = "org.ceg.test")
public class EmployeeOther
{
    @XmlElement
    @XmlJavaTypeAdapter(value=AddressAdapter.➥
        class)
    public AddressWithMap address;

    @XmlElement(namespace = "org.ceg.test")
    public String name;

    @XmlElement
    public String sex;

    @XmlElement
    public int kor;

    @XmlElement
    public double suly;
}
```

Vegyük észre, hogy most az *address* adattagot egy *Map*-ben tároljuk, aminek nincs alapértelmezett XML szerializációja, pedig mi azt szeretnénk. Sőt az is jó lenne, ha kompatibilis maradna az eddigi, korábbi XML szerkezettel. A megoldás az, hogy az *AddressWithMap* típusú *address* jellemzőre definiálunk egy adapter funkciót betöltő Java class-t, amit a *@XmlJavaTypeAdapter* jelöléssel adhatunk meg és a *value* attribútum értéke maga az adapter osztály neve, ami esetünkben *AddressAdapter* lesz. Ennek a mágikus osztálynak azt a feladatot fogjuk adni, hogy minden olyan értéket, ami egy *AddressWithMap* osztályból származik (esetünkben az *EmployeeOther* class *address* tagja) képezzen le a már jól ismert *Address* osztályunkra, majd bízzuk arra a szerializálást. Nézzük meg az *Address* osztályt újra:

```
// Address.java

package cs.org.jaxb.test.adapter;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlType(namespace = "org.ceg.test")
public class Address
{
    @XmlElement
    public String street;

    @XmlElement
    public int houseNumber;
}
```

Ennyi előzmény után már biztos mindenki kíváncsi magára az adapterre, az *AddressAdapter* class-ra (1-31. Programlista). Ez egy olyan *XmlAdapter* ősosztályt kiterjesztő osztály, aminek 2 metódusa van:

- *marshal()* → Leképezi a szerializálandó objektumot arra az objektumra, amit már tudunk szerializálni
- *unmarshal()* → Az XML-ből visszaolvasásra képes objektumból legyártja azt az objektumot, amit használunk, azaz a deszerializációját visszavezettük az első objektumra.

Látható, hogy az öröklésnél mindig konkretizálni kell (AddressAdapter.java, 7. sor) az éppen használt un. *BoundType* és *ValueType* osztályokat, amiket meg is tettünk ezen 2 osztály használatával: *Address*, *AddressWithMap*. A *marshal()* és *unmarshal()* kódja egyértelmű és egyszerű, megadja az *AddressWithMap* ↔ *Address* oda-vissza konvertálás algoritmusát. Az 1-32. Programlista bemutatja, hogy ezek után egy *EmployeeOther* objektumot milyen módon szerializálhatunk XML-be. Semmi különleges nincs benne, csak a teljesség kedvéért mutatjuk meg. A futtatás során előállított *EmployeeOther.xml* fájlt (1-33. Programlista) megtekintve látható, hogy a kívánt eredményt kaptuk, az adapter a tervnek megfelelően működött.

```
1  // 1-31. Programlista: AddressAdapter.java
2
3  package cs.org.jaxb.test.adapter;
4
5  import javax.xml.bind.annotation.adapters.XmlAdapter;
6
7  public class AddressAdapter extends XmlAdapter<Address, AddressWithMap>
8  {
9      @Override
10     public AddressWithMap unmarshal(Address vt) throws Exception
11     {
12         AddressWithMap am = new AddressWithMap();
13         am.addressMap.put("street", vt.street);
14         am.addressMap.put("houseNumber", vt.houseNumber);
15         return am;
16     }
17
18     @Override
19     public Address marshal(AddressWithMap bt) throws Exception
20     {
21         Address a = new Address();
22         a.street = (String) bt.addressMap.get("street");
23         a.houseNumber = Integer.parseInt((String) bt.addressMap.get("houseNumber"));
24         return a;
25     }
26 }
```

```
1  // 1-32. Programlista: TestJAXBAdapter.java
2
3  package cs.org.jaxb.test.adapter;
4
5  import java.io.File;
6  import javax.xml.bind.JAXBContext;
7  import javax.xml.bind.JAXBException;
8  import javax.xml.bind.Marshaller;
9
10 public class TestJAXBAdapter
11 {
12     public static void main(String[] args) throws JAXBException
```

```
13      {
14          AddressWithMap a = new AddressWithMap();
15          a.addressMap.put("street", "Budapest, Görgey Artúr utca");
16          a.addressMap.put("houseNumber", "102");
17
18          EmployeeOther e = new EmployeeOther();
19          e.address = a;
20          e.kor = 32;
21          e.name = "Alma Ferenc";
22          e.sex = "férfi";
23          e.suly = 88;
24
25          JAXBContext jaxbContext = JAXBContext.newInstance(cs.org.jaxb.test.adapter.EmployeeOther
                .class);
26          Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
27          jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
28          File XMLfile = new File("/home/tanulas/xml/EmployeeOther.xml");
29          jaxbMarshaller.marshal(e, XMLfile);
30      }
31  }
```

```
// 1-33. Programlista: EmployeeOther.xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:employeeOther xmlns:ns2="org.ceg.test">
    <address>
        <street>Budapest, Görgey Artúr utca</street>
        <houseNumber>102</houseNumber>
    </address>
    <ns2:name>Alma Ferenc</ns2:name>
    <sex>férfi</sex>
    <kor>32</kor>
    <suly>88.0</suly>
</ns2:employeeOther>
```

JAXB alapú séma validáció

XML validáció alatt azt értjük, hogy van egy XML szöveg és egy XML séma (vagy bármely más módon megadott grammatika), aminek az előírásaihoz való illeszkedését vizsgáljuk a kérdéses XML szövegnek. A JAXB alapú séma validálás módját az Informatikai Navigátor 4. számának 9.2 pontjánál már ismertettük, akinek ez szükséges, ott átnézheti.

Összefoglalás

A JAXB API a Java 6 óta része az alap SDK-nak, ezért mindenütt érdemes használni, ahol fontos az XML sémára épülő precíz XML kezelés. Az is előnye ennek az eszközkészletnek, hogy a programozó képes Java objektumokkal kezelni az XML adatstruktúrákat, ami sokszor sokkal áttekinthetőbb, mint a *SAX* vagy *DOM* alapú feldolgozás. A JAXB alapszintű megértése azért is fontos, mert a korszerű Java webservice architektúra (*JAX-WS*) a Java és XML binding-hoz természetes módon ezt használja és bármikor szükségünk lehet a szerializációt kicsi vagy ritkábban akár nagyobb mértékben is testre szabni. Befejezésül fontosnak tartjuk megemlíteni, hogy több éve rendelkezésre áll egy másik API is, az *XMLBeans* (`http://xmlbeans.apache.org/`), ami jóval megelőzte időben a JAXB megoldásait, de képesség tekintetében szintén tudja a fenti szolgáltatásokat is.

2. BeanShell - Könnyűsúlyú scriptek Java nyelvbe ágyazva

A fordított (*compiler*) és értelmezett (*interpreter*) környezetek mindig egy kettősséget jelentettek a számítógépes nyelvek világában. Mindkét megközelítésnek vannak előnyei és hátrányai, ezért gyakori, hogy egy erősen típusos nyelvet kiegészítenek egy interpreter alapú elemmel. A *BeanShell* (és a másik elterjedt nyelv a *Groovy*) pontosan ilyen. Az érdekességet többek között az a lehetőség jelenti, hogy a BeanShell scripteket Java környeztből is tudjuk futtatni, amivel hatékony scriptnyelv kiegészítéshez jutunk.

Mi a BeanShell?

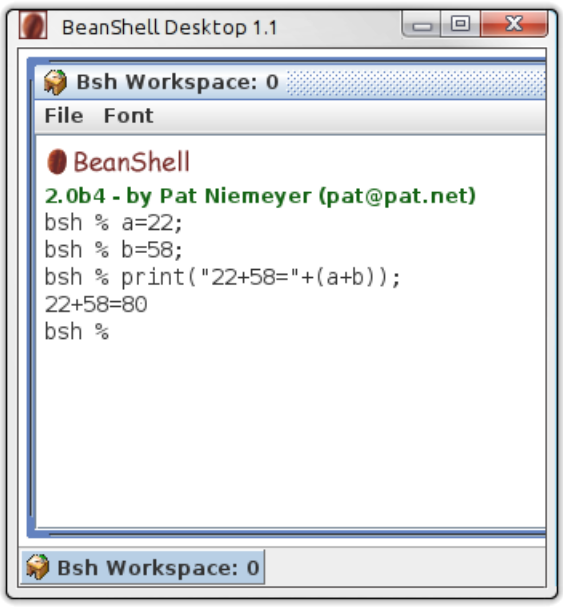
A *BeanShell* egy olyan script nyelv, amit a Java virtuális gép tud futtatni, ugyanis annak értelmezőjét a Javaban megvalósított *bsh.Interpreter* osztály valósítja meg. A webhelye: `http://www.beanshell.org/`. Nem célunk részletesen elemezni az értelmezett és fordított nyelvek előnyeit és hátrányait, de 2 jellemvonást mégis szeretnénk kiemelni, amivel biztosan érdekes lehetőséggel egészíthetjük ki Java programjainkat:

1. A BeanShell scriptek a Java program futása közben is használhatóak, így az őket reprezentáló programszövegek végrehajthatóak. Ugyanakkor ezek a szövegek dinamikusan is előállíthatók előtte, ami érdekes algoritmus szótárak kialakítását teszi lehetővé. Ez a gyakorlatban azt is jelentheti, hogy az adatkonfiguráció mellett algoritmikus konfigurációkat is rendelhetünk a programunkhoz.

2. A script nyelvek lazábban kezelik az egyébként nagyon fontos típus fogalmat, de pont ez a tulajdonságuk lehetővé teszi a használatukat ott, ahol csak 2-3 soros programtöredékre van szükség.

Írjuk be a parancsértelmezőbe az alábbi utasítást, miután megjelenik a 2.1. ábrán mutatott grafikus parancsértelmező.

```
java -cp /home/java/bean-shell/bsh-2.0b4.jar bsh.➥
    Console
```



2.1. ábra: BeanShell console

Ennek van egy szöveges alapú testvére is, amit ezzel a paranccsal tudunk elindítani (az *exit();* utasítással tudunk kilépni belőle):

```
java -cp bsh-2.0b4.jar bsh.Interpreter

BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh %
```

Az *Interpreter* class arra is használható, hogy egy textfájlban megírt *bsh* programot futtassunk. Legyen például ez a program, amit a *p1.bsh* fájlba mentettünk:

```
// 2-1. Programlista: p1.bsh

for (i=1; i<=10; i++)
{
  print(i);
}
```

```
Futtatás:
java -cp bsh-2.0b4.jar bsh.Interpreter p1.bsh
```

A program 1-10-ig kiírja az egész számokat a képernyőre.

Alapvető nyelvi elemek

A *BeanShell* létrehozásának volt néhány célja, a nyelvi elemek kialakítását ezek nagymértékben befolyásolták:

- A Java kiegészítése szkriptelési lehetőségekkel.
- Platformfüggetlen shell környezet elérhetősége.
- Gyors unit tesztek elkészíthetősége.
- Nagyobb rendszerekbe való beágyazhatóság.

A Java szintaxis ismerete

Egy teljes Java programot nem feltétlenül tud futtatni a *BeanShell*, de ennek nem az az oka, hogy Java utasítások sorozatát ne tudná futtatni. Például az alábbi Java program gond nélkül fut:

```
System.out.println("Indulok...");
System.out.println(new Date());

int fact = 1;
for (int i = 1; i < 8; ++i)
{
  fact *= i;
  System.out.println( i + ".fact=" + fact );
}

A futási eredmény vége:
1.fact=1
2.fact=2
3.fact=6
4.fact=24
5.fact=120
6.fact=720
7.fact=5040
```

Itt az a szabály, hogy mindent használhatunk, amit egy metódus törzsében írnánk, sőt írhatunk metódusokat is, de a Java nyelv rendelkezik néhány olyan szerkezeti konstrukcióval, amit ez az eszköz (még) nem ismer.

Gyengített Java típusok használata

A Java nyelvet a *BeanShell* ismeri, így azokat a kiegészítéseket érdemes áttekinteni, amik ehhez képest jelentenek új elemeket. Az igazi újdonság a Java eszközeinek gyenge típusossággal való használata, amire egy példa a következő:

```
bsh % map = new HashMap();
bsh % map.put("alma", 12);
bsh % map.put("körte", 21);
bsh % print(map);
{alma=12, körte=21}
bsh %
```

Egy másik egyszerű példa:

```
print( System.currentTimeMillis() );
1367078180006
```

Végül az egyszerűsített, típus nélküli *for* ciklus:

```
for(i=1; i<=10; i++)
{
  print( i );
};
```

A típus elhagyásának lehetősége mindenütt lehetséges, például a kivételkezelés *catch* ágánál csak a változó nevét írhatjuk:

```
try
{
...
} catch ( e )
{
  print( "caught exception: "+e );
}
```

A változók hatóköre

Van 2 fontos szabály, amit mindig érdemes észben tartani, amikor *BeanShell* scriptet készítünk:

- A deklarált változók láthatósága olyan, mint Java-ban.
- A nem deklarált (más szóval: statikus típus nélküli) változók láthatóságát nem módosítják a *{}* blokkok.

Az alapvető viselkedés a Java nyelvre hasonlít, azonban a program szövege explicit módon nem olyan, hogy minden változó egy osztálynak lenne a tagja, így az ilyen változók hatóköréről érdemes néhány megállapítást tenni. A következő példa mutatja, hogy egy kapcsos zárójelek közötti változó a blokkra lokális, ha típusos. Ez a Java-val való kompatibilitás miatt szükséges. Amennyiben nem típusos (esetünkben az y), úgy a változó az őt létrehozó blokkon kívül is látható.

```
// kódblokk
{
  y = 2;      // Untyped variable assigned
  int x = 1; // Typed variable assigned
}
print( y ); // 2
print( x ); // Error! x is undefined.
```

A fenti jelenség az összes ilyen szituációra igaz, például a *for* indexváltozó elérhető a ciklusmag után is, amennyiben annak nem volt explicit típusmegadása.

A változók módosító szavai

Ismerek a Javaban a private, protected, public, *final*, transient, volatile, static kulcsszavak, ezeket a *BeanShell* elfogadja, de hatása csak a *final*-nak van. Ugyanakkor nem deklarált változóknál nem használható semmilyen módosítószó.

JavaBean szelektorok

Java-ban megtanultuk a *setter* és *getter* metódusok használatát, azonban a *BeanShell* mégis a hagyományos hivatkozást részesíti előnyben, azaz a '.' karaktert. Itt probléma lehet, ha egy *prop* nevű adattag létezik, mert akkor nem annak a setter/getter metódusa fog meghívódni, hanem magára az adattagra fogunk hivatkozni. Erre a *BeanShell* ezt a szintaxist vezette be:

```
// A setter:
obj.setProp( valami );

// Helyette:
obj{"prop"} = valami;

// Pedig ezt akartuk:
obj.prop = valami;
```

A getter metódusok használata hasonló.

Ciklus

A Java 1.5-ben bevezetett ciklus használható, sőt annak értelmezése tovább lett tágítva az alábbi típusokra:

- *Enumeration*
- *Vector*
- *String*
- *StringBuffer* (*StringBuilder*-re nem működik)
- *Collection*
- *Iterator*
- tömb

A következő példa egy *String* karakterein iterál végig és közéjük illeszt egy '@' jelet:

```
str="";
for(char c : "Alma").
{
  str += c + "@";
}
print(str);

Futási eredmény:
A@l@m@a@
```

Kiterjesztett elágazás

A *BeanShell switch* kulcsszava lehetővé teszik a script nyelveken szokásos „laza" elágazások megvalósítását, ami a gyakorlatban ezt jelenti, hogy a szelektor bármilyen objektum lehet. A feltétel teljesülését az *equals()* metódussal vizsgálja az *Interpreter*:

```
maiNap = new Date();
switch( maiNap )
{
   case mikulas:
        break;
   case karacsony: break;
   default:
}
```

Kivételkezelés

A Java nyelv módszerét ismeri, de itt a *catch* ágban megadhatunk típus nélküli objektumot is, ahogy a következő példa mutatja:

```
try { ... }
catch (e)
{
   print(e);
}
```

Importálás

Osztályok vagy csomagok használatára a Java nyelv *import* utasítása használható. Érdekes lehetőség az un. szuper import, aminek az a lényege, hogy a *classpath*-ról elérhető összes class automatikusan importálódik:

```
import *;
```

Néha felvetődhet a kérdés, hogy egy osztály melyik *jar* fájlban volt, ezért ki is lehet íratni a scriptben a *which()* függvény segítségével:

```
bsh % which( java.lang.String );
Jar: file:/usr/java/j2sdk1.4.0/jre/lib/rt.jar
```

Látni fogjuk, hogy a BeanShell rendelkezik azokkal a fontosabb, előre megírt parancsokkal, amik mindegyik shell részét szokták képezni. Ezek importálására egy külön függvényhívás szolgálhat:

```
importCommands("/bsh/commands");
```

Scriptben készített elemek

Metódusok

Egy metódus hasonlóan készíthető, mint Javaban. Nézzük az alábbi példát:

```
int addTwoNumbers( int a, int b )
{
   return a + b;
}
```

Ennek meghívása semmi különlegességet nem mutat:

```
sum = addTwoNumbers( 5, 7 );
```

A fenti összeadást megvalósító script részlet a Java szintaxist követ és típusok is vannak benne. Mindez persze típusok nélkül is megy:

```
addTwoNumbers( a, b )
{
   return a + b;
}
```

Ekkor azonban futás közben oldódik fel, hogy *a* és *b* típusától függően a *'+'* operátornak mi lesz a jelentése:

```
// A foo 3 lesz.
foo = add(1, 2);

// A foo New York lesz
foo = add("New", "␣York");
```

Látható, hogy típus nélküli esetben nem kell megadni a visszatérési típust, így bármilyent vissza is lehet adni. A private, protected, public, *synchronized*, final, native, abstract és static módosítók elfogadottak, de jelenleg hatása csak a *synchronized* kulcsszónak van. A *throws* ellenőrzött, de nem kierőszakolt.

Scope módosító

Néha ugyanolyan nevű egy külső és a lokális változó. A kérdés az, hogy egy metódusból milyen módon tudunk hivatkozni a külső változóra. Erre szolgál a *super* kulcsszó, aminek a megértéséhez szintén segítsen egy példa:

```
int a = 42;
foo()
{
  int a = 97;
  print( a );
  print( super.a );
}

foo();
 // prints 97, 42
```

Objektumok

A Java osztályok létrehozását ismeri a *BeanShell*, de működik a JavaScripthez hasonló mechanizmus is, amit a következő példa mutat:

```
foo()
{
  int bar = 42;
  return this;
}

fooObject = foo();
print( fooObject.bar ); // prints 42!
```

A *foo()* metódus visszaad egy objektumot a *this* érték miatt, azaz ekkor ez a függvény úgy viselkedik, mint egy konstruktor. Ezután már a *fooObject* objektumon keresztül elérhetjük a *bar* adattagot. A *foo()* függvényen belül metódusokat is létrehozhatunk:

```
foo()
{
...
  bar()
  {
     ...
  }
...
}
```

Interfészek

Nézzük például a Java *ActionListener* interface egy lehetséges implementációját!

```
buttonHandler = new ActionListener()
{
  actionPerformed( event )
  {
    print(event);
  }
};

button = new JButton();
button.addActionListener( buttonHandler );
frame(button);
```

A *buttonHandler* egy *ActionListener* típusú objektum, ami a *button* nyomógombnak átadható eseménykezelőként. Egy script láthatósági körébe tartozó, megfelelő nevű és paraméterezésű metódus szintén használható:

```
actionPerformed( event )
{
  print( event );
}

button = new JButton("Foo!");
button.addActionListener( this );
frame( button );
```

Az *actionPerformed()* metódus látható a *this* hatókörében, amiatt használható is eseménykezelőként. Természetesen a legjobb megoldás, amikor mindezt egy objektumba zárjuk, ahogy ebben a példában is tesszük:

```
messageButton( message )
{
  JButton button = new JButton("Press␣Me");
  button.addActionListener( this );
  JFrame frame = frame( button );

  actionPerformed( e )
  {
    print( message );
    frame.setVisible(false);
  }
}

messageButton("Hey␣you!");
messageButton("Another␣message...");
```

A *messageButton()* konstruktor képes egy (névtelen grafikus nyomógomb) objektumot létrehozni és implementálja is annak *actionPerformed()* metódusát. A fenti *BeanShell* program egymás után 2 nyomógombot tesz ki és annak megnyomására kiírja a megfelelő szövegeket.

Párhuzamos szálak

Amennyiben egy scriptelt class belsejében a *run()* metódust is megvalósítjuk, az képes futni egy Java futási szálon, ahogy a következő kis példa is szemlélteti:

```
Futok().
{
   run().
   {
     print("Futok...");
   }
   return this;
}

Rohanok()
{
   run()
   {
     print("Rohanok...");
   }
   return this;
}

fut = Futok();
rohan = Rohanok();

// Start two threads on foo.run()
```

```
new Thread( fut ).start();
new Thread( rohan ).start();

Futási eredmény:
Futok...
Rohanok...
```

Beágyazás a Java nyelvbe

Egyszerű használat

Az egyik leghatékonyabb és ötletesebb használata a BeanShell-nek az, amikor egy Java programból, beágyazva használjuk. Erre nézzünk meg mindjárt egy példát (2-1. Programlista)! Láthatjuk, hogy a lényeg most is az *Interpreter* class 1 példányának létrehozásában és használatában van. A példában az *Os* osztály (28-33 sorok) csak egy egyszerű *value object*, aminek egyetlen feladata a 3 tagváltozó befoglalása. A 17. sorban tekinthetjük meg az *ip Interpreter* példány létrehozását, majd a 18-21 sorok közötti *set()* metódusokkal átadjuk a Java program változóit a BeanShell most létrehozott *ip* interpreterének. A *set()* 1. paramétere annak a változónak a neve, ahogy azt a BeanShell is látja, míg a 2. paraméter az átadott érték, ami nem csak egy skalár, hanem egy objektum referencia is lehet természetesen. Az 1. és 2. paraméterek által létrehozott megfeleltetést a két környezet (Java és BeanShell) közötti változó mappingnek nevezzük. A 23. sor *eval()* metódusa egy BeanShell script részletet futtat le, azaz az *os* objektum *b* adattagjának az értékét *3333*-ra állítja. A végén a Java program kiírja az *os.b* értékét és láthatjuk, hogy a lefuttatott script darabka megtette a feladatát, a képernyőn *3333* jelent meg.

```
1  // 2-1. Programlista: Hello.java
2
3  package org.cs.beanshell;
4
5  import java.util.Date;
6  import bsh.EvalError;
7  import bsh.Interpreter;
8
9  public class Hello
10 {
11         public static void main(String[] args) throws EvalError
12         {
13                 Os os = new Os();
14                 os.a = 2;
15                 os.c = 10;
16
17                 Interpreter ip = new Interpreter(); // Construct an interpreter
18                 ip.set("foo", 5); // Set variables
19                 ip.set("date", new Date());
20
21                 ip.set("os", os);
22                 System.out.println(os.b);
23                 ip.eval("os.b=3333"); // Fut a script
24                 System.out.println(os.b);
25         }
26 }
27
28 public class Os
29 {
30     public int a;
31     public int b;
32     public int c;
33 }
```

Természetesen a *Hello* class-ban bemutattakon kívül sokkal komplexebb lehetőségeink vannak, amire nemsokára példát is adunk. A 19. sorban átadott dátum értékét Java oldalról így kérhetjük vissza:

```
Date date = (Date) ip.get("date");
```

Ez már egy példa, amikor az inline script segítségével egy beszúrt algoritmusként szorzást is végzünk:

```
ip.eval("bar = foo*10");
System.out.println(ip.get("bar"));
```

Végül ez a 2 Java oldalról hívott BeanShell darabka kiírja a dátumot, majd a Java-ból is ismert *javap()* metódus segítségével a képernyőre írja a *date* objektum osztályáról való tudnivalókat, azaz az őt felépítő metódusokat.

```
ip.eval("print(date); javap(date)");

Sat Apr 27 19:03:39 GMT 2013
Class class java.util.Date extends class java.➥
    lang.Object
public boolean java.util.Date.equals(java.lang➥
    .Object)
public java.lang.String java.util.Date.➥
    toString()
public int java.util.Date.hashCode()
public java.lang.Object java.util.Date.clone()
public int java.util.Date.compareTo(java.util.➥
    Date)
public int java.util.Date.compareTo(java.lang.➥
    Object)
public boolean java.util.Date.after(java.util.➥
    Date)
public boolean java.util.Date.before(java.util➥
    .Date)
public static long java.util.Date.parse(java.➥
    lang.String)
...
```

Külső script használata

Az nehézkes és nem is eredményezne szép kódot, ha a BeanShell scripteket mindig Java Stringként állítanánk össze, ezért ennek elkerülésére van egy egyszerű megoldás, amit a következő példával ismertetünk:

```
// 2-2. Programlista: MyValueObject.java

package org.cs.beanshell;

import java.util.Map;

public class MyValueObject
{
        public Os os;

        public static class BelsoClass
        {
                public Map<String, String> map➥
                    ;
        }

        int i;
        public int getI()
        {
                return i;
        }
        public void setI(int i)
        {
                this.i = i;
        }
        public double getD()
        {
                return d;
        }
        public void setD(double d)
        {
                this.d = d;
        }
        public String getS()
        {
                return s;
        }
        public void setS(String s)
        {
                this.s = s;
        }
        public BelsoClass getBc()
        {
                return bc;
        }
        public void setBc(BelsoClass bc)
        {
                this.bc = bc;
        }
        double d;
        String s;
        BelsoClass bc=null;
}
```

A *MyValueObject* (2-2. Programlista) egy értékeket tárolni képes class, aminek még egy belső statikus osztállyal (*BelsoClass*) megvalósított adattagja is van. Mindezt csak azért készítettük, hogy azt is megmutassuk, hogy bármilyen komplex objektumot átadhatunk egy BeanShell scriptnek, miközben azt most egy külső *kulso-kod.bsh* nevű fájlba (2-3. Programlista) tettük. Maga a *kulso-kod.bsh* csak egyetlen *test()* metódust tartalmaz, ami átvesz egy *MyValueObject* objektumot és tesz rajta változtatásokat.

```
// 2-3. Programlista: kulso-kod.bsh

import org.cs.beanshell.*;

test( obj )
{
  obj.i = 3;
  obj.d = 2.1*3.2;
  obj.s = "Almafa";
  obj.bc.map = new java.util.HashMap();
  obj.bc.map.put("elso", "32");
}
```

A 2-4. Programlista a külső script használatát szemlélteti. A 6. sorban a BeanShell *Interpreter*, a 7-9-ben pedig a *MyValueObject* egy példányát legyártjuk. A 11. sorban az egész *vo* value object-et átadjuk az *ip* interpreternek, amit láthatóan ő is *vo* néven fog ismerni (az első paraméterben adtuk ezt a nevet). A 13. sorban van egy lényeges lépés! Az *ip* változó *source()* metódushívása betölti és lefuttatja az *ip* interpreterrel a külső kódot. Itt most semmi végrehajtandó nincs, mindössze a *test()* metódus ismerete lesz deklarálva az *ip* számára úgy, mintha azt konzolról beírtuk volna. A 14. sorban lefuttatjuk a *test()* metódust a BeanShell által már megismertetett saját *vo* objektumára, ami másfelől Java oldalról is látszik az előzetes összerendelés miatt. A 16-19 sorok kiírásai visszaigazolják, hogy a külső forrásból beolvasott script *test()* metódusa elvégezte a dolgát, a *vo* objektum megváltozott.

```
1  // 2-4. Programlista: TestBeanShellSource.java
2
3  ...
4  public static void testCaseComplex() throws Exception
5  {
6          Interpreter ip = new Interpreter();
7          MyValueObject vo = new MyValueObject();
8
9          vo.bc = new MyValueObject.BelsoClass();
10
11         ip.set("vo", vo);
12
13         ip.source("/home/tanulas/beanshell/kulso-kod.bsh");
14         ip.eval( "test(vo);"  );
15
16         System.out.println( vo.i );
17         System.out.println( vo.d );
18         System.out.println( vo.s );
19         System.out.println(vo.bc.map.get("elso"));
20 }
21 ...
```

Speciális változók és értékek

A BeanShell hatékony használata megkövetelheti, hogy pontosan ismerjük azokat az előre definiált változókat és értékeket, amiket praktikus okok miatt tettek a nyelvbe. Nézzük meg őket röviden!

- *$_*: Visszaadja az legutoljára kiértékelt kifejezés értékét.
- *$_e*: A legutolsó el nem kapott kivétel objektum.
- *bsh*: A script nyelv root objektuma, eddig is használtuk már. Ezen keresztül érhető el a legtöbb bsh szolgáltatás.
- *bsh.args*: A script által kapott paraméterek *String* tömbként reprezentálva.
- *bsh.cwd*: Az aktuális munkakönyvtárat adja vissza.

Fontos körülmény annak az ismerete, hogy egy változó definiálva van, azaz van-e már kezdőértéke. Ezt így lehet megvizsgálni:

```
// undefined
if ( foobar == void )
```

Egy tetszőleges változót bármikor definiálatlanra lehet állítani az *unset()* hívással:

```
a == void; // true
a=5;
unset("a"); // note the quotes
a == void; // true
```

BeanShell parancsok

A BeanShell egy valódi shell program, ezért tartalmazza azokat a parancsokat, amiket egy ilyentől elvárunk. Az alábbiakban kategorizálva röviden áttekintjük őket.

Az Interpreter mód parancsai

- *exit()* → Kilépés az interpreterből.
- *show()* → Be lehet vele kapcsolni azt az üzemmódot, amikor az interpreter minden kiértékelés eredményét kiírja. Például egy *a=3;* értékadás eredményét visszhangozza ilyen formában: *<3>*. Kapcsolóként működik, azaz a következő *show()* hívás kikapcsolja ezt az üzemmódot.
- *setAccessibility()* → Szabályozni lehet vele, a *private* és *protected* tagok elérhetőségét.
- *server(port)* → Ekkor a BeanShell egy távoli klienssel is elérhető. Ez egy böngésző vagy a *telnet* parancs is lehet. Például a *server(3456);* hívás hatására a böngészőből, a *3456* porton keresztül használható lesz ez a távoli BeanShell.

Kiíró utasítások

- *print()* → Kiírja a standard output-ra a paraméterül kapott értéket.
- *error()* → Kiírja a standard error-ra a paraméterül kapott értéket.
- *frame()* → Megjelenít egy *AWT* vagy *Swing* komponenst (például nyomógomb) egy keretben.

Forrásprogramok kiértékelése

- *eval()* → Egy *String*-et kiértékel és végrehajt, mint egy BeanShell scriptet.
- *source()* → A fájlrendszerből betölti a specifikált fájlt és végrehajtja, mint egy BeanShell scriptet.
- *sourceRelative()* → Hasonló a *source()*-hoz, de a fájl helyét a *cwd*-hez (current working directory) relatív módon kell megadni.
- *run(), bg()* → Itt a külső fájl egy új szálon létrehozott másik *Interpreter*-ben lesz végrehajtva.
- *exec()* → Egy külső, operációs rendszerbeli natív programot futtat le.

Közhasznú parancsok (utilities)

- *javap()* → Egy objektum osztályának a metódusait és adattagjait írja ki.
- *which()* → Egy Java class esetén visszaadja azt a *jar* fájlt, amiből őt betöltötték.
- *load(), save()* → Egy tetszőleges, szerializációra engedélyezett objektumot képes betölteni és elmenteni a fájlrendszerbe.
- *object()* → Egy üres object létrehozása.

Változók

- *clear()* → Kitörli az összes változót, metódust és importált neveket az aktuális scope-ból.
- *unset()* → A paraméterként megadott *String* egy változó neve, amit kitöröl a script aktuális látóköréből.
- *setNameSpace()* → A script aktuális scope-jára egy namespace nevet lehet vele beállítani.

CLASSPATH kezelés

Ezek a parancsok lekérdezik vagy megváltoztatják a classpath-t.

- *addClassPath()* → A meglévő CLASSPATH-hoz hozzátesz egy új elemet.
- *setClassPath()* → Beállítja a CLASSPATH-t, az előző megszűnik.
- *getClassPath()* → Lekérdezi az aktuális CLASSPATH-t.
- *reloadClasses()* → Újra betölt a JVM-be egy osztályt vagy az osztályok egy csoportját.
- *getClass()* → Betölt egy osztályt.
- *getResource()* → Lekér egy erőforrást a CLASSPATH-ról.

Fájlok és könyvtárak

- *cd()* → Könyvtárváltás (change directory).
- *pwd()* → Visszaadja az aktuális könyvtárat (print working directory).
- *dir()* → Kilistázza az aktuális könyvtár tartalmát.
- *cp()* → Fájlmásolás (copy).
- *rm()* → Kitöröl egy fájlt (remove)
- *mv()* → Átmozgat egy fájlt másik helyre (move)
- *cat()* → Egy textfájl tartalmát kiírja.
- *pathToFile()* → Egy relative path abszolútra alakítása.

Saját BeanShell parancs

A BeanShell parancsok scriptelt metódusok vagy lefordított Java class-ok. Vegyük például ezt a nagyon egyszerű scriptet, amit a */home/pat/mycommands* könyvtárba mentettünk:

```
// File: helloWorld.bsh
helloWorld()
{
    print("Hello World!");
}
```

Ezt parancsként is használhatjuk, :

```
// ha ezt a 2 sort kiadjuk előtte
addClassPath("/home/pat");
importCommands("/mycommands");

// használhatjuk ezt a parancsot:
helloWorld();
```

A két bevezető utasításra azért van szükség, mert az első hozzáteszi a CLASSPATH-ra a */home/pat* könyvtárat, míg a második az alatta lévő *mycommands* könyvtárból, mint csomagból (package) importál. Amennyiben egy Java classban vannak az új parancsok, úgy csak a szokásos *importCommands("com.xyz.utils");* forma használata szükséges.

Class loading

Alapvető lehetőségek

A BeanShell kifinomult módon képes az osztálybetöltéseket és a CLASSPATH-t kezelni, ezért érdemes azt röviden áttekinteni. Az osztálybetöltő (Class Loader) a CLASSPATH-on megadott helyekről igyekszik az igényelt class-t betölteni. Amennyiben egy megadott *jar* vagy

könyvtár nincs rajta, úgy a már említett *addClassPath()* metódust hívhatjuk segítségül, amire itt van néhány példa:

```
addClassPath( "/home/pat/java/classes" );
addClassPath( "/home/pat/java/mystuff.jar" );
addClassPath( new URL("http://myserver/~pat/➥
    somebeans.jar") );
```

A következő parancs minden osztályt újratölt, ami elérhető:

```
reloadClasses();
```

Ennek létezik egy finomabban hangolt változata is, ahol megadjuk azt a csomagot, amit újra szeretnénk tölteni:

```
reloadClasses("mypackage.*");
```

Természetesen egy konkrét osztály ismételt betöltésére is van lehetőség:

```
reloadClasses("mypackage.EgyClass");
```

Eddig mindig az osztályok újratöltéséről volt szó, de annak manuális első betöltése is lehetséges:

```
name="foo.bar.MyClass";
c = getClass( name );
// vagy:
c = BshClassManager.classForName( name );
```

A BeanShell használati módjai

A BeanShell 5 üzemmódban képes működni:

- Konzol módban scriptek futtatása.
- Java nyelvbe beágyazott módon.
- Remote Server mód
- Servlet mód
- Applet mód.

Az első két üzemmódról már volt szó, az *Applet* módról pedig nem érdemes írni, mert az mára egy elavult technológia.

Remote mód

Ekkor egy másik, távoli shell servletként fut és képes végrehajtani a mi BeanShell scriptünket:

```
java bsh.Remote http://localhost/bshservlet/➥
    eval test1.bsh
```

Lehetőség van egy saját, natív *bsh* hálózati protokoll használatára is:

```
java bsh.Remote bsh://localhost:1234/ test1.➥
    bsh
```

Arról már volt szó, hogy egy futó BeanShell esetén a *server(port);* paranccsal kapcsolható be a listener, azaz ezután ezen a porton tudjuk vele a fenti kliens kéréseket megtenni. A böngészőbe is beírhatjuk a fenti URL-t, aminek a hatására egy command ablakban távolról vezérelhetjük a shell-t. Bár kissé nehézkes, de mindezt természetesen telnet-en keresztül is elvégezhetjük.

Servlet mód

A BeanShell *jar* fájlban a servlet megvalósítás is rendelkezésre áll, azt azonban a szokásos módon egy *war* csomaggal kell körbevenni, ahol a *web.xml* így néz ki:

```
<web-app>
  <servlet>
    <servlet-name>bshservlet</servlet-name>
    <servlet-class>bsh.servlet.BshServlet</➥
        servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>bshservlet</servlet-name>
     <url-pattern>/eval</url-pattern>
  </servlet-mapping>
</web-app>
```

A használható URL a böngészőben:

```
http://localhost/bshservlet/eval
```

Amennyiben helyi scriptet szeretnénk futtatni a távoli servlet segítségével, úgy ezt így tehetjük meg:

```
java bsh.Remote http://localhost/bshservlet/➥
    eval test1.bsh
```

A reflective használati mód

A Java önelemzésen alapuló használat azért fontos, mert néha dinamikus elérésre vagy valamilyen ötletes megoldásra van szükség.

Az *eval()* használata

Ez a legegyszerűbb formája a reflective stílusnak, a korábbiakban már sokszor használtuk is, íme egy emlékeztető példa:

```
eval("a=5;");
print( a ); // 5
```

Ha ismerjük a metódus nevét és szignatúráját, akkor mi is összerakhatunk egy stringet, amivel így meghívhatjuk a metódust. Készítsük el a következő *foo()* és *bar()* metódusokat:

```
// Nincs paramétere
foo() { ... }

// 2 paramétere van
bar( int arg1, String arg2 ) { ... }
```

Ekkor az *eval()* segítségével így hívhatjuk meg őket:

```
name="foo";
// invoke foo() using eval()
eval( name+"()");

name="bar";
arg1=5;
arg2="stringy";
eval( name+"(arg1,arg2)");
```

A *this.methods* egy *String* tömb, ami visszaadja a névtérben elérhető összes metódus nevét.

Az *invokeMethod()* használata

Ezzel a metódussal 1 lépésben is meghívhatjuk dinamikusan a metódust. Például a *bar()* így használható:

```
this.invokeMethod( "bar", new Object [] { new ➥
    Integer(5), "stringy" } );
```

Az 1. paraméter a meghívandó metódus neve, a 2. pedig egy objektum tömb, ami a paramétereket tartalmazza.

A metódusok megkeresése

Az eddigiekben azt mutattuk meg, hogy egy ismert paraméterezésű metódust milyen módon hívhatunk meg. Tekintsük ezt a metódust, ami *bsh.BshMethod* objektumok tömbjével tér vissza:

```
this.namespace.getMethods();
```

Egy konkrét nevű és paraméterezésű metódust is lekérhetünk:

```
// A neve
name="bar";
// A paraméterei
signature = new Class []
{
   Integer.TYPE,
   String.class
};

// És lekérjük az erre illeszkedőt:
bshMethod = this.namespace.getMethod( name, ➥
    signature );
```

A *BshMethod* használata

Ilyen értékeket kaptunk a *getMethods()* metódustól, ugyanakkor ez teszi lehetővé a klasszikus önelemzést. Az alábbi példa mutatja, ahogy lekérdezhetjük egy metódus nevét, paramétereinek és visszatérési értékének típusát:

```
// Önelemzés:
name = bshMethod.getName();
Class [] types = bshMethod.getArgumentTypes();
Class returnType = bshMethod.getReturnType();
```

A metódus meghívása pedig így lehetséges:

```
// Hívás:
bshMethod.invoke( new Object [] { new Integer➥
    (1), "blah!" },
this.interpreter, this.callstack );
```

A *bshMethod* mindent tud magáról (például a metódus nevét), így rajta keresztül egy hívás lebonyolítható. Azt kell megadnunk, hogy milyen paraméter értékek mellett, melyik interpreter futtassa és milyen stack-et használjon hozzá. Ez utóbbi 2 paraméter mindig így jön a futási környezetből, egyszerűen csak át kell adni.

3. Apache Commons - BeanUtils

A Java nyelv komponens technológiája a *JavaBean* fogalomra épül, ami egy olyan osztály, aminek jellemzői vannak (*property*) és rendelkezik paraméter nélküli konstruktorral is. Gyakran szinonim fogalomként emlegetik a *POJO*-t, ami a *Plain Old Java Object* kifejezésből származó mozaikszó. Ebben a részben áttekintjük az Apache *BeanUtils* könyvtár fontosabb lehetőségeit, hiszen az ott megvalósított funkciókra időnként szükségünk lehet.

Az Apache *BeanUtils* könyvtár egy alacsonyszintű Java Bean önelemzést és property manipulálást (Bean Introspection Utilities) megvalósító programozói csomag.

Szabványos JavaBean

A Java egyik legismertebb fogalma a *JavaBean*. Ez egy olyan egyszerű class, aminek birtokolnia kell egy paraméter nélküli konstruktort. Kiemelt feladata, hogy a komponens technológiáknál megismert property adatokat tároljon, amiket *setter* és *getter* metódusokkal lehet beállítani, illetve elérni. Az alábbiakban látható *Vitamin* osztály (3-1. Programlista) egy JavaBean. Megjegyezzük, hogy nem adtunk meg explicit konstruktort, ezért a rendszer generál neki egy alapértelmezettet, ami paraméter nélküli.

```
// 3-1. Programlista: Vitamin.java

package org.cs.beanutils;

public class Vitamin
{
  String name;
  double volumen;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public double getVolumen() {
    return volumen;
  }
  public void setVolumen(double volumen) {
    this.volumen = volumen;
  }
}
```

A *Gyumolcs* class (3-2. Programlista) egy kicsit összetettebb JavaBean, tartalmaz adattagként egy *Vitamin* osztálybeli változót is. Ezen osztály segítségével a továbbiakban bemutatjuk a *BeanUtils* csomag legfontosabb használati lehetőségeit.

```
// 3-2. Programlista: Gyumolcs.java

package org.cs.beanutils;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Gyumolcs
{
  String name;
  int price;
  String quality;
  double weight;
  String[] props;
  List<String> propList;
  Map<String, String> propMap;
  Vitamin vitamin;

  public Gyumolcs()
  {
    super();
    props = new String[3];
    propList = new ArrayList<String>();
    propMap = new HashMap<String, String>();
    vitamin = new Vitamin();
  }

  public List<String> getPropList()
  {
    return propList;
  }

  public void setPropList(List<String> propList)
  {
    this.propList = propList;
  }



  public Vitamin getVitamin()
  {
    return vitamin;
  }

  public void setVitamin(Vitamin vitamin)
  {
    this.vitamin = vitamin;
  }

  public Map<String, String> getPropMap()
  {
    return propMap;
  }

  public void setPropMap(Map<String, String> propMap)
  {
    this.propMap = propMap;
  }

  public String[] getProps()
  {
    return props;
  }
```

```
  public void setProps(String[] props)
  {
    this.props = props;
  }

  public String getName()
  {
    return name;
  }

  public void setName(String name)
  {
    this.name = name;
  }

  public int getPrice()
  {
    return price;
  }

  public void setPrice(int price)
  {
    this.price = price;
  }

  public String getQuality()
  {
    return quality;
  }

  public void setQuality(String quality)
  {
    this.quality = quality;
  }

  public double getWeight()
  {
    return weight;
  }

  public void setWeight(double weight)
  {
    this.weight = weight;
  }
}
```

Még itt az elején bemutatjuk a bizonyára sokak által ismert dinamikus osztály példány létrehozás szokásos módját (3-3. Programlista). A példát a *createGyumolcs()* metódus mutatja be nekünk, ami visszaad egy *Gyumolcs* objektumot. *A className* változó tartalmazza annak a class-nak a teljes nevét, amiből gyártani szeretnénk 1 példányt. A *beanClass* objektum egy meta objektum, ami egy betöltött *Gyumolcs* osztályra mutat. Az objektum legyártását a *beanClass.newInstance()* sor valósítja meg, azaz megkérjük a meta objektumot, hogy a maga képéről készítsen el egy class instance-t, amit az utolsó sorban vissza is adunk a metódus hívójának.

```
// 3-3. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
  public static Gyumolcs createGyumolcs() throws
      Exception
  {
    String className = "org.cs.beanutils.Gyumolcs";
    Class beanClass = Class.forName(className);
    Object beanInstance = beanClass.newInstance();
    return (Gyumolcs)beanInstance;
  }
...
} // end class
```

A JavaBean jellemzők elérése

Az Apache BeanUtils könyvtárat a project hivatalos webhelyéről lehet letölteni: `http://commons.apache.org/proper/commons-beanutils/`. Ebben a pontban áttekintjük, hogy egy objektum adattagjait milyen dinamikus eszközökkel lehet elérni.

A jellemzők elérése

A 3-4. Programlista *testGetSetProperty()* metódusa bemutatja nekünk azt a módszert, ahogy a BeanUtils támogatja a bean property-k elérését és értékük megváltoztatását. Az *alma* változó egy *Gyumolcs* példányra mutat. Az *alma.setName()* hívás még az ismert módon állítja be a *name* property értékét *Starking*-ra. A jellemzők elérését a *PropertyUtils* osztály támogatja, rendelkezik néhány statikus, saját felfogású setter és getter metódussal. A példában használt *PropertyUtils.getProperty()* hívás első paramétere az a bean (esetünkben az *alma*), amelyiknek a második paraméterben megadott nevű jellemzőjének (most ez *name*) értékét szeretnénk visszakapni. A metódus *Object*-et ad vissza, ezért *String*-re cast-oltuk, majd a következő sor kiírása visszaigazolta, hogy eredményesek vagyunk, a képernyőn megjelent a *Starking* szó. A tesztmetódus utolsó 2 sora a jellemző beállítást szemlélteti. Ehhez a *PropertyUtils.setProperty()* használható, ebben a paraméterezésben:

1. A bean-re hivatkozó változó
2. A bean jellemző neve

3. A beállítandó érték (esetünkben most *Golden*)

```
// 3-4. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
...
  public static void testGetSetProperty() throws ➥
      Exception
  {
    String propValue = null;
    Gyumolcs alma = createGyumolcs();
    alma.setName("Starking");

    String name = (String)PropertyUtils.getProperty(➥
        alma, "name");
    System.out.println( name );
    PropertyUtils.setProperty(alma, "name", "Golden");
    System.out.println( alma.name );
  }
...
} // end class
```

Indexelt jellemzők elérése

Egy JavaBean property lehet kollekció vagy tömb is. A *Gyumolcs* osztályban ezek az adattagok indexelt elemeket tartalmaznak:

- *String[] props*
- *List<String> propList*

Fontos, hogy az ilyen típusú elemek dinamikus kezelése is megoldott legyen, amit a *PropertyUtils* class alábbi metódusai meg is tesznek:

- *setIndexedProperty(bean, property, index, value)*
- *getIndexedProperty(bean, property, index)*

A következőkben 2 példát is bemutatunk, mert szeretnék demonstrálni a tömbök és a listák használatát is. Az első példánk (3-5. Programlista) a listák indexelt elérését mutatja, erre a *propList* bean property-t használtuk fel, amihez a *piros* és *kerek* értékeket hozzá is tettük 2 darab *add()* metódushívással. A következő sorban jön az első újdonság, ugyanis a *PropertyUtils* class *setIndexedProperty()* módszerével a *propList* lista jellemző 1. tagjának (azaz a fizikai 2. tagot) eddigi *kerek* értékét *kerekded*-re cseréltük. Ez egy indexelt, írásra való property elérés. A *for* ciklusban hagyományos és a most tanult indexelt módon való eléréssel is kiírtuk a képernyőre *propList* jellemzőt, ami visszaigazolta a helyes működést, ugyanis a 2. listaelem *kerekded* értékű lett.

```
// 3-5. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
...
  public static void testIndexedListProperty() throws ➥
      Exception
  {
    Gyumolcs alma = createGyumolcs();

    alma.getPropList().add("piros");
    alma.getPropList().add("kerek");

    PropertyUtils.setIndexedProperty(alma, "propList",➥
        1, "kerekded");

    for (int i=0; i<alma.getPropList().size(); i++)
    {
      System.out.println( alma.getPropList().get(i) );
      String s = (String)PropertyUtils.➥
          getIndexedProperty(alma, "propList", i);
      System.out.println( s );
    }
  }
...
} // end class
```

Mindez persze a leghagyományosabb indexelt elérésű konstrukcióra, a Java tömbre is működik, ahogy azt a 3-6. Programlista be is mutatja. Ez a tömb esetünkben most a *props* jellemző. A működés megegyezik a listás esettel.

```
// 3-6. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
...
  public static void testIndexedArrayProperty() throws➥
      Exception
  {
    Gyumolcs alma = createGyumolcs();

    alma.getProps()[0]="sárga";
    alma.getProps()[1]="kerek";
    alma.getProps()[2]="finom";

    PropertyUtils.setIndexedProperty(alma, "props", 1,➥
        "kerekded");

    for (int i=0; i<alma.getProps().length; i++)
    {
      System.out.println( alma.getProps()[i]);
      String s = (String)PropertyUtils.➥
          getIndexedProperty(alma, "props", i);
```

```
      System.out.println( s );
    }
  }
...
} // end class
```

Map típusú jellemzők elérése

A mai korszerű nyelveknek a *Map* adatszerkezet fontos konstrukciója. Ezt asszociatív elérésű tömbnek is tekinthetjük, ugyanis nem egy fizikai index választja ki az onnan megkapott értéket, hanem egy objektum értéke, tipikusan sok esetben egy *String* értéke. Tekintsük a *Gyumolcs* osztály *propMap* jellemzőjét, ami egy *Map*. A 3-7. Programlista példázza ennek a *PropertyUtils* class segítségével történő írásra-olvasásra való elérését. Programozói szemszögből láthatjuk, hogy a használata hasonló az indexelt eléréssel, mindössze 2 lényeges változtatásra volt szükség. A *setMappedProperty()* és *getMappedProperty* metódusokat kell használni. Mindkét metódus első 3 paramétere az adott bean, annak a megfelelő nevű jellemzője és a kulcsérték, amivel elérjük a *Map* egyik elemét. A *setMappedProperty()* 4. paramétere az az érték, amire a kiválasztott elemet be szeretnénk állítani. A fenti 2 metódus paraméterezése természetszerűleg annyiban változott, hogy ahol index volt, egy *Map* kulcsértéknek kell szerepelnie. Ez esetünkben most írásnál az *íz*, lentebb az olvasásnál pedig a *szin String* érték.

```
// 3-7. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
...
  public static void testMapProperty() throws ➥
      Exception
  {
    Gyumolcs alma = createGyumolcs();

    alma.getPropMap().put("szin", "zöld");
    alma.getPropMap().put("alak", "kerek");

    PropertyUtils.setMappedProperty(alma,"propMap","íz➥
        ","finom");

    System.out.println( alma.getPropMap().get("szin") ➥
        );
    System.out.println( alma.getPropMap().get("alak") ➥
        );
    System.out.println( alma.getPropMap().get("íz") );

    String s = (String)PropertyUtils.getMappedProperty➥
        (alma,"propMap","szin");
    System.out.println( s );
  }
...
} // end class
```

A beágyazott jellemzők elérése

Tipikus eset, hogy valamelyik property maga is egy bean, aminek persze saját jellemzője is van, amit ebben a kontextusban emiatt beágyazott property-nek nevezünk. A 3-8. Programlista ezt az esetet vizsgálja, ahol a *vitamin* jellemző egy *Vitamin* bean. A példa bemutatja, hogy a *vitamin* tagváltozó saját *name* és *volumen* jellemzőihez a *PropertyUtils* class erre kitalált *getNestedProperty()* és *setNestedProperty()* statikus metódusai milyen módon férnek hozzá. Láthatjuk, hogy a trükk egészen egyszerű, a beágyazott property nevét minősített névként (*vitamin.name*, *vitamin.volumen*) kell átadni a metódusoknak, ettől eltekintve a használat megegyezik a nem beágyazott jellemző esetével.

```
// 3-8. Programlista: Test.java

package org.cs.beanutils;

import java.lang.reflect.InvocationTargetException;
import java.util.List;
import org.apache.commons.beanutils.PropertyUtils;

public class Test
{
...
  public static void testNestedProperty() throws ➥
      Exception
  {
    Gyumolcs alma = createGyumolcs();
    alma.getVitamin().setName("C");
    alma.getVitamin().setVolumen(32.54);

      System.out.println( alma.getVitamin().getName() ➥
          );

    String s = (String)PropertyUtils.getNestedProperty➥
        (alma, "vitamin.name");
    double v = (Double)PropertyUtils.getNestedProperty➥
        (alma, "vitamin.volumen");
    System.out.println( v );

    PropertyUtils.setNestedProperty(alma, "vitamin.➥
        volumen", 41.0);
    v = (Double)PropertyUtils.getNestedProperty(alma, ➥
        "vitamin.volumen");
    System.out.println( v );
  }
...
} // end class
```

Dinamikus JavaBean

A *PropertyUtils* class képes kezelni egy már létező JavaBean osztályt, azonban arra is szükség lehet, hogy egy osztály jellemzőit futás közben alakítsuk ki. A *BeanUtils* csomag erre egy interface-t definiált, aminek a neve *DynaBean*. A továbbiakban bemutatjuk, hogy milyen módon lehet olyan objektumokat készíteni, amik ezen az interface-en keresztül manipulálhatóak. Ezek nem klasszikus JavaBean-ek, de azokhoz hasonlóan lehet használni.

Az alap dinamikus JavaBean

A dinamikus bean-ek megismerésére tekintsük a *TestDynaBean* osztályt (3-9. Programlista). Létrehozunk futás közben egy olyan bean-t, aminek a következő típusú property-ei vannak:

- *java.util.Map*
- egy *Vitamin* példányokból álló tömb
- *String* jellemző

Egy *DynaProperty* tömbben (esetünkben most a neve *props*) írhatjuk le a létrehozandó bean jellemzőit, ahol a tömb egyes elemei a következő információkat tárolják el:

- a jellemző neve (például esetünkben most ezek: *jellemzok*, *vitamin*, *name*, *latinName*) és
- az osztályának teljes minősítéssel megadott megnevezése

A jellemzők ismeretében megkonstruálható egy *BasicDynaClass* metaobjektum, ami maga az új bean osztály. Paraméterül megadtuk, hogy *zoldseg* a neve és a *props* tömbben lévő proprty-k alapján képezze a saját adattagjait. A *dynaClass* metaobjektum *newInstance()* metódusa segítségével már létre is tudtuk hozni a *DynaBean* típusú *zoldseg* változót. A *testBasic()* tesztmetódus utolsó 2 sorában a *name* property írását és olvasását mutattuk be.

```
// 3-9. Programlista: TestDynaBean.java

package org.cs.beanutils;

import org.apache.commons.beanutils.BasicDynaBean;
import org.apache.commons.beanutils.BasicDynaClass;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.DynaProperty;
import org.apache.commons.beanutils.WrapDynaBean;

public class TestDynaBean
{
...
  public static void testBasic() throws Exception
  {
    DynaProperty[] props = new DynaProperty[]
    {
        new DynaProperty("jellemzok", java.util.Map.➥
            class),
        new DynaProperty("vitamin", org.cs.beanutils.➥
            Vitamin[].class),
        new DynaProperty("name", String.class),
        new DynaProperty("latinName", String.class)
    };

    BasicDynaClass dynaClass = new BasicDynaClass("➥
        zoldseg", null, props);
    DynaBean zoldseg = dynaClass.newInstance();
    zoldseg.set("name", "krumpli");
    System.out.println( zoldseg.get("name") );
  }
...
}
```

A JavaBean becsomagolása

A BeanUtils *DynaBean* arra is képes, hogy becsomagoljon egy szabványos JavaBean-t, aminek a jellemzőit ezután név szerint, a saját set és get metódusával érje el. Mindezt a 3-10. Programlista *testWrapper()* metódusán tanulmányozhatjuk. A *Gyumolcs* nevű JavaBean-t ismerjük, ennek a *gyumolcsBean* változón keresztül elérhető példányát fogjuk becsomagolni egy *DynaBean* objektumba. Az ilyen wrapping-re alkalmas osztály neve a *WrapDynaBean*. A példában a *wrapper* változó egy már becsomagolt *gyumolcsBean* referenciáját tartalmazza. Ahogy a *testWrapper()* utolsó 4 sorából látjuk, ez már egy ugyanolyan *DynaBean*, mintha mi konstruáltuk volna meg, ugyanakkor a jellemzők nevét használhatjuk a get és set elérésekhez.

```
// 3-10. Programlista: TestDynaBean.java

package org.cs.beanutils;

import org.apache.commons.beanutils.BasicDynaBean;
import org.apache.commons.beanutils.BasicDynaClass;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.commons.beanutils.DynaBean;
```

```
import org.apache.commons.beanutils.DynaProperty;
import org.apache.commons.beanutils.WrapDynaBean;

public class TestDynaBean
{
...
  public static void testWrapper() throws Exception
  {
    Gyumolcs gyumolcsBean = new Gyumolcs();
    gyumolcsBean.setName("Alma");
    gyumolcsBean.setPrice(320);

    DynaBean wrapper = new WrapDynaBean( gyumolcsBean
        );
    String name = (String)wrapper.get("name");
    int price = (Integer)wrapper.get("price");
    System.out.println( name );
    System.out.println( price );
  }
...
}
```

Az SQL ResultSet becsomagolása

A BeanUtils képes a JDBC *ResultSet* eredményt is *DynaBean* interface-en keresztül prezentálni. Ennek létezik online és offline változata is, most először nézzük az elsőt (3-11. Programlista)! Az *rs* változó az ismert SQL kurzort tartalmazza. A példában azt mutattuk meg, hogy ezt a *ResultSetDynaClass* segítségével miként csomagolhatjuk be, illetve kérhetünk le ettől a csomagolt objektumtól egy *Iterator*-t, amit *rows*-nak neveztünk el. A *while* ciklus belsejében látható, ahogy az iterátor segítségével végigmegyünk az eredményhalmaz sorain és egy *DynaBean* interfaceszel rendelkező *row* objektumhoz férünk minden lépésben. A ciklusból kilépve lezárjuk az adatbázis eléréshez szükséges erőforrásokat.

```
// 3-11. Programlista: Online ResultSet

...
Connection conn = ...;
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery
    ("select account_id, name from customers");
  Iterator rows = (new ResultSetDynaClass(rs)).
      iterator();
  while (rows.hasNext()) {
    DynaBean row = (DynaBean) rows.next();
    System.out.println("Account number is " +
                  row.get("account_id") +
                  " and name is " + row.get("name"));
  }
  rs.close();
  stmt.close();
...
```

Az SQL ResultSet becsomagolása (offline)

A 3-12. Programlista a JDBC *ResultSet* offline használatát tanítja meg, amihez a *RowSetDynaClass* osztály használata szükséges. Ez utóbbi osztály szintén az eredményhalmazt csomagolja be, de ennek tartalmát át is másolja magába, ami lehetővé teszi, hogy még a használata előtt lezárjuk az adatbázishoz kötődő erőforrásokat. A példában az *rsdc* változót használtuk, aminek a *getRows()* metódusa ad vissza egy olyan listát, aminek az elemeit már a *DynaBean* interface-en keresztül is elérhetjük.

```
// 3-12. Programlista: Offline ResultSet

...
Connection conn = ...;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT ...");
RowSetDynaClass rsdc = new RowSetDynaClass(rs);
rs.close();
stmt.close();
...;     // Return connection to pool
List rows = rsdc.getRows();
...;     // Process the rows as desired
```

A lusta Dinamikus JavaBean

A lusta jelző most a programozóra utal, aki gyorsabban szeretne hozzáférni egy *DynaBean* objektumhoz. Ezt a BeanUtils *LazyDynaBean* class támogatja is, ahogy azt a 3-13. Programlista be is mutatja. Nincs más dolgunk csak létrehozni egy *LazyDynaBean* példányt, amit esetünkben most egy *dynaBean* változón keresztül látunk. Az egyszerű, a mappelt és az indexelt elérést is megmutatja a *lustaBean()* metódus. Látható, hogy egyszerű esetben csak kitalálunk egy property nevet (most ez *foo*) és már tehetjük is bele az értéket. A mapped használat sem nehezebb, de ott még a kulcs nevét (esetünkben: *title* és *surname*) is meg kellett adnunk. Az indexelt elérés is hasonló, de itt az indexet kell megadni a kulcs neve helyett.

```
// 3-13. Programlista: LazyDynaBean

package org.cs.beanutils;

import org.apache.commons.beanutils.BasicDynaBean;
import org.apache.commons.beanutils.BasicDynaClass;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.commons.beanutils.DynaBean;
import org.apache.commons.beanutils.DynaProperty;
import org.apache.commons.beanutils.WrapDynaBean;

public class TestDynaBean
{
...
  public static void lustaBean()
  {
    DynaBean dynaBean = new LazyDynaBean();
```

```
        dynaBean.set("foo", "bar");                          //➥
              simple

        dynaBean.set("customer", "title", "Mr");             //➥
              mapped
        dynaBean.set("customer", "surname", "Smith");        //➥
              mapped

        System.out.println( dynaBean.get( "foo" ) );

        Gyumolcs alma = new Gyumolcs();
        Gyumolcs korte = new Gyumolcs();
        Gyumolcs szilva = new Gyumolcs();

        dynaBean.set("address", 0, alma);       // indexed
        dynaBean.set("address", 1, korte);      // indexed
        dynaBean.set("address", 2, szilva);     // indexed
    }
...
}
```

Létezik egy *LazyDynaMap* osztály is, amit hasonlóan használhatunk (3-14. Programlista). Érdekességként megjegyezzük, hogy az utolsó sorban látható *getMap()* metódussal bármikor egy szabványos *Map* objektumot kérhetünk le ettől a *DynaBean* objektumtól.

```
// 3-14. Programlista:

DynaBean dynaBean = new LazyDynaMap();      // create ➥
     DynaBean

dynaBean.set("foo", "bar");                 // simple
dynaBean.set("customer", "title", "Mr");  // mapped
dynaBean.set("address", 0, addressLine1); // indexed

Map myMap = dynaBean.getMap()          // retrieve the ➥
     Map
```

Biztos sok olvasónak eszébe jutott a kérdés, hogy egy létező *Map* objektumból vajon lehet-e *DynaBean*-t készíteni. A 3-15. Programlista pont ezt tartalmazza:

```
// 3-15. Programlista:

Map myMap = ....    // exisitng Map
DynaBean dynaBean = new LazyDynaMap(myMap);  // wrap ➥
     Map in DynaBean
dynaBean.set("foo", "bar");                  // set ➥
     properties
```

A 3-16. Programlista és 3-17. Programlista példák további használati lehetőségeket mutatnak, amiket az eddigiek alapján könnyen meg tudunk érteni, ezért nem is fűzünk magyarázatot hozzájuk. Természetesen mindkét lehetőség célja, hogy egy *DynaBean* interface-szel rendelkező objektumhoz jussunk, amivel kezeljük a jellemzők értékeit.

```
// 3-16. Programlista:

MutableDynaClass dynaClass = new LazyDynaClass();     // create DynaClass

dynaClass.add("amount", java.lang.Integer.class);     // add property
dynaClass.add("orders", OrderBean[].class);           // add indexed property
dynaClass.add("orders", java.util.TreeMapp.class);    // add mapped property

DynaBean dynaBean = new LazyDynaBean(dynaClass);      // Create DynaBean with associated DynaClass
```

```
// 3-17. Programlista:

DynaBean dynaBean = new LazyDynaBean();               // Create LazyDynaBean
MutableDynaClass dynaClass =
          (MutableDynaClass)dynaBean.getDynaClass();  // get DynaClass

dynaClass.add("amount", java.lang.Integer.class);     // add property
dynaClass.add("myBeans", myPackage.MyBean[].class);   // add 'array' indexed property
dynaClass.add("myMap", java.util.TreeMapp.class);     // add mapped property
```

JavaBean konverziós lehetőségek

Egy bean jellemzőit írhatjuk, olvashatjuk és szükség esetén annak értékeit átalakíthatjuk. Sok esetben az a program, amelyik ezeket az érték konvertálásokat végzi szintén fekete doboz számunkra, bár ők is a getter és setter metódusokkal szerzik és írják az értékeket. Annak érdekében, hogy ezen metódusok által visszaadott értékeket mégis befolyásolni tudjuk, készíthetünk olyan objektumokat, amelyek a *Converter* interface-t implementálják. Az alábbiakban 2 egyszerű *Converter* osztály mutatunk be:

- *MyStringConverter* (3-18. Programlista)
- *MyLongConverter* (3-19. Programlista)

```
// 3-18. Programlista: MyStringConverter.java

package org.cs.beanutils;

import org.apache.commons.beanutils.Converter;

public class MyStringConverter implements Converter
{
  public Object convert(Class type, Object value)
```

```
  {
    if (value == null)
    {
    return (String) null;
    } else
    {
    return (value.toString().replaceAll("ma", "x"));
    }
  }
}
```

```
// 3-19. Programlista: MyLongConverter.java

package org.cs.beanutils;

import org.apache.commons.beanutils.➥
    ConversionException;
import org.apache.commons.beanutils.Converter;

public class MyLongConverter implements Converter
{
   private Object defaultValue;
    private boolean useDefault;

    public MyLongConverter() {
      this(true, new Long(0));
    }

    public MyLongConverter(boolean useDefault, Object ➥
        defaultValue) {
      this.useDefault = useDefault;
      this.defaultValue = defaultValue;
    }

    public Object convert(Class type, Object value) {
      if(value == null) {
        if(useDefault) {
          return defaultValue;
        } else {
   throw new ConversionException("No default value ➥
        specified");
        }
      }

      if(value instanceof Long) {
        return new Long(((Long)value).longValue() + ➥
            1000);
      } else {
        try {
          return new Long(new Long(value.toString()).➥
              longValue() + 1000);
        } catch (Exception e) {
          System.err.println(e);
          if(useDefault) {
            return defaultValue;
          } else {
            throw new ConversionException(e);
          }
        }
      }
    }
}
```

A fenti 2 *Converter* class implementációja nem túl bonyolult. A *ConverterTest* (3-20. Programlista) bemutatja a *MyStringConverter* használatát.

```
1  // 3-20. Programlista: ConverterTest.java
2
3  package org.cs.beanutils;
4
5  import org.apache.commons.beanutils.BeanUtilsBean;
6  import org.apache.commons.beanutils.ConvertUtilsBean;
7  import org.apache.commons.beanutils.PropertyUtils;
8  import org.apache.commons.beanutils.PropertyUtilsBean;
9
10 public class ConverterTest
11 {
12   public static void main(String[] args) throws Exception
13   {
14     Gyumolcs gyumolcs = new Gyumolcs();
15     gyumolcs.name = "Alsssssssma";
16     gyumolcs.price = 55;
17
18     ConvertUtilsBean convertUtilsBean = new ConvertUtilsBean();
19     convertUtilsBean.deregister(String.class);
20     convertUtilsBean.register(new MyStringConverter(), String.class);
21
22     BeanUtilsBean beanUtilsBean = new BeanUtilsBean(convertUtilsBean,
23         new PropertyUtilsBean());
24
25     System.out.println("By PropertyUtils: " +
26             PropertyUtils.getProperty(gyumolcs, "name"));
27
28     System.out.println("By BeanUtils: " +
29             beanUtilsBean.getProperty(gyumolcs, "name"));
30   }
31 }
32
33 Futási eredmény:
34 By PropertyUtils: Alsssssssma
35 By BeanUtils: Alssssssx
```

A 14-16 sorok között létrehozunk egy *gyumolcs* objektumot. A 18-20 sorokban létrehozzuk a *convertUtilsBean* konvertáló objektumot, amiben az alapértelmezett *String.class*-hoz rendelt konvertert leregisztráljuk. Ezután a mi *MyStringConverter* osztályunkhoz rendeljük az összes *String.class* típushoz szükséges elérő konverziót. A 22. sorban létrehozott *beanUtilsBean* objektum a *BeanUtils* osztály „objektumos" változata, ahol *convertUtilsBean* és *PropertyUtilsBean* objektumokat is összerendeljük. Ennyi előzetes beállítás után a 25. és 28. sor által használt *gyumolcs* bean jellemző lekérdezés alapvetően másképpen működik. Az első a már jól ismert statikus *PropertyUtils* class segítségével dolgozik, ez semmit sem tud arról, hogy van egy saját konverterünk, emiatt a tényleges *Alsssssssma* értéket jeleníti meg. A 2. kiírás már az „objektumos" *beanUtilsBean* bean-t használja ugyanerre a feladatra, de itt a *String*re regisztráltunk egy sajátot, emiatt most az *Alsssssssx* fog megjelenni, ugyanis az minden „*ma*" részletet „*x*"-re cserél.

Végül itt jegyezzük meg, hogy a BeanUtils statikus utility osztályokat (*BeanUtils*, *ConvertUtils*, *PropertyUtils*) és az ezekhez hasonló osztály példányokat (*BeanUtilsBean*, *ConvertUtilsBean*, *PropertyUtilsBean*) is rendelkezésünkre bocsátja. ugyanazzal az API-val. Ugyanakkor az „objektumos" változat több egyedi működést is tud hordozni, ahogy a konvertereknél láttuk is.

A *BeanUtils* további lehetőségei

Befejezésül nézzünk meg néhány hasznos lehetőséget, az egyszerűség érdekében csak a *BeanUtils* statikus metódusain keresztül vizsgálva. A *cloneBean(Object)→Object* metódus egy tetszőleges klón objektumot ad vissza. A *copyProperties(Object dest, Object orig)* az *orig* objektum minden olyan jellemzőjének az értékét másolja a *dest* objektumba, ahol a property nevek megegyeznek. Itt akármilyen 2 eltérő objektum lehet, ami nagy rugalmasságot biztosít. A következő példánkban most egy ugyanolyan *Gyumolcs* típusú objektumba másoltunk, de ez lehetett volna akármilyen más class példány is:

```
Gyumolcs gyumolcs2 = new Gyumolcs();
BeanUtils.copyProperties(gyumolcs2, gyumolcs);
System.out.println( gyumolcs2.name );
```

A *populate(object, map)* használatát a következő kis programtöredék mutatja:

```
Map p =  new HashMap();
p.put("name", "Körte");
BeanUtils.populate(gyumolcs, p);
System.out.println( gyumolcs.name );
```

A *Map* objektum azon értékeit, ahol a kulcs neve egyezik a *gyumolcs* objektum property nevével, átmásolja az objektumba, így a képernyőre a *Körte* szó kerül kiírásra. A *describe(Object)→Map* a paraméter objektum leírását egy *Map*-be teszi:

```
Map p =  new HashMap();
p = BeanUtils.describe(gyumolcs);
System.out.println( p );

Eredmény:
{propMap={}, weight=0.0, price=55, name=Körte, quality
    =null, propList=null, class=class org.cs.
    beanutils.Gyumolcs, vitamin=org.cs.beanutils.
    Vitamin@a8c488, props=null}
```

Kollekciók

A *BeanPropertyValueChangeClosure* class használatát a következő példa szemlélteti:

```
// create the closure
BeanPropertyValueChangeClosure closure =
  new BeanPropertyValueChangeClosure( "activeEmployee"
      , Boolean.TRUE );

// update the Collection
CollectionUtils.forAllDo( peopleCollection, closure );
```

A *closure* kód darab egy *activeEmployee* property-t (bármilyen objektumban is van) képes igazra állítani. Használatát az utolsó sor mutatja.

4. Apache Commons - Lang

Az Apache Commons Lang könyvtári csomag létrehozását az motiválta, hogy a standard *java.lang* package tudását kiegészítse. A benne lévő eszközök nagyon általánosak, így a mindennapi programozói feladatokban sokat segíthetnek. A project webhelye: `http://commons.apache.org/proper/commons-lang/`. A bemutatott példák nagy része az API leírásból származik.

Az Apache Commons Lang jelenleg 2 fő változatban használatos, amiket a 2. és 3. verzió néven emlegetünk és nem keverednek egymással, mert különböző csomagokban vannak:

- 2. verzió: *org.apache.commons.lang*
- 3. verzió: *org.apache.commons.lang3*

A továbbiakban elsősorban a 3. verziót tekintjük az ismertetés alapjául, de megjegyezzük, hogy sok szoftver még a 2. verziót használja.

String műveletek - StringUtils

Rész String lekérés

A *left()* és *right()* metódusok a String bal, illetve jobb részét adják vissza annyi karakterben, amennyi a 2. paraméterben meg lett adva.

```
StringUtils.left(null, *)    = null
StringUtils.left(*, -ve)     = ""
StringUtils.left("", *)      = ""
StringUtils.left("abc", 0)   = ""
StringUtils.left("abc", 2)   = "ab"
StringUtils.left("abc", 4)   = "abc"

StringUtils.right(null, *)    = null
StringUtils.right(*, -ve)     = ""
StringUtils.right("", *)      = ""
StringUtils.right("abc", 0)   = ""
StringUtils.right("abc", 2)   = "bc"
StringUtils.right("abc", 4)   = "abc"
```

A *leftPad()* egy olyan Stringet ad vissza, ami az 1. paraméterből úgy keletkezik, hogy a 2. paraméterben megadott számérték lesz a String hossza. Amennyiben az eredeti String ennél hosszabb, úgy azt balról szóközzel egészíti ki. A *rightPad*() működése hasonló, de szükség esetén itt jobbról lesznek a szóközök hozzáragasztva.

```
StringUtils.leftPad(null, *)   = null
StringUtils.leftPad("", 3)     = "␣␣␣"
StringUtils.leftPad("bat", 3)  = "bat"
StringUtils.leftPad("bat", 5)  = "␣␣bat"
StringUtils.leftPad("bat", 1)  = "bat"
StringUtils.leftPad("bat", -1) = "bat"

StringUtils.rightPad(null, *)   = null
StringUtils.rightPad("", 3)     = "␣␣␣"
StringUtils.rightPad("bat", 3)  = "bat"
StringUtils.rightPad("bat", 5)  = "bat␣␣"
StringUtils.rightPad("bat", 1)  = "bat"
StringUtils.rightPad("bat", -1) = "bat"
```

A következő 2 metódus működése hasonló, de megadhatjuk azt a karaktert is, ami ez esetben a szóköz helyett foglalja a helyet.

```
leftPad(String str, int size, char padChar);
rightPad(String str, int size, char padChar);
```

A *mid()* függvény igyekszik visszaadni *len* darab karakterből álló Stringet a kezdőpozíciótól

```
mid(String str, int pos, int len);
oStr = StringUtils.mid("0123456789", 3, 3);

// Eredmény (10 hosszú String):
345
```

A könyvtár természetesen tartalmazza a klasszikus *substring()* metódust is, aminek a működése megegyezik a String beépített lehetőségével, de nem fut kivételre a szélsőséges esetekben:

```
// Amikor csak a start van megadva
StringUtils.substring(null, *)   = null
StringUtils.substring("", *)     = ""
StringUtils.substring("abc", 0)  = "abc"
StringUtils.substring("abc", 2)  = "c"
StringUtils.substring("abc", 4)  = ""
StringUtils.substring("abc", -2) = "bc"
StringUtils.substring("abc", -4) = "abc"

// Start és End is van
StringUtils.substring(null, *, *)    = null
StringUtils.substring("", * ,  *)    = "";
StringUtils.substring("abc", 0, 2)   = "ab"
StringUtils.substring("abc", 2, 0)   = ""
StringUtils.substring("abc", 2, 4)   = "c"
StringUtils.substring("abc", 4, 6)   = ""
StringUtils.substring("abc", 2, 2)   = ""
StringUtils.substring("abc", -2, -1) = "b"
StringUtils.substring("abc", -4, 2)  = "ab"
```

Van néhány további *substring()* mutáció is, ezekről jó tudni. A *substringBefore()* metódus például visszaadja azt a rész Stringet, amit a 2. paraméterben megadott szeparátor előtt talált:

```
StringUtils.substringBefore("almafa", "af") = "alm"
```

A *substringAfter()* a szeparátor utáni String részletet szolgáltatja. A *substringBeforeLast()* és *substringAfterLast()* hasonló működéssel bír, de fel van készítve arra, hogy a szeparátor több alkalommal is előfordul és mindig az utolsó előfordulást fogja referenciaként tekinteni:

```
StringUtils.substringBeforeLast("abcba", "b") = "abc"
StringUtils.substringAfterLast("abcba", "b") = "a"
```

A *substringBetween()* a 2. paraméterben megadott String, mint 2 befoglaló közötti Stringet adja vissza. Létezik egy másik alakja is, ahol a nyitó és záró befoglaló Stringek különbözőek lehetnek.

```
StringUtils.substringBetween(null, *)            = ➥
    null
StringUtils.substringBetween("", "")             = ""
StringUtils.substringBetween("", "tag")          = ➥
    null
StringUtils.substringBetween("tagabctag", null)  = ➥
    null
StringUtils.substringBetween("tagabctag", "")    = ""
StringUtils.substringBetween("tagabctag", "tag") = "➥
    abc"
```

Az *overlay()* metódus 2 String egymást való takarásából (a 2. takarja az 1. String-et) adódó eredmény String-et ad vissza. Itt a 3. és 4. paraméterek a takarás kezdő és végző pozíciói.

```
StringUtils.overlay(null, *, *, *)            = null
StringUtils.overlay("", "abc", 0, 0)          = "abc"
StringUtils.overlay("abcdef", null, 2, 4)     = "abef"
StringUtils.overlay("abcdef", "", 2, 4)       = "abef"
StringUtils.overlay("abcdef", "", 4, 2)       = "abef"
StringUtils.overlay("abcdef", "zzzz", 2, 4)   = "➥
    abzzzzef"
StringUtils.overlay("abcdef", "zzzz", 4, 2)   = "➥
    abzzzzef"
StringUtils.overlay("abcdef", "zzzz", -1, 4)  = "➥
    zzzzef"
StringUtils.overlay("abcdef", "zzzz", 2, 8)   = "➥
    abzzzz"
StringUtils.overlay("abcdef", "zzzz", -2, -3) = "➥
    zzzzabcdef"
StringUtils.overlay("abcdef", "zzzz", 8, 10)  = "➥
    abcdefzzzz"
```

Egy String részlet cseréje

A cserére szolgáló *replace()* metódus is sokféle megvalósításban áll a rendelkezésünkre, de ez a leggyakrabban használt alakja:

```
public static String replace(String text,
                             String searchString,
                             String replacement)
```

A használatára egy példa:

```
oStr = StringUtils.replace("aaabbaaa", "bb", "cc");

// Eredmény
aaaccaaa
```

Egy 4. paraméterben is meg lehet adni, hogy maximum hány cserét engedélyezünk. A következő példában az „a” 3 alkalommal lenne cserélve, de a megadott 2 érték csak a balról számított első kettőt engedi.

```
StringUtils.replace("abaa", "a", "z", 2)   = "zbza"
```

Amennyiben az engedélyezett cserék száma 1, úgy erre a *replaceOnce()* függvény a legalkalmasabb, aminek nincs 4. paramétere, mert az konstans 1. A *replaceEach()* egy hatékonyságnövelő metódus, mert sokszor több String illeszkedést kell cserélni egy forrás Stringre:

```
String replaceEach(String text,
                   String[] searchList,
                   String[] replacementList)
```

Látható, hogy ekkor a kereső kifejezések és a csere értékek is egy-egy tömbben vannak. Nézzünk néhány példát!

```
StringUtils.replaceEach(null, *, *)        = null
StringUtils.replaceEach("", *, *)          = ""
StringUtils.replaceEach("aba", null, null) = "aba"
StringUtils.replaceEach("aba", new String[0], null) = ➥
    "aba"
StringUtils.replaceEach("aba", null, new String[0]) = ➥
    "aba"
StringUtils.replaceEach("aba", new String[]{"a"}, null➥
    ) = "aba"
StringUtils.replaceEach("aba", new String[]{"a"}, new ➥
    String[]{""}) = "b"
StringUtils.replaceEach("aba", new String[]{null}, new➥
     String[]{"a"}) = "aba"
StringUtils.replaceEach("abcde", new String[]{"ab", "d➥
    "}, new String[]{"w", "t"}) = "wcte"
StringUtils.replaceEach("abcde", new String[]{"ab", "d➥
    "}, new String[]{"d", "t"}) = "dcte"
```

Törlés a Stringből

A *remove()* metódus törli a megadott karakter vagy String összes előfordulását.

```
StringUtils.remove(null, *)        = null
StringUtils.remove("", *)          = ""
StringUtils.remove(*, null)        = *
StringUtils.remove(*, "")          = *
StringUtils.remove("queued", "ue") = "qd"
StringUtils.remove("queued", "zz") = "queued"
```

A *removeEnd()* csak a String végéről törli az előfordulást, persze csak amennyiben ott megtalálható. Ellenkező esetben visszaadja az eredeti forrás Stringet.

```
StringUtils.removeEnd(null, *)      = null
StringUtils.removeEnd("", *)        = ""
StringUtils.removeEnd(*, null)      = *
StringUtils.removeEnd("www.domain.com", ".com.")  = "➥
    www.domain.com"
StringUtils.removeEnd("www.domain.com", ".com")   = "➥
    www.domain"
```

```
StringUtils.removeEnd("www.domain.com", "domain") = "➥
    www.domain.com"
StringUtils.removeEnd("abc", "")    = "abc"
```

A *removeEndIgnoreCase()* hasonlót csinál, de nem veszi figyelembe a kisbetű/nagybetű eltéréseket. A *removeStart()* és *removeStartIgnoreCase()* függvények a String elejével művelik mindezt. A *trim()* család a klasszikus String eleje és vége törlést valósítja meg, amikor az valamilyen vezérlő karakter (ASCII kód $\leqq 32$).

```
StringUtils.trim("␣␣␣␣abc␣␣␣␣") = "abc"
StringUtils.trim("␣␣␣␣␣")       = ""
```

A *trimToNull()* üres String helyett *null* értéket ad vissza:

```
StringUtils.trimToNull(null)          = null
StringUtils.trimToNull("")            = null
StringUtils.trimToNull("␣␣␣␣␣")       = null
StringUtils.trimToNull("abc")         = "abc"
StringUtils.trimToNull("␣␣␣␣abc␣␣␣␣") = "abc"
```

A *trimToEmpty()* pedig mindig üres Stringet ad vissza, amikor a fentiek *null*-t adnának. A *strip()* család hasonló a *trim()*-hez, de a vezérlőkarakterek helyett a whitespace-t figyeli. A *stripAll()* String tömbön is elvégzi a feladatot. A *stripAccents()* érdekes lehetőség, mert képes egy olyan Stringet visszaadni, amiben már nincsenek ékezetes karakterek (diacritics=ékezetek):

```
StringUtils.stripAccents("éclair") = "eclair"
```

String készítés ismétléssel

A *repeat()* az 1. paraméterben megadott mintából olyan eredmény karaktersorozatot készít, ami annak a 2. paraméterben megadott számú ismétlésével áll elő. A 4. példa azt is megmutatja, hogy amennyiben a 3 paraméteres változatot használjuk, úgy szeparátor is megadható.

```
StringUtils.repeat("ab", 2) = "abab"
StringUtils.repeat(null, 2) = null
StringUtils.repeat("", 0)   = ""
StringUtils.repeat("", 2)   = ""
StringUtils.repeat("?", ",␣", 3)  = "?,␣?,␣?"
```

String darabokra vágása

Egy forrás String több darabra vágása a *split()* családdal valósítható meg. Alapesetben az 1. paraméter a szétvágandó String, a 2. pedig a szeparátor. Az eredmény egy String tömbbe kerül.

```
StringUtils.split("aa.bb.cc", '.')    = ["aa", "bb", "➥
    cc"]
```

Az 1 paraméteres *split()* a whitespace karakterek mentén vágja fel a String inputját. Lehetőség van egy 3. *split()* paraméter megadására is, ami az előállított tömb elemszámának a maximumát is figyelembe veszi:

```
StringUtils.split("ab:cd:ef", ":", 2)    = ["ab", "cd:➥
    ef"]
```

A *splitByWholeSeparator()* egy egész mintát képes elválasztóként használni:

```
StringUtils.splitByWholeSeparator("ab-!-cd-!-ef", "-!-➥
    ") = ["ab", "cd", "ef"]
```

A többi split változatot az API leírásból javasoljuk megnézni, azokra ritkábban lehet szükség.

Középre igazítás

A *center()* metódusnak több változata is van, de mindegyiknek az a célja, hogy egy szöveget középre igazítson.

```
String oStr = null;
oStr = StringUtils.center("aaa", 10);
System.out.println( oStr );

// Eredmény (10 hosszú String):
   aaa
```

Megadhatjuk 3. paraméterként a kitöltő karaktert is:

```
String oStr = null;
oStr = StringUtils.center("aaa", 10, "@");
System.out.println( oStr );
oStr = StringUtils.center("aaa", 10, "@X");
System.out.println( oStr );

// Eredmény:
@@@aaa@@@@
@X@aaa@X@X
```

String megfordítása

A *reverse()* metódus az input Stringet visszafelé írva adja vissza:

```
StringUtils.reverse(null)  = null
StringUtils.reverse("")    = ""
StringUtils.reverse("bat") = "tab"
```

Ennek létezik egy *reverseDelimited()* nevű változata, ahol egy szeparátort is meg lehet adni,

a Stringet pedig a csoportok visszafelé olvasásával adja vissza:

```
StringUtils.reverseDelimited("aa.b2.c1", '.');

// Eredmény:
c1.b2.aa
```

Összehasonlítás

Az *equals()* 2 String egyenlőségét vizsgálja:

```
StringUtils.equals(null, null)   = true
StringUtils.equals(null, "abc")  = false
StringUtils.equals("abc", null)  = false
StringUtils.equals("abc", "abc") = true
StringUtils.equals("abc", "ABC") = false
```

Ennek létezik *equalsIgnoreCase()* alakja is. Megvizsgálhatjuk, hogy egy String bizonyos karaktersorozattal kezdődik vagy végződik. Erre szolgál az *endsWith()* és *startsWith()* metódusok:

```
StringUtils.endsWith(null, null)      = true
StringUtils.endsWith(null, "def")     = false
StringUtils.endsWith("abcdef", null)  = false
StringUtils.endsWith("abcdef", "def") = true
StringUtils.endsWith("ABCDEF", "def") = false
StringUtils.endsWith("ABCDEF", "cde") = false

StringUtils.startsWith(null, null)      = true
StringUtils.startsWith(null, "abc")     = false
StringUtils.startsWith("abcdef", null)  = false
StringUtils.startsWith("abcdef", "abc") = true
StringUtils.startsWith("ABCDEF", "abc") = false
```

Ezen 2 alap metódusnak lézetik néhány változata:

- *startsWithIgnoreCase()*
- *endWithIgnoreCase()*
- *endsWithAny()*
- *startWithAny()*

Az első két függvény egyértelmű. Az „any” metódusok azt vizsgálják, hogy a tömbben megadott esetek valamelyikével kezdődik vagy végződik-e a String.

```
StringUtils.endsWithAny(null, null)      = false
StringUtils.endsWithAny(null, new String[] {"abc"})  = false
StringUtils.endsWithAny("abcxyz", null)     = false
StringUtils.endsWithAny("abcxyz", new String[] {""}) = true
StringUtils.endsWithAny("abcxyz", new String[] {"xyz"}) = true
StringUtils.endsWithAny("abcxyz", new String[] {null, "xyz", "abc"}) = true
```

A *getCommonPrefix()* egy érdekes lehetőség, mert képes megvizsgálni egy String tömb összes elemét és amennyiben van, úgy visszaadja a közös prefixüket.

```
StringUtils.getCommonPrefix(null) = ""
StringUtils.getCommonPrefix(new String[] {}) = ""
StringUtils.getCommonPrefix(new String[] {"abc"}) = "abc"
StringUtils.getCommonPrefix(new String[] {null, null}) = ""
StringUtils.getCommonPrefix(new String[] {"", ""}) = ""
StringUtils.getCommonPrefix(new String[] {"", null}) = ""
StringUtils.getCommonPrefix(new String[] {"abc", null, null}) = ""
StringUtils.getCommonPrefix(new String[] {null, null, "abc"}) = ""
StringUtils.getCommonPrefix(new String[] {"", "abc"}) = ""
StringUtils.getCommonPrefix(new String[] {"abc", ""}) = ""
StringUtils.getCommonPrefix(new String[] {"abc", "abc"}) = "abc"
StringUtils.getCommonPrefix(new String[] {"abc", "a"}) = "a"
StringUtils.getCommonPrefix(new String[] {"ab", "abxyz"}) = "ab"
StringUtils.getCommonPrefix(new String[] {"abcde", "abxyz"}) = "ab"
StringUtils.getCommonPrefix(new String[] {"abcde", "xyz"}) = ""
StringUtils.getCommonPrefix(new String[] {"xyz", "abcde"}) = ""
StringUtils.getCommonPrefix(new String[] {"i_am_a_machine", "i_am_a_robot"}) = "i_am_a_"
```

Rész String tartalmazás

A *contains()* megvizsgálja, hogy az első String tartalmazza-e a másodikat:

```
boolean b = StringUtils.contains("Almafa", "ma");
System.out.println( b );

// Eredmény:true
```

A *containsIgnoreCase()* metódus változat figyelmen kívül hagyja a kisbetű/nagybetű különbségeket. A *containsWhitespace()* metódus megvizsgálja, hogy egyetlen String paramétere tartalmaz-e *whitespace* karaktert. A *containsAny()* kicsit rugalmasabb, mert fel lehet sorolni azokat az eseteket, amiknek a tartalmazását vizsgáljuk:

```
boolean b;
b = StringUtils.containsAny("Alma és körte", 'a', 'ö',
    't' );
System.out.println( b );

// vagy:
boolean b = StringUtils.containsAny("Alma és körte", "
    aöt" );

// Eredmény: true
```

Ennek az ellentéte a *containsNone()* metódus, aminek ugyanilyen a paraméterezése. A *containsOnly()* azért fontos, mert ezzel azt tudjuk levizsgáltatni, hogy az 1. paraméter csak azokat a karaktereket tartalmazza-e, amit a 2. paraméterben megadtunk.

Keresés

A String class is tartalmaz egy *indexOf()* metódust, ezért felmerülhet a kérdés, hogy a *StringUtils* ugyanilyen nevű módszere miben tud többet. Azt mindjárt az elején fontos megállapítani, hogy ez a változat amikor csak lehet delegálja tovább a feladatot a *String.indexOf(String, int)* metódusnak.

```
// java.lang.NullPointerException lesz:
int pos = "alma".indexOf(null);

// Jól működik null-ra is, -1 lesz:
StringUtils.indexOf("alma", null);
```

Az *indexOf()* függvénynek létezik egy olyan változata is, ahol a keresés kezdőpozícióját is meg lehet adni. Az *ordinalIndexOf()* által megvalósított keresésnél még azt is meg lehet adni, hogy a keresett karaktersorozat hányadik előfordulásának pozíciója érdekel bennünket:

```
StringUtils.ordinalIndexOf(null, *, *)          = -1
StringUtils.ordinalIndexOf(*, null, *)          = -1
StringUtils.ordinalIndexOf("", "", *)           = 0
StringUtils.ordinalIndexOf("aabaabaa", "a", 1)  = 0
StringUtils.ordinalIndexOf("aabaabaa", "a", 2)  = 1
StringUtils.ordinalIndexOf("aabaabaa", "b", 1)  = 2
StringUtils.ordinalIndexOf("aabaabaa", "b", 2)  = 5
StringUtils.ordinalIndexOf("aabaabaa", "ab", 1) = 1
StringUtils.ordinalIndexOf("aabaabaa", "ab", 2) = 4
StringUtils.ordinalIndexOf("aabaabaa", "", 1)   = 0
StringUtils.ordinalIndexOf("aabaabaa", "", 2)   = 0
```

Az *indexOfAny()* több lehetséges keresendő érték közül bármelyikre való találat indexét adja vissza:

```
// Ez 3 lesz, mert b vagy y a keresendő:
int pos = StringUtils.indexOfAny("zzabyycdxx", 'b','y'
    );

// String-re
StringUtils.indexOfAny(null, *)             = -1
StringUtils.indexOfAny("", *)               = -1
StringUtils.indexOfAny(*, null)             = -1
StringUtils.indexOfAny(*, "")               = -1
StringUtils.indexOfAny("zzabyycdxx", "za") = 0
StringUtils.indexOfAny("zzabyycdxx", "by") = 3
StringUtils.indexOfAny("aba","z")           = -1
```

Az *indexOfAnyBut()* azt a pozíciót keresi, ami először nem illeszkedik balról a mintára.

```
StringUtils.indexOfAnyBut("zzabyycdxx", new char[] {'z
    ', 'a'} ) = 3
StringUtils.indexOfAnyBut("aba", new char[] {'z'} )
                = 0
StringUtils.indexOfAnyBut("aba", new char[] {'a', 'b'}
    )       = -1

StringUtils.indexOfAnyBut("zzabyycdxx", "za") = 3
StringUtils.indexOfAnyBut("zzabyycdxx", "")   = -1
StringUtils.indexOfAnyBut("aba","ab")         = -1
```

Az *indexOfDifference()* a paraméterében lévő 2 Stringre visszaadja azt a pozíciót, ahol a prefixük először nem egyeznek meg egymással. Az *indexOfIgnoreCase()* nem tesz különbséget kis és nagybetű között. A *lastIndexOf()* az utolsó előfordulás indexét adja vissza:

```
int pos = StringUtils.lastIndexOf("sbxabaddxaddd", "xa
    ");

// Eredmény: 8
```

Itt jegyezzük meg, hogy ezek a metódusok egy *CharSequence* interface-szel rendelkező objektumon tudnak dolgozni, ilyen többek között a jól ismert *String* is. A már ismert *lastIndexOfAny()*, *lastIndexOfIgnoreCase()* és *lastOrdinalIndexOf()* változatok itt is elérhetőek.

Logikai lekérdezések

Az *is...()* kezdetű metódusok különféle kérdéseket tesznek fel a paraméterül kapott String-nek, amikre igen/nem válasz érkezhet. Az *isAllLowerCase()* akkor és csak akkor (továbbiakban: a.cs.a), ha minden karaktere kisbetű.

```
StringUtils.isAllLowerCase(null)   = false
StringUtils.isAllLowerCase("")     = false
StringUtils.isAllLowerCase("␣␣")   = false
StringUtils.isAllLowerCase("abc")  = true
StringUtils.isAllLowerCase("abC") = false
```

Az *isAllUpperCase()* a.cs.a igaz, ha minden karaktere nagybetű. Az *isAlpha()* a.cs.a igaz, ha legalább 1 hosszú és minden karaktere betű. Az *isAlphanumeric()* a.cs.a igaz, ha legalább 1 hosszú és minden karaktere számjegy vagy betű:

```
StringUtils.isAlphanumeric(null)   = false
StringUtils.isAlphanumeric("")     = false
StringUtils.isAlphanumeric("␣␣")   = false
StringUtils.isAlphanumeric("abc")  = true
StringUtils.isAlphanumeric("ab␣c") = false
StringUtils.isAlphanumeric("ab2c") = true
StringUtils.isAlphanumeric("ab-c") = false
```

Az *isAlphanumericSpace()* használati eseteit a következő tesztesetekből láthatjuk:

```
StringUtils.isAlphanumericSpace(null)   = false
StringUtils.isAlphanumericSpace("")     = true
StringUtils.isAlphanumericSpace("␣␣")   = true
StringUtils.isAlphanumericSpace("abc")  = true
StringUtils.isAlphanumericSpace("ab␣c") = true
StringUtils.isAlphanumericSpace("ab2c") = true
StringUtils.isAlphanumericSpace("ab-c") = false
```

Az *isAlphaSpace()* a.cs.a igaz, ha betűt vagy szóközt tartalmaz és nem *null*. Itt az üres String *true* értéket ad vissza. Az *isAsciiPrintable()* a.cs.a fog igazzal visszatérni, ha a String nyomtatható és csak eredeti ASCII karaktereket tartalmaz. Aki ránéz az *isBlank()* és *isNotBlank* metódusokhoz adott példákra, egyből rájön a működésére:

```
StringUtils.isBlank(null)      = true
StringUtils.isBlank("")        = true
StringUtils.isBlank("␣")       = true
StringUtils.isBlank("bob")     = false
StringUtils.isBlank("␣␣bob␣␣") = false

StringUtils.isNotBlank(null)      = false
StringUtils.isNotBlank("")        = false
StringUtils.isNotBlank("␣")       = false
StringUtils.isNotBlank("bob")     = true
StringUtils.isNotBlank("␣␣bob␣␣") = true
```

Ugyanezt mondhatjuk el az *isEmpty()* és *isNotEmpty()* esetére is:

```
StringUtils.isEmpty(null)      = true
StringUtils.isEmpty("")        = true
StringUtils.isEmpty("␣")       = false
StringUtils.isEmpty("bob")     = false
StringUtils.isEmpty("␣␣bob␣␣") = false

StringUtils.isNotEmpty(null)      = false
StringUtils.isNotEmpty("")        = false
StringUtils.isNotEmpty("␣")       = true
StringUtils.isNotEmpty("bob")     = true
StringUtils.isNotEmpty("␣␣bob␣␣") = true
```

Az *isNumeric()* természetesen azt dönti el, hogy egy Stringben csak számjegyek vannak-e:

```
StringUtils.isNumeric(null)   = false
StringUtils.isNumeric("")     = false
StringUtils.isNumeric("␣␣")   = false
StringUtils.isNumeric("123")  = true
StringUtils.isNumeric("12␣3") = false
StringUtils.isNumeric("ab2c") = false
StringUtils.isNumeric("12-3") = false
StringUtils.isNumeric("12.3") = false
```

Az *isNumericSpace()* azt is megengedi, hogy a String üres legyen vagy szóközt tartalmazzon. Az *isWhitespace()* a.cs.a igaz, ha az csak whitespace karaktereket tartalmaz vagy empty a String. A *null* értékre hamis.

Összekapcsolás

Szintén alapfeladat néhány String vagy karakter összeragasztása, amit a *join()* metódus családdal tudunk különféle inputokra elvégezni. Látható, hogy az összeragasztandó értékek vagy változó számú paraméterként vagy egy tömbben lehetnek. Amennyiben a tömbös verziónál megadunk egy 2. paramétert is, úgy az egy elválasztóként fog funkcionálni és felsorolásként lesz a join eredménye megvalósulva. Lehetőség van ekkor egy 3. és 4. paraméter megadására is, amikor azok a tömb figyelembe vett kezdő és záró elemét adják meg.

```
StringUtils.join("a", "b", "c"); = abc
StringUtils.join(new String[] {"a", "á", "b"}); = aáb

StringUtils.join(new String[] {"a", "b", "c"}, ";"); =➥
     a;b;c
```

A *join()* első paramétere egy *Iterator<?>* vagy *Iterable<?>* is lehet.

Alapértelmezett értékek üres Stringre

A *defaultIfBlank()* az 1. paraméter vizsgálata alapján visszaadja a 2. paramétert, amennyiben az „ „, üres vagy *null*. Minden más esetben az 1. paraméterben kapott String-et kapjuk vissza.

```
StringUtils.defaultIfBlank(null, "NULL")  = "NULL"
StringUtils.defaultIfBlank("", "NULL")    = "NULL"
StringUtils.defaultIfBlank(" ", "NULL")   = "NULL"
StringUtils.defaultIfBlank("bat", "NULL") = "bat"
StringUtils.defaultIfBlank("", null)      = null
```

Az is lehet, hogy csak az üres (empty) és *null* értékek alapján szeretnénk a fenti működést, erre szolgál a *defaultIfEmpty()*. Valószínű, hogy legtöbbet a *defaultString()* metódus lesz használva, ami csak a *null* értéket vizsgálja:

```
StringUtils.defaultString(null, "NULL")  = "NULL"
StringUtils.defaultString("", "NULL")    = ""
StringUtils.defaultString("bat", "NULL") = "bat"
```

Ennek létezik 1 paraméteres változata is, amikor *null* esetén mindig üres String az eredmény:

```
StringUtils.defaultString(null)  = ""
StringUtils.defaultString("bat") = "bat"
```

Törlés

A *deleteWhitespace()* metódus törli a fehér szóközöket a String-ből, ahogy a példa mutatja:

```
StringUtils.deleteWhitespace(null)         = null
StringUtils.deleteWhitespace("")           = ""
StringUtils.deleteWhitespace("abc")        = "abc"
StringUtils.deleteWhitespace("  ab  c  ") = "abc"
```

Előfordulás számlálás

A *countMatches()* visszaadja, hogy az 1. paraméterben hány alkalommal fordult elő a 2. paraméter.

```
StringUtils.countMatches(null, *)       = 0
StringUtils.countMatches("", *)         = 0
StringUtils.countMatches("abba", null)  = 0
StringUtils.countMatches("abba", "")    = 0
StringUtils.countMatches("abba", "a")   = 2
StringUtils.countMatches("abba", "ab")  = 1
StringUtils.countMatches("abba", "xxx") = 0
```

Rövidítés

Az *abbreviate()* metódus képes egy Stringet lerövidíteni, a folytatását ...-al jelölni:

```
String str="Alma a fa alatt, nyári piros alma.";
String oStr = StringUtils.abbreviate(str, 10);
System.out.println( oStr );

// Eredmény:
Alma a ...
```

Amennyiben az *abbreviate()* 3 paraméteres változatát adjuk meg, úgy ezzel azt határozzuk meg, hogy honnan akarjuk a Stringet látni, előtte és utána ... lesz:

```
oStr = StringUtils.abbreviate(str, 5, 15);
```

Ekkor az eredmény ez lesz: *...a fa alat...* Végül létezik az *abbreviateMiddle()* metódus, amelyik a String közepét cseréli a 2. paraméterben megadott szövegre, miközben az eredmény String hossza a 3. paraméterrel van specifikálva:

```
String str="Alma a fa alatt, nyári piros alma.";
String oStr = null;
oStr=StringUtils.abbreviateMiddle(str, " @csere@ ", 
    15);

// Eredmény:
Alm @csere@ ma.
```

Az újsor karakter törlése

A *chomp()* segítségével törölhetjük a String végéről az újsor jellegű karaktereket:

```
// Újsor karakterek:
// "\n", "\r", or "\r\n"

String oStr = null;
oStr = StringUtils.chomp("alma\n");
```

Az utolsó karakter törlése

A *chop()* eltávolítja a String utolsó karakterét, ahogy ez a példa is mutatja:

```
String oStr = null;
oStr = StringUtils.chop("alma");
System.out.println( oStr );

// Eredmény:
alm
```

Nagy és kisbetű

A *capitalize()* metódus egy String első karakterét nagybetűre alakítja.

```
oStr = StringUtils.capitalize("alma a fa alatt");
System.out.println( oStr );

// Eredmény:
Alma a fa alatt
```

A *swapCase()* a kisbetűt nagyra, a nagyot kicsire konvertálva adja vissza a forrás Stringet. Az *uncapitalize()* a fentebb bemutatott *capitalize()* ellentétes párja.

Két String különbsége

Ezt a műveletet a *difference()* metódus képes elvégezni:

```
StringUtils.difference(null, null) = null
StringUtils.difference("", "") = ""
StringUtils.difference("", "abc") = "abc"
StringUtils.difference("abc", "") = ""
StringUtils.difference("abc", "abc") = ""
StringUtils.difference("ab", "abxyz") = "xyz"
StringUtils.difference("abcde", "abxyz") = "xyz"
StringUtils.difference("abcde", "xyz") = "xyz"
```

Az *indexOfDifference()* metódus azt a pozíciót adja vissza, ahol a paraméter két darab String-je elkezd különbözni. Létezik egy olyan függvény, ami visszaadja azt a számot, hogy az egyik String-ben mennyi karakter változtatás (törlés, beszúrás, helyettesítés) lenne szükséges, hogy visszakapjuk a másik Stringet: *getLevenshteinDistance()*. Nézzünk erre is példákat!

```
StringUtils.getLevenshteinDistance(null, *)             = IllegalArgumentException
StringUtils.getLevenshteinDistance(*, null)             = IllegalArgumentException
StringUtils.getLevenshteinDistance("","")               = 0
StringUtils.getLevenshteinDistance("","a")              = 1
StringUtils.getLevenshteinDistance("aaapppp", "")       = 7
StringUtils.getLevenshteinDistance("frog", "fog")       = 1
StringUtils.getLevenshteinDistance("fly", "ant")        = 3
StringUtils.getLevenshteinDistance("elephant", "hippo") = 7
StringUtils.getLevenshteinDistance("hippo", "elephant") = 7
StringUtils.getLevenshteinDistance("hippo", "zzzzzzzz") = 8
StringUtils.getLevenshteinDistance("hello", "hallo")    = 1
```

Számok kezelése - NumberUtils

String konvertálása szám osztályra

A *NumberUtils* osztály *create...()* metódusai egy Stringet kapnak és visszaadnak egy olyan objektumot, ami ez alapján hozható létre. Az alábbiakban felsoroljuk ezeket a metódusokat:

```
public static BigDecimal createBigDecimal(String str)
public static BigInteger createBigInteger(String str)
public static Double createDouble(String str)
public static Integer createInteger(String str)
public static Long createLong(String str)
public static Float createFloat(String str)

public static Number createNumber(String str)
                          throws ➥
                              NumberFormatException
```

A *java.lang.Number* ősosztálya az összes megelőző class-nak.

String konvertálása szám skalárra

A *to...()* metódus család skalár számokat ad vissza a paraméterben megadott String alapján. Amennyiben megadunk egy 2. paramétert is, úgy az egy alapértelmezett érték arra az esetre, ha a konverzió nem lett sikeres. Nézzük őket!

```
public static int toInt(String str)
public static int toInt(String str, int defaultValue)
public static long toLong(String str)
public static long toLong(String str, long ➥
    defaultValue)
public static double toDouble(String str)
public static double toDouble(String str, double ➥
    defaultValue)
public static float toFloat(String str)
public static float toFloat(String str, float ➥
    defaultValue)
public static short toShort(String str)
public static short toShort(String str, short ➥
    defaultValue)
public static byte toByte(String str)
public static byte toByte(String str, byte ➥
    defaultValue)
```

Szélsőérték visszaadása

A *max()* és *min()* metódusok inputként valamilyen számtömböt kapnak és a tömb elemtípusának megfelelő értéket adnak vissza.

Logikai lekérdezések

Az *isDigits()* leellenőrzi, hogy a paraméterül kapott String csak számjegyet tartalmaz-e. A *null* és az üres String is hamissal tér vissza. Az *isNumber()* azt nézi meg, hogy a forrás String átalakítható-e egy Java *Number* objektummá.

A dátum és idő kezelése

A Java nyelv beépített *Date* és *Calendar* osztályai mutatnak némi hiányosságot. Az ebben a

pontban bemutatott lehetőségek előrelépést jelentenek ebben, de az igazi megoldást a következő fejezetben bemutatott *Joda.org* könyvtár fogja jelenteni. A Java környezet 2 fontos osztályt tartalmaz a dátumokkal kapcsolatosan:

- *java.util.Calendar* (*java.util.GregorianCalendar*)
- *java.util.Date*

Ezekről többet a következő cikkben írunk. Az *org.apache.commons.lang3.time* csomag a következő osztályokkal segíti a programozót:

- *DateUtils*: Közhasznú *Date* és *Calendar* rutinok
- *DateFormatUtils*, *DurationFormatUtils* és *FastDateFormat*: A dátum Stringgé alakításának támogatása
- *StopWatch*: Időzítés kezelés támogatása

A *DateUtils* néhány fontos előre felvett konstanst is tartalmaz, például az 1 napra eső milliszekundumok száma.

A dátumok növelése és csökkentése

A következő kód a *tz* változóban egy budapesti időzónát reprezentál, a *c Calendar* objektumot ezzel hoztuk létre. A *d* a *Calendar* objektumtól lekért időpillanatot reprezentálja, gyakorlatilag egy olyan eltárolására képes *Date* objektum. A *DateUtils* osztály *addDays()* metódusa egy dátumhoz napokat tud hozzáadni, ami lehet negatív szám is. Esetünkben most 3 nappal növeltük a dátumot.

```
TimeZone tz = TimeZone.getTimeZone("Europe/Budapest");
Calendar c = GregorianCalendar.getInstance( tz );
SimpleDateFormat dt = new SimpleDateFormat("yyyy-MM-dd␣HH:mm:ss");
dt.setTimeZone(tz);

Date d = c.getTime();
// Eredmény: 2013-05-29 17:59:22
System.out.println( dt.format( d ) );

d = DateUtils.addDays(d, 3);
// Eredmény: 2013-06-01 17:59:22
System.out.println( dt.format( d ) );
```

A következőkben csak megadjuk a többi hasonló metódus deklarációját:

```
public static Date addYears(Date date, int amount);
public static Date addMonths(Date date, int amount);
public static Date addWeeks(Date date, int amount);
public static Date addHours(Date date, int amount);
public static Date addMinutes(Date date, int amount);
public static Date addSeconds(Date date, int amount);
public static Date addMilliseconds(Date date, int amount);
```

Dátumok beállítása

A dátumok beállítása azt jelenti, hogy egy input *Date* objektum valamelyik dátum komponensét átállítjuk és az így kialakított dátumot adjuk vissza, miközben az eredeti dátum ettől nem változik meg. A lenti példában például a pillanatnyi időponthoz rendelt *Date* objektum hónapját decemberre állítjuk:

```
TimeZone tz = TimeZone.getTimeZone("Europe/Budapest");
Calendar c = GregorianCalendar.getInstance( tz );
SimpleDateFormat dt = new SimpleDateFormat("yyyy-MM-dd␣HH:mm:ss");
dt.setTimeZone(tz);

Date d = c.getTime();
// Eredmény: 2013-05-30 08:34:49
System.out.println( dt.format( d ) );

d = DateUtils.setMonths(d, 11);
// Eredmény: 2013-12-27 07:34:49
System.out.println( dt.format( d ) );
```

A további beállító metódusok a következőek:

```
public static Date setYears(Date date, int amount);
public static Date setDays(Date date, int amount);
public static Date setHours(Date date, int amount);
public static Date setMinutes(Date date, int amount);
public static Date setSeconds(Date date, int amount);
public static Date setMilliseconds(Date date, int amount);
```

Szövegből dátum

A *parseDate()* és *parseDateStrictly()* metódusok egy dátum és egy minta *String* alapján előállítanak egy *Date* objektumot. Az alábbi példából láthatjuk, hogy több mintát is megadhatunk, a metódus végigpróbálja az összeset, hogy sikerrel járjon.

```
String[] pattern = new String[]
{
  "yyyy-MM-dd␣HH:mm:ss",
  "yyyy-MM-dd"
};
Date pd = DateUtils.parseDate("2015-03-01␣08:07:02", pattern);
```

A *parseDateStrictly()* gyakorlatilag ugyanezt csinálja, de az elemzésnél szigorúbb, azaz egy-egy résznél nem próbálja kitalálni a jelentés és például ilyen részletet nem fogad el: *February 942, 1996*. A gyakorlatban emiatt érdemes a 2. változatot használni.

Dátumok kerekítése

Azt érdemes mindig szem előtt tartani, hogy a Java *Date* class egy *long* típusban reprezentálja a dátumot, mint időpontot. Emiatt a *Date* nem is tudhatja magáról, hogy ő milyen időzónában mutat egy konkrét értéket, de nem is erre lett kitalálva. A feladata csupán annyi, hogy egy konkrét értéket eltároljon. Ennyi megjegyzés után nézzük meg a *ceiling()* metódust, ami úgy működik, hogy egy dátumra megadjuk azt a részt amitől „jobbra" eső értékeket kinullázunk és a megadott részt inkrementáljuk. Nevezhetnénk ezt a dátumok felfelé kerekítésének is, ahol a 2. paraméter a kerekítési pontot jelöli ki. Nézzünk rá példákat!

```
String[] pattern = new String[] { "yyyy-MM-dd HH:mm:ss"};
Date pd = DateUtils.parseDateStrictly("2015-03-01 08:07:02", pattern);
// Eredmény: Sun Mar 01 08:07:02 GMT 2015
System.out.println( pd );

pd = DateUtils.ceiling(pd, Calendar.HOUR);
// Eredmény: Sun Mar 01 09:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.ceiling(pd, Calendar.DAY_OF_MONTH);
// Eredmény: Mon Mar 02 00:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.ceiling(pd, Calendar.MONTH);
// Eredmény: Wed Apr 01 00:00:00 GMT 2015
System.out.println( pd );
```

Amikor például a *DAY_OF_MONTH* értéket adjuk meg, akkor ez a nap pozícióján való felfelé kerekítést jelentett, ahogy láthattuk is. A *ceiling()* függvénynek van olyan változata is, ami *Calendar* objektumon működik. A *round()* metódus számára szintén a releváns dátum komponenst (hó, nap, óra, perc, ...) kell megadni, de ez már a kerekítés logikája szerint fog működni, ahogy a következő példák mutatják:

```
String[] pattern = new String[] { "yyyy-MM-dd HH:mm:ss" };
Date pd = DateUtils.parseDateStrictly("2015-03-01 08:29:02", pattern);
// Eredmény: Sun Mar 01 08:29:02 GMT 2015
System.out.println( pd );

pd = DateUtils.round(pd, Calendar.HOUR);
// Eredmény: Sun Mar 01 08:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.parseDateStrictly("2015-03-01 08:30:02", pattern);
// Eredmény: Sun Mar 01 08:30:02 GMT 2015
System.out.println( pd );
pd = DateUtils.round(pd, Calendar.HOUR);
// Eredmény: Sun Mar 01 09:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.parseDateStrictly("2015-03-16 08:30:02", pattern);
pd = DateUtils.round(pd, Calendar.MONTH);
// Eredmény: Sun Mar 01 00:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.parseDateStrictly("2015-03-17 08:30:02", pattern);
pd = DateUtils.round(pd, Calendar.MONTH);
// Eredmény: Wed Apr 01 00:00:00 GMT 2015
System.out.println( pd );
```

A fenti példákban látható, hogy mikor kerekítünk lefelé, illetve felfelé, amikor a releváns mező az óra vagy a hónap. A *truncate()* gondolkodás nélkül kinulláz minden olyan dátum mezőt, ami a megadott komponens után van:

```
String[] pattern = new String[] { "yyyy-MM-dd HH:mm:ss"};
Date pd = DateUtils.parseDateStrictly("2015-03-01 08:07:02", pattern);
// Eredmény: Sun Mar 01 08:07:02 GMT 2015
System.out.println( pd );

pd = DateUtils.truncate(pd, Calendar.HOUR);
// Eredmény: Sun Mar 01 08:00:00 GMT 2015
System.out.println( pd );

pd = DateUtils.parseDateStrictly("2015-03-16 08:30:02", pattern);
pd = DateUtils.truncate(pd, Calendar.MONTH);
// Eredmény: Sun Mar 01 00:00:00 GMT 2015
System.out.println( pd );
```

Ennek is létezik *Calendar* objektumon működő változata.

Az eddig eltelt idő

A fragment metódus család segítségével le tudjuk kérdezni az év, hónap, ... eddig eltelt idejét különféle időegységekben (nap, óra, perc, ...). A következő példák tanulmányozásával mindez gyorsan világos lesz:

```
TimeZone tz = TimeZone.getTimeZone("Europe/Budapest");
Calendar c = GregorianCalendar.getInstance( tz );
SimpleDateFormat dt = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssz");
dt.setTimeZone(tz);
// Eredmény: 2013-06-01T11:33:47CEST
System.out.println( dt.format( c.getTime() ) );

long f = DateUtils.getFragmentInDays(c, Calendar.MONTH);
```

```
// Eredmény: 1
System.out.println( f );

f = DateUtils.getFragmentInDays(c, Calendar.➥
    DAY_OF_MONTH);
// Eredmény: 0
System.out.println( f );

f = DateUtils.getFragmentInHours(c, Calendar.➥
    DAY_OF_MONTH);
// Eredmény: 11
System.out.println( f );

f = DateUtils.getFragmentInDays(c, Calendar.YEAR);
// Eredmény: 152
System.out.println( f );

// Eredmény: 21
System.out.println( f / 7 );

f = DateUtils.getFragmentInHours(c, Calendar.YEAR);
// Eredmény: 3659
System.out.println( f );
```

Látva a pillanatnyi időpontot, értelmezzük gyorsan az egyes eredményeket! Az 1 érték azért jött ki, mert az ekvivalens a *c.get(Calendar.DAY_OF_MONTH))* lekérdezéssel, itt pedig éppen elseje van. A 0 azt jelenti, hogy a hónapban még nem telt el egyetlen egész nap sem. A 11 arra utal, hogy 11:33 perckor már 11 óra eltelt a napból. A 153 az év eddig eltelt napjai, amit 7-tel osztva az eddig elmúlt heteket adja, ami 21. Ez igaz is, mert most éppen a 22. hétben járunk. Végül a 3659 az év eddig eltelt napjait jelenti. Röviden a többi metódust érdemes áttekinteni, de megjegyezzük, hogy ezeknek a *Date* típusra épülő változatai is rendelkezésre állnak:

```
public static long getFragmentInMinutes(Calendar ➥
    calendar, int fragment);
public static long getFragmentInSeconds(Calendar ➥
    calendar, int fragment);
public static long getFragmentInMilliseconds(Calendar ➥
    calendar, int fragment);
```

Logikai lekérdezések

Az *isSameDay()* visszaadja, hogy egy dátum ugyanarra a napra esik-e. Ez azt jelenti gyakorlatilag, hogy a dátum időrészétől eltekintve vizsgálja az egyezőséget.

```
public static boolean isSameDay(Calendar cal1, ➥
    Calendar cal2);
public static boolean isSameDay(Date date1, Date date2➥
    );
```

Sokszor arra vagyunk kíváncsiak, hogy 2 időpont teljesen egyezik-e, amire a *isSameInstant()* lekérdezés ad segítséget:

```
public static boolean isSameInstant(Calendar cal1, ➥
    Calendar cal2);
public static boolean isSameInstant(Date date1, Date ➥
    date2);
```

A használatára egy rövid példa:

```
Date pd = DateUtils.parseDateStrictly("2015-03-01 ➥
    08:29:02", pattern);
Date pd2 = DateUtils.parseDateStrictly("2015-03-01 ➥
    08:29:01", pattern);
// Eredmény: false, mert a secundum eltér
System.out.println( DateUtils.isSameInstant(pd, pd2));
```

Az *isSameLocalTime()* csak a *Calendar* objektum esetén értelmes, ugyanis az eltárolja ezt az információt is:

```
public static boolean isSameLocalTime(Calendar cal1, ➥
    Calendar cal2);
```

Date Calendar konverzió

Egy *Calendar* objektum *getTime()* metódusával kapunk egy *Date* objektumot. Ennek a fordítottja így lehetséges:

```
public static Calendar toCalendar(Date date);
```

A formázás támogatása

A *DateFormatUtils* class számos statikus *format()* metódus változattal támogatja a formázott dátum vagy *Calendar* kiírást. Példa:

```
TimeZone tz = TimeZone.getTimeZone("Europe/Budapest");
Calendar c = GregorianCalendar.getInstance( tz );
DateFormatUtils.format(c, "yyyy-MM-dd'T'HH:mm:ssz");
```

Stopper óra

A *StopWatch* class segítségével egy stopper órát tudunk használni.

Tömbműveletek - ArrayUtils

Elemek hozzáadása egy tömbhöz

Az *add()* és *addAll()* metódusok képesek elemet vagy elemeket betenni egy tömbbe, amit eredményül visszaadnak. Az *int* típusra például így néz ki a deklaráció:

```
public static int[] add(int[] array, int element);
public static int[] add(int[] array, int index, int ➥
    element);
public static int[] addAll(int[] array1, int... array2➥
    );
```

A használatra egy rövid példa, ami a tömb utolsó elemeként a 88-at is beszúrja:

```
int[] ai = new int[] { 12, 4, 5 };
ai = ArrayUtils.add(ai, 88);
ai = ArrayUtils.addAll(ai, 88, 33, 44);
```

Az index arra szolgál, hogy megadjuk a beszúrás pozícióját.

Elemek törlése

A *remove()* függvények képesek elemeket törölni egy tömbből. Ebből is annyi változat van, ahány alaptípus, nézzünk megint egy *int*-re:

```
ArrayUtils.remove(new int[]{2, 6, 3}, 1);
```

A példában az 1. pozíciójú elem kerül majd törlésre, azaz az eredmény tömb *[2, 3]* lesz. A *removeElement()* a 2. paraméterben megadott értéket törli, esetünkben a 6-ot.

```
int[] ai = ArrayUtils.removeElement(new int[]{2, 6, ➥
    3}, 6);
```

A *removeAll()* az összes tömbelemet törli.

Egy tömb klónozása

A *clone()* függvény lehetővé teszi, hogy egy tömbnek elkészítsük a másolatát. A paraméter egy valamilyen típusú tömb, az eredmény pedig ennek a klónja. Az alábbiakban a *float* esetét mutatjuk:

```
public static float[] clone(float[] array);
```

Konstrukciós műveletek

A *toMap()* feladata, hogy egy alkalmas tömbből Java *Map* objektumot adjon vissza:

```
public static Map<Object,Object> toMap(Object[] array)➥
    ;
```

Ezt teszi a következő szemléltető példa is:

```
Map colorMap = MapUtils.toMap(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}});
```

A *toArray()* egy típus biztos tömböt ad vissza, amit a paramétereiből állít össze:

```
public static <T> T[] toArray(T... items);
```

Erre is nézzünk 2 példát:

```
String[] array = ArrayUtils.toArray("1", "2");
String[] emptyArray = ArrayUtils.<String>toArray();
```

A *toObject()* szintén egy metódus család, az ismert összes alaptípusra működik. A feladata az, hogy egy skalár tömb alapján egy azzal ekvivalensnek tekinthető „becsomagoló osztályos” tömböt adjon vissza.

```
public static Double[] toObject(double[] array);
public static Integer[] toObject(int[] array);
...
```

Tartalmazás vizsgálat

A *contains()* család egy tömbben keres egy értéket, siker esetén *true* a visszaadott érték:

```
public static boolean contains(int[] array, int ➥
    valueToFind);
public static boolean contains(char[] array, char ➥
    valueToFind);
...
```

Keresés

Az *indexOf()* és *lastIndexOf()* metódusok egy értéket keresnek a tömbben, találat esetén visszaadják annak az indexét, ellenkező esetben *-1* a visszatérés. Minden típusra létezik egy változat, a példa kedvéért nézzük ismét az *int*-et:

```
public static int indexOf(int[] array, int valueToFind➥
    );
public static int indexOf(int[] array, int valueToFind➥
    , int startIndex);
public static int lastIndexOf(int[] array, int ➥
    valueToFind);
public static int lastIndexOf(int[] array, int ➥
    valueToFind, int startIndex);
```

A lebegőpontos esetek egy további lehetőséggel is rendelkeznek, megadhatunk egy deltát:

```
public static int indexOf(double[] array, double ➥
    valueToFind, double tolerance);
public static int indexOf(double[] array, double ➥
    valueToFind, int startIndex, double tolerance);
```

Logikai lekérdezések

Az *isEmpty()* és *isNotEmpty()* vizsgálja, hogy a tömb üres vagy nem üres. Ez minden alaptípusra működik természetesen:

```
public static boolean isEmpty(double[] array);
public static boolean isNotEmpty(int[] array);
public static <T> boolean isNotEmpty(T[] array);
...
```

Az *isEquals()* 2 – akár több dimenziós – tömböt hasonlít össze és egyenlőségük esetén igazat ad vissza:

```
public static boolean isEquals(Object array1, Object ➥
    array2);
```

Az *isSameLength()* 2 tömb paramétert kap és igaz a visszatérés, ha azok egyforma méretűek.

Résztömb visszaadása

A *subarray()* egy olyan tömböt ad vissza, ami az input tömb egy része:

```
public static long[] subarray(long[] array,
                              int startIndexInclusive,
                              int endIndexExclusive);
...
public static <T> T[] subarray(T[] array,
                               int startIndexInclusive➥
                                 ,
                               int endIndexExclusive);
```

Skalár tömb lekérése

Időnként jól jön, ha egy *Integer*, *Double*, ... tömb skalár megfelelőjéhez gyorsan hozzá tudunk jutni, ezt valósítják meg a *toPrimitive()* függvény különböző esetei:

```
public static double[] toPrimitive(Double[] array);
public static double[] toPrimitive(Double[] array, ➥
    double valueForNull);
...
```

Egyéb hasznos lehetőségek

Egy tömb elemeinek *String* reprezentációját adja a *toString()*, aminek használatára egy példa:

```
int[] ai = ArrayUtils.removeElement(new int[]{2, 6, ➥
    3}, 6);
System.out.println( ArrayUtils.toString(ai) );

// Eredmény: {2,3}
```

A *{}* jelek is részei a generált *String*nek. A másik hasznos metódus a *reverse()*, ami egy fordított sorrendbe állított tömböt ad vissza.

Osztályok - ClassUtils

A *ClassUtils* osztály egy olyan könyvtári rutingyűjtemény, ami kiegészíti a BeanUtils csomag lehetőségeit és könnyebben elérhetővé teszi a Java reflection (önelemzés) funkciókat. A következő főbb függvényei vannak:

- Nevek megszerzése.
- Logikai lekérdezések

Az alábbi néhány példa a nevek megszerzését mutatja be:

```
oStr = ClassUtils.getPackageCanonicalName( c );
oStr = ClassUtils.getPackageName( c );
oStr = ClassUtils.getShortCanonicalName( c );
```

Természetesen a *Class<T>* megszerzése is lehetséges:

```
Class c = ClassUtils.getClass("org.cs.beanutils.➥
    Gyumolcs");
```

A metódusok kezelését egy egyszerű példán keresztül érdemes áttekinteni, de előtte nézzük meg a szignatúráját:

```
public static Method getPublicMethod
      (Class<?> cls,
       String methodName,
       Class<?>... parameterTypes)
       throws SecurityException,
       NoSuchMethodException
```

A példa pedig a következő:

```
Set set = Collections.unmodifiableSet(...);
Method method = ClassUtils.getPublicMethod(set.➥
    getClass(), "isEmpty", new Class[0]);
Object result = method.invoke(set, new Object[]);
```

Amikor Java önelemzésre is szükség van, érdemes mindig rápillantani a *ClassUtils* osztály további lehetőségeire is.

Szövegekkel kapcsolatos műveletek

StrSubstitutor

Az *StrSubstitutor* class egy template, amibe név szerinti feloldás hivatkozási pontokat tehetünk. Nézzünk egy példát!

```
String resStr = StrSubstitutor
        .replaceSystemProperties("java = ${java.➥
            version} és OS = ${os.name}.");

// Eredmény: java = 1.6.0_26 és OS = Linux.
System.out.println(resStr);
```

A cserélendő név, érték párokat egy *Map* objektumba is tehetjük:

```
Map valuesMap = HashMap();
valuesMap.put("animal", "quick brown fox");
valuesMap.put("target", "lazy dog");
String templateString = "The ${animal} jumped over the➥
    ${target}.";
StrSubstitutor sub = new StrSubstitutor(valuesMap);
String resolvedString = sub.replace(templateString);

// Eredmény: The quick brown fox jumped over the lazy ➥
    dog.
```

Az osztály számos metódussal támogatja a Stringek minták alapján való megkonstruálását, így amikor ilyen feladataink vannak, érdemes megfontolni a használatát.

StrTokenizer

Amikor a Java beépített *StringTokenizer* osztályt használjuk, de valamilyen funkció nincs meg vagy túl nehézkes, akkor javasolt ennek az osztálynak az átnézése, lehetséges, hogy pont itt lesz a hiányolt funkció.

CharUtils

Az osztály a karakterekkel kapcsolatos legfontosabb műveletek és lekérdezések gyűjtőhelye. Nem soroljuk fel a összes lehetőséget, de néhány példán keresztül ízelítőt szeretnénk adni a lehetőségekről:

```
CharUtils.isAscii('3')  = true
CharUtils.isAscii('-')  = true
CharUtils.isAscii('\n') = true
CharUtils.isAsciiAlpha('a')  = true
CharUtils.isAsciiAlpha('3') = false
CharUtils.unicodeEscaped(' ') = \u0020
```

StringEscapeUtils

Ezzel az osztállyal egy Java, Java Script, HTML és XML menekülő karaktersorozatokra lehet egy Stringet alakítani.

```
public static final String escapeEcmaScript(String ➥
    input);
Input:  He didn't say, "Stop!"
Output: He didn\'t say, \"Stop!\"

public static final String escapeHtml4(String input);
Input: "bread" & "butter"
Output: &quot;bread&quot; &amp;&quot;butter&quot;

public static final String unescapeHtml4(String input)➥
    ;
Input: &quot;bread&quot; &amp;&quot;butter&quot;
Output: "bread" &"butter"

public static final String escapeXml(String input);
public static final String unescapeXml(String input);
```

Egyéb lehetőségek

BooleanUtils

A Boolean standard Java osztály könnyebb kezelését támogatja. Példa:

```
BooleanUtils.or(Boolean.TRUE, Boolean.TRUE) ➥
                    = Boolean.TRUE
BooleanUtils.or(Boolean.FALSE, Boolean.FALSE) ➥
                   = Boolean.FALSE
BooleanUtils.or(Boolean.TRUE, Boolean.FALSE) ➥
                   = Boolean.TRUE
BooleanUtils.or(Boolean.TRUE, Boolean.TRUE, Boolean.➥
    TRUE)    = Boolean.TRUE
BooleanUtils.or(Boolean.FALSE, Boolean.FALSE, Boolean.➥
    TRUE)  = Boolean.TRUE
BooleanUtils.or(Boolean.TRUE, Boolean.FALSE, Boolean.➥
    TRUE)   = Boolean.TRUE
BooleanUtils.or(Boolean.FALSE, Boolean.FALSE, Boolean.➥
    FALSE) = Boolean.FALSE
```

RandomStringUtils

Véletlen karaktersorozatok generálását teszi lehetővé. A *random()* metódusa több lehetséges módon használható.

Az Apache Commons Lang még tartalmaz számos, ritkábban használt osztályt, azoknak a felderítését az olvasóra bízzuk.

5. Joda.org - Programozás dátumokkal és időkkel

Az egyik leggyakoribb típus a dátum és az idő. Az idő kezelésének igénye szinte minden alkalmazásban felmerül, így ennek hatékony, sokrétegű és kényelmes használata elengedhetetlen. A Java ezen a téren nem ad kielégítő megoldást, a *Calendar* osztály is éppen csak megfelelő. A problémát felismerve született a *Joda Time* könyvtár (webhely: `http://joda-time.sourceforge.net/`). A Java 8 kiadásban tervezik a dátum és időkezelés funkciókat erőteljesebbé tenni, amihez majd ezt a könyvtárat is felhasználhatják, így érdemes már most megtanulni.

Miért használjuk a Joda-Time-ot?

A 2002 óta fejlesztett Joda-Time egy kiváló minőségű dátum és időtípus implementáció, amit a Java beépített *Calendar* vagy *Date* osztálya helyett érdemes használni, mert azoknál sokkal többet tud, rugalmasabb és könnyebben használható. A következő naptári rendszereket ismeri:

- Gregorián
- Júlián
- Buddhista
- Etióp
- Kopt keresztény
- Iszlám

A könyvtár képes a beépített Java típusokkal együttműködni, hiszen sok más API azokat igényli.

Alapfogalmak

Időpontok (Instants)

Az időpont a legalapvetőbb fogalom, a folyamatosan előrehaladó idő 1 pontja, amit milliszekundumban mérünk. Az origó az *1970-01-01T00:00Z*. Ez azt jelenti, hogy például az *1963.01.05* értéke a következő negatív szám: *-220579200000*. Fontos kiemelni, hogy ez az ezredmásodperc reprezentáció kompatibilis a Java dátum reprezentációjával, alapvetően ez biztosítja az oda-vissza integrációt is. A Joda-Time ezt a fogalmat a *ReadableInstant* interface-en keresztül biztosítja, aminek a következő implementációi léteznek:

- *Instant*: Egy egyszerű, csak olvasható implementáció, ami csak az UTC-t tudja. Akkor érdemes használni, ha időzóna és naptár független adatkezelésre van szükség.
- *DateTime*: Ez a leggyakrabban használt típus, mindent tud és ez is csak olvasható, azaz úgy kell kezelni, mint például a *String* típust.
- *DateMidnight*: A *DateTime* változata, ahol az idő komponens mindig nulla, azaz egy nap pontos kezdetét reprezentálja.
- *MutableDateTime*: A *DateTime* változtatható esete, olyan, mint a *String* esetén a *StringBuffer*.

Részleges dátumok (Partials)

A *ReadablePartial* interface jelenti ezt a fogalmat, aminek a jellegzetessége az, hogy valamilyen időkomponenst nem tartalmaz. Az elérhető implementációk a következőek:

- *LocalDate*: Az időzóna nincs eltárolva, az adatszerkezet elsősorban az év, hó és nap adatokra fókuszál, erre biztosít számos metódust.

- *LocalTime*: Az időzóna nincs eltárolva, az adatszerkezet elsősorban az idő adatokra fókuszál (a dátumra nem), erre biztosít számos metódust.
- *LocalDateTime*: Az időzóna nincs eltárolva, de a dátum és idő részek igen.
- *YearMonth*: Ez is csak olvasható class és egy év, hó párost reprezentál.
- *MonthDay*: Csak olvasható osztály és egy hónap, nap adategyüttest (például születésnap) tud eltárolni.
- *Partial*: Ez a class képes a dátum és idő adatrészek bármely kombinációját eltárolni.

A dokumentáció az alábbi összefüggést adja meg az *Instants* és *Partials* között:

```
Partial + Missing Fields + Time Zone = Instant
```

Intervallumok (Intervals)

Amikor egy *tól-ig* időpontot szeretnénk eltárolni, akkor *ReadableInterval* interface-t implementáló 2 class egyike lehet a legalkalmasabb:

- *Interval*: A dátum-idő intervallumok létrehozásához mindig meg kell adnunk a kezdő és záró dátumot, ezt sokféle módon biztosítja ez a csak olvasható osztály. Általában ezt használjuk.
- *MutableInterval*: Ezen class az *Interval* osztályhoz hasonló, de írható objektum, azaz „helyben” szerkeszthetőek az adatai.

Időtartamok (Durations)

A *ReadableDuration* interface-t megvalósító objektumok az ezredmásodpercben mért eltelt időt képes tárolni és kezelni. Az interface-t a *Duration* osztály implementálja. Az egyik legtermészetesebb művelete az, amikor egy időponthoz adjuk hozzá:

```
Instant + Duration = Instant
```

Periódusok (Periods)

A periódus valamely időközönkénti újabb és újabb időpont előállítását jelenti. Itt nem egyszerűen arról van szó, hogy egy fix értéket adok az előzőhöz, mert a periódusok ezredmásodpercben vett távolsága eltérő lehet. Itt gondoljunk csak arra, hogy vedd a következő hónapot. Ennek távolsága eltér január→február és február→március esetében. Az általános logikát a következő séma adja meg:

```
Instant + Period = Instant
```

Periódus persze az is lehet, hogy, hogy 4 év, 3 hónap és 2 nap. A periódust a *ReadablePeriod* interface-t implementáló osztályok valósítják meg, nézzük őket!

- *Years*: Valahány évente változó periódus.
- *Months*: Valahány havonta változó periódus.
- *Weeks*: Valahány hetente változó periódus.
- *Days*: Valahány naponta változó periódus.
- *Hours*: Valahány óránként változó periódus.
- *Minutes*: Valahány percenként változó periódus.
- *Seconds*: Valahány másodpercenként változó periódus.
- *Period*: Ez egy csak olvasható és rugalmas megvalósítása a periódusnak. Ellentétben az eddigieknek, itt megadhatjuk az összes dátum-idő komponenst, azaz olyan periódust is megalkothatunk, aminek év, hó, nap, óra, ... része is van.

- *MutablePeriod*: A működése hasonló a *Period* osztályhoz, de változtatható az értéke. Ez akkor hasznos, ha nem mindig ugyanakkora lépéssel akarunk haladni az időben, mert alkalmas pillanatban megváltoztathatjuk az értékét.

Naptár rendszerek (Chronology)

Már a Java is rendelkezik a *GregorianCalendar* osztállyal, de a Joda-Time *Chronology* class még ennél is több naptár rendszert képes kezelni. A használata így lehetséges:

```
DateTime dt = new DateTime(CopticChronology.➥
    getInstance());
```

Ebben az esetben a *DateTime* objektum a mostani időpont Kopt keresztény kalendárium szerinti időpontját fogja tartalmazni. A következő származtatott osztályok, azaz kronológiák vannak implementálva:

- *ISOChronology*
- *GJChronology*
- *GregorianChronology*
- *JulianChronology*
- *CopticChronology*
- *BuddhistChronology*
- *EthiopicChronology*

Időzónák (Time Zones)

A *Chronology* class támogatja az időzónák használatát. A jelenlegi *UTC* rendszer szerinti idő a világidő, ehhez képest vannak besorolva az egyes (fő)városok, amihez ilyen azonosító String-eket használunk:

```
America/Montreal
America/Nassau
America/New_York
...
Europe/Bratislava
Europe/Brussels
Europe/Budapest
Europe/Copenhagen
Europe/Gibraltar
...
```

A Joda-Time az időzónát a *DateTimeZone* osztály segítségével tárolja és kezeli. Így tudunk egy ilyen objektumot elkészíteni:

```
DateTimeZone zone = DateTimeZone.forID("Europe/London"➥
    );
```

Az UTC így adható meg:

```
DateTimeZone zoneUTC = DateTimeZone.UTC;
```

További példák:

```
DateTimeZone defaultZone = DateTimeZone.getDefault();
DateTimeZone.setDefault(myZone);
```

Példák a Joda-Time használatára

Az alapfogalmak megértése után a Joda-Time használatát úgy tanulhatjuk meg a leggyorsabban, ha a tipikus használatokra egy-egy kis példát mutatunk be. Nézzük őket!

A *DateTime* class

A *DateTime* egy időpont tárolására alkalmas osztály, aminek többféleképpen hozhatjuk létre az objektumait. Az alábbiakban bemutatjuk a leggyakoribb objektum készítési módszereket, mindegyiket egy rövid megjegyzéssel láttuk el.

```
package org.cs.jodatime;

import java.util.Calendar;
import java.util.Date;
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;

public class DateTimeTest
{
 public static void test1()
 {
  // Rendszeridő alapján
  DateTime dateTime = new DateTime();
  System.out.println("dateTime = " + dateTime);

  // Rendszeridő + időzóna
  DateTimeZone dtz = DateTimeZone.forID("Europe/➥
      Budapest");
  dateTime = dateTime.withZone(dtz);
  System.out.println("dateTime_1 = " + dateTime);

  // Direkt időpont megadással
  // év, hó, nap, óra, perc, másodperc, millisec
  dateTime = new DateTime(2013, 6, 8, 0, 0, 0, 0);
  System.out.println("dateTime_2 = " + dateTime);

  // java.util.Calendar alapján
  Calendar calendar = Calendar.getInstance();
  dateTime = new DateTime(calendar);
  System.out.println("dateTime_3 = " + dateTime);

  // java.util.Date millisec alapján
  Date date = new Date();
  dateTime = new DateTime(date.getTime());
  System.out.println("dateTime_4 = " + dateTime);

  // java.util.Date alapján
  dateTime = new DateTime(date);
  System.out.println("dateTime_5 = " + dateTime);
```

```
  // Alapértelmezett formátum alapján
  dateTime = new DateTime("2016-02-03T14➥
      :15:00.000+08:00");
  System.out.println("dateTime 6 = " + dateTime);
  dateTime = new DateTime("2016-02-03");
  System.out.println("dateTime 7 = " + dateTime);
 }

 public static void main(String[] args)
 {
  test1();
 }
}
```

A fenti programot lefuttatva ez az eredmény jelenik meg a képernyőn:

```
dateTime = 2013-06-08T09:07:34.915Z
dateTime 1 = 2013-06-08T11:07:34.915+02:00
dateTime 2 = 2013-06-08T00:00:00.000Z
dateTime 3 = 2013-06-08T09:07:35.033Z
dateTime 4 = 2013-06-08T09:07:35.128Z
dateTime 5 = 2013-06-08T09:07:35.128Z
dateTime 6 = 2016-02-03T06:15:00.000Z
dateTime 7 = 2016-02-03T00:00:00.000Z
```

Szeretnénk kiemelni a *dateTime 1* sort, ahol egy *DateTimeZone* class példányt használva olyan *DateTime* objektumot kértünk le, ami a budapesti időzóna szerint mutatja az időt. Láthatjuk, hogy ez +2 óra az UTC-hez képest, azaz a 9 óra helyett 11 órát mutat.

Dátumok növelése, csökkentése

Egy *DateTime* objektum minden olyan metódussal rendelkezik, ami az értékének értelmes megváltoztatását jelenti. Itt most nézzük meg a növelések és csökkentések lehetőségét!

```
public class DateTimeTest
{
...
  public static void test2()
  {
    DateTime dateTime = new DateTime();
    System.out.println("dateTime = " + dateTime);

    dateTime = dateTime.plusHours(3);
    System.out.println("dateTime = " + dateTime);

    dateTime = dateTime.plusWeeks(2);
    System.out.println("dateTime = " + dateTime);
  }
...
}
```

A példában a *dateTime* objektumhoz előbb 3 órát, majd 2 hetet adtunk, aminek a futási eredménye ez lett:

```
dateTime = 2013-06-08T11:34:39.259Z
dateTime = 2013-06-08T14:34:39.259Z
dateTime = 2013-06-22T14:34:39.259Z
```

Egy *DateTime* objektumbak minden komponense előjelesen növelhető, az éveket például a *plusYears()* metódussal változtathatjuk.

A következő hónap első napja

Használhatnánk a *DateTime* osztályt is, de most gyakorlásképpen a „helyben” változtatható, azaz írható *MutableDateTime* class segítségével nézzük meg, hogyan kérhetjük le például a következő hónap első napját.

```
public class DateTimeTest
{
...
  public static void firstDayOfNextMonth()
  {
    MutableDateTime dateTime = new MutableDateTime();
    System.out.println("dateTime = " + dateTime);

    dateTime.addMonths(1);
    dateTime.setDayOfMonth(1);
    dateTime.setMillisOfDay(0);

    System.out.println("dateTime = " + dateTime);
    System.out.println("A hét napja = " + dateTime.➥
        getDayOfWeek());
  }
...
}
```

A *dateTime* objektumba először a pillanatnyi dátumot kérjük le, majd hozzáadunk 1 hónapot, beállítjuk a napot 1-re. A szépség kedvéért az időt éjfélre állítottuk. Az eredmény a következő:

```
dateTime = 2013-06-08T19:39:00.531Z
dateTime = 2013-07-01T00:00:00.000Z
A hét napja = 1
```

Azt is láthatjuk, hogy ez a nap a hét első napja, azaz hétfő.

Napok száma 2 dátum között

Itt az idő, hogy kipróbáljuk a *DateMidnight* osztályt is, bár a többivel is el tudnánk végezni a feladatot. Ez viszont egyből készen adja az éjfélre állítást, azaz 2 napkezdést tudunk csinálni, amit a *start* és *end* változók hivatkoznak meg. Azt is látjuk, hogy a *Days* periódus fajtájú osztályt olyan célra is használhatjuk, hogy segítségével lekérjük a 2 instant közötti napok számát.

```
public class DateTimeTest
{
...
public static void daysBetween2Days()
  {
    DateMidnight start = new DateMidnight("1963-01-05"➥
        );
    DateMidnight end = new DateMidnight("2013-06-08");

    int days = Days.daysBetween(start, end).getDays();
```

```
    System.out.println("Days between " + start.➥
        toString("yyyy-MM-dd")
        + " and " + end.toString("yyyy-MM-dd") + " = "➥
            + days
        + " day(s)");
  }
...
}
```

A futási eredmény ez lett:

```
Days between 1963-01-05 and 2013-06-08 = 18417 day(s)
```

Formázott kiírás

A formázás az ismert Java szabályok szerint történik itt nincs semmi újdonság. A *DateTime* esetén például így tudunk formázni a *toString()* metóduson keresztül:

```
private static final String pattern = "E MM/dd/yyyy HH:mm:ss.SSS";

DateTime dateTime = new DateTime();
System.out.println(dateTime.toString(pattern));
System.out.println(dateTime.toString(pattern, Locale.GERMANY));
System.out.println(dateTime.toString(pattern, Locale.FRENCH));
System.out.println(dateTime.toString(pattern, Locale.JAPANESE));
```

Hónapok száma 2 dátum között

Gyakorlásképpen nézzük meg a 2 időpont között eltelt hónapok számát is, amihez persze most a *Months* class lesz használva.

```
public class DateTimeTest
{
...
public static void monthsBetween2Days()
  {
    DateMidnight start = new DateMidnight("1963-01-05");
    DateMidnight end = new DateMidnight(new Date());

    int months = Months.monthsBetween(start, end).getMonths();

    System.out.println("Months between " + start.toString("yyyy-MM-dd")
        + " and " + end.toString("yyyy-MM-dd") + " = " + months);
  }
...
}
```

Az eredmény:

```
Months between 1963-01-05 and 2013-06-08 = 605
```

A dátum egyes részeinek megszerzése

A most következő példában azt fogjuk bemutatni, hogy egy *DateTime* objektum részeit milyen módon kérhetjük le. Talán nem is kell kiemelnünk, de azért megjegyezzük, hogy természetesen a többi instant class (*DateMidnight*, ...) esetén is pont ugyanígy tennénk. Nézzük a példát!

```
public class DateTimeTest
{
...
  public static void partOfDateTime()
  {
    DateTime dateTime = new DateTime();
    System.out.println("dateTime = " + dateTime);
```

```
    // Hányadik napon vagyunk
    System.out.println("DOY = " + dateTime.getDayOfYear());
    System.out.println("DOM = " + dateTime.getDayOfMonth());
    System.out.println("DOW = " + dateTime.getDayOfWeek());

    // Az év hetének száma
    System.out.println("WOW = " + dateTime.getWeekOfWeekyear());

    // óra, perc, másodperc a napon belül
    System.out.println("HOD = " + dateTime.getHourOfDay());
    System.out.println("MOH = " + dateTime.getMinuteOfHour());
    System.out.println("SOM = " + dateTime.getSecondOfMinute());

    // óra, perc, másodperc az éven belül
    System.out.println("MOD = " + dateTime.getMinuteOfDay());
    System.out.println("SOD = " + dateTime.getSecondOfDay());
  }
...
}
```

A példában bemutatott mezőkön (Date/Time fields) kívül még számos másik is létezik.

Az ISO dátumformátumok

Az *ISODateTimeFormat* class sok alapértelmezett formázót tartalmaz, érdemes megtanulni a használatukat. Vegyünk néhány esetet át!

```
// Legyen egy dátumunk
DateTime dateTime = new DateTime();

// yyyyMMdd: 20120228
dateTime.toString(ISODateTimeFormat.basicDate());

// 20120228T163810.037+0800
dateTime.toString(ISODateTimeFormat.basicDateTime());

// 20120228T163810+0800
dateTime.toString(ISODateTimeFormat.basicDateTimeNoMillis());

// yyyyDDD: 2012059
dateTime.toString(ISODateTimeFormat.basicOrdinalDate());

// 2012W092
dateTime.toString(ISODateTimeFormat.basicWeekDate());

// 2012W092T163810.037+0800
dateTime.toString(ISODateTimeFormat.basicWeekDateTime());
```

A *basicWeekDate()* és *basicWeekDateTime()* eredményét úgy kell értelmezni, hogy ekkor nem a megszokott év, hó, nap koordinátában gondolkodunk, hanem azt mondjuk meg, hogy melyik év, hányadik hetének, hányadik napja. A *2012W092* ezek szerint azt jelenti, hogy 2012 év, 9. hetének 2. napja, azaz kedd (mindig 1=hétfő).

Integráció a Java SDK-val

Már említettük, hogy ez fontos, mert más API-k a Java beépített idő és dátum kezelését használják, így a Joda-Time használata esetén lesznek olyan helyzetek, amikor *Calendar* vagy *Date* objektumokkal kell kommunikálni:

```
DateTime dateTime = new DateTime();
Calendar calendar = dateTime.toCalendar(Locale.➥
    getDefault());
Date date = dateTime.toDate();
```

A Java SDK→*DateTime* (és egyéb instant) irányt már korábban bemutattuk.

Intervallumok használata

Elsőként nézzük meg, hogy egy intervallumot miként tudunk létrehozni egy *start* és *end* dátumból! A *testInterval()* metódus 2 dátumból legyárt egy *interval* nevű példányt, majd kiírja annak annak különféle információit. A következő lépésben az intervallum végét 1 hónappal meghosszabbítjuk, így 6. hó helyett 7. lesz. Befejezésül lekérünk egy *Duration* példányt, ami az intervallum elejének és végének a különbségéből adódik. Az utolsó sorban a *duration*-t napokban írjuk ki, de számos más mértékegységben is megtehettük volna.

```
public class DateTimeTest
{
...
public static void testInterval()
  {
    DateMidnight start = new DateMidnight("1963-01-05");
    DateMidnight end = new DateMidnight("2013-06-08");

    Interval interval = new Interval(start, end);

    // Interval = 1963-01-05T00:00:00.000Z/2013-06-08T00:00:00.000Z
    System.out.println("Interval = " + interval);

    // Start    = 1963-01-05T00:00:00.000Z
    System.out.println("Start    = " + interval.getStart());

    // End      = 2013-06-08T00:00:00.000Z
    System.out.println("End      = " + interval.getEnd());

    interval = interval.withEnd(interval.getEnd().plusMonths(1));
    // Interval = 1963-01-05T00:00:00.000Z/2013-07-08T00:00:00.000Z
    System.out.println("Interval = " + interval);

    Duration duration = interval.toDuration();
    // Duration = 18447
    System.out.println("Duration = " + duration);
  }
...
}
```

Az Instant class

Érdemes a ritkábban használt Instant class-ra is rápillantani. A folyamatosan telő idő zérus pontját már említettük, ez a *1970-01-01T00:00Z*. A következő *instant* objektum egy ennél 1 másodperccel (1000 ezredmásodperccel) későbbi időpontot tárol:

```
Instant instant = new Instant(1000);
```

Ehhez adjunk még fél másodpercet, majd vonjunk ki negyedet:

```
instant = instant.plus(500);
instant = instant.minus(250);
```

Időzónák használata

Az üzleti alkalmazásokban a lokális idő kezelése nagyon fontos, hiszen amikor Amerikában alszanak, akkor Európában dolgoznak. Az egyezményes koordinált világidő vagy röviden koordinált világidő (angolul rövidítéssel: *UTC*) az a hivatkozási időzóna, amelyhez a Föld többi időzónáját viszonyítjuk. Az *UTC* a greenwichi középidőt (*GMT*) váltotta 1961-ben, de máig mindkét jelölést használják, noha a két fogalom nem azonos. Az *UTC* használata ajánlott, a *GMT* mint fogalom elavultnak számít. Az egyezményes ko-

ordinált világidő a nagy pontossággal, a világ 50 különböző laborjában egyenletesen mért nemzetközi atomidőből (International Atomic Time, TAI) származik. A Föld időzónáit az UTC-hez viszonyítva állapítják meg és mivel a greenwichi középidőt váltotta, az az időzóna maradt a viszonyítási pont. Az attól keletre eső időzónák pozitív, míg a nyugatra találhatók negatív értékű órával térnek el (vannak nem egész órával eltérő időzónák is). A Joda-Time mindezt remekül kezeli, sőt ismeri az egyes országokra jellemző óra előre és hátraállítást is. Szokásunkhoz híven mindezt az alábbi *testTimeZone()* metóduson keresztül szeretnénk elmagyarázni.

```
1  public class DateTimeTest
2  {
3  ...
4    public static void testTimeZone()
5    {
6      DateTimeZone tzKiev = DateTimeZone.forID("Europe/Kiev");
7      DateTimeZone tzBp = DateTimeZone.forID("Europe/Budapest");
8
9      DateTime utcTime = new DateTime(2013, 5, 9, 21, 9, DateTimeZone.UTC);
10     // UTC:2013-05-09T21:09:00.000Z
11     System.out.println("UTC:"+utcTime);
12
13     DateTime localTime = utcTime.withZone( tzKiev );
14     // Kiev:2013-05-10T00:09:00.000+03:00
15     System.out.println("Kiev:" + localTime);
16
17     localTime = utcTime.withZone( tzBp );
18     // Budapest:2013-05-09T23:09:00.000+02:00
19     System.out.println("Budapest:" + localTime);
20
21     localTime = new DateTime(2012, 11, 29, 13, 40, tzBp);
22     // Budapest (november):2012-11-29T13:40:00.000+01:00
23     System.out.println("Budapest (november):" + localTime);
24     utcTime = localTime.withZone(DateTimeZone.UTC);
25     // UTC (november):2012-11-29T12:40:00.000Z
26     System.out.println("UTC (november):"+utcTime);
27
28     localTime = new DateTime(2012, 5, 11, 13, 40, tzBp);
29     // Budapest (május):2012-05-11T13:40:00.000+02:00
30     System.out.println("Budapest (május):" + localTime);
31     utcTime = localTime.withZone(DateTimeZone.UTC);
32     // UTC (május):2012-05-11T11:40:00.000Z
33     System.out.println("UTC (május):"+utcTime);
34   }
35 ...
36 }
```

A 6. és 7. sorban egy budapesti és egy kievi időzónát reprezentáló objektumot készítettünk. A paraméterül átadott szöveg szabvány, ezt a Java SDK is tartalmazza. A 9. sorban egy 2013.05.09 napi 21:09 időpontot rögzítettünk az *utcTime* változóhoz, UTC, azaz világidővel. A 11. sor megjegyzésében láthatjuk is, hogy a 12. sor kiíró utasítása mit jelenít meg erről. A 13. sor nagyon tanulságos, mert láthatjuk belőle, hogy egy UTC időpontot miként alakíthatunk a kievi időre. A 17. sorban pedig a budapestire. Láthatjuk a megjelenített értékekből (14. és 18. sorok), hogy ezen a napon Kiev már 2013.05.10-i napra tért át (0 óra 9 perc), míg Budapest még aznap van, de 2 órával előbbre (23 óra 09 perc). A 21-33 sorok kódjában 2 érdekesség is rejtőzik, de mindkettő nagyon lényeges. Láthatjuk, hogy a helyi időponthoz milyen egyszerűen tud-

juk megszerezni az UTC időt. A másik érdekesség, hogy egyes országok (Magyarország is) az időzónán belül téli és nyári időszámítással is rendelkezik. Ez az oka annak, hogy októberben (2012.11.29) a helyi és UTC idő között csak 1 óra különbséget látunk, míg májusban (2012.05.11) 2 órát.

A *java.sql.Date* integráció

A Java JDBC dátum típus esetén az SQL adatbázisokkal a *java.sql.Date* típuson keresztül tartja a kapcsolatot. Szerencsére ezt is könnyen használni tudjuk, mert a lekért ezredmásodperc segítségével például így tudunk ilyen típust készíteni:

```
java.sql.Date startDate = new java.sql.Date(
  new DateMidnight(2012, 11, 5, DateTimeZone.UTC).➥
      getMillis());

java.sql.Date endDate = new java.sql.Date(
  new DateMidnight(2012, 11, 8, DateTimeZone.UTC).➥
      getMillis());
```

Sokat segít az is, ha látjuk a kétféle Java dátum típus közötti oda-vissza konvertálás módját is:

```
// A mostani pillanat
java.util.Date now = new java.util.Date();

// Alakítsuk át SQL Date-re:
java.sql.Date sqlDate = new java.sql.Date(now.getTime➥
    ());

// Alakítsuk vissza Java Date-re:
java.util.Date utilDate = new java.util.Date(sqlDate.➥
    getTime());
```

Deklarált nap meghatározása

Ismerős mindenkinek az a nyelvjárás, hogy legyen a következő újévkor, vagy jövő év első csütörtökön. Ilyenkor úgy adunk meg egy időpontot, hogy nem a pontos számszerű jellemzőit specifikáljuk, hanem csak körbeírjuk. Nézzük meg először ennek a 2 példának a Joda-Time-ra épülő megfogalmazását! Lekérjük a mostani pillanatnyi időt a világidő szerint. A *dayOfYear()* metódus egy *DateTime.Property* osztálybeli objektummal tér vissza, aminek a *withMaximumValue()* metódusa olyan *DateTime* objektumot ad vissza, ami ezen mező (azaz property) maximumát veszi fel. Esetünkben ez a field most a nap természetesen. Ezzel elérkeztünk a szilveszter időpontjához, amihez 1 napot adva az újévet kaptuk vissza.

```
DateTime newYear = new DateTime(DateTimeZone.UTC).➥
    dayOfYear().withMaximumValue().plusDays(1);
```

Nézzük az új év első szerda napját:

```
DateTime firstWednesday = newYear.plusDays(➥
    DateTimeConstants.THURSDAY - newYear.getDayOfWeek➥
    ());
```

A kód lekéri a kivonás jobb oldalán azt, hogy a *newYear* időponthoz (azaz az újév napja) azt az információt, hogy ez most a hét hányadik napja (a mi konkrét esetünkben most 3, azaz szerda lesz). A *THURSDAY* konstans értéke 4, ezért a *plusDays(1)* lesz meghívva, azaz 2014. év első csütörtök 2014.01.02 napon lesz.

A következő példánk legyen az, hogy meghatározzuk a következő péntek napját, ahol a kiindulás napját paraméterül kapjuk (ez lesz a *d*). A *dayOfWeek()* egy *LocalDate.Property* objektumot adja vissza, aminek a kérdéses mezőjét (azaz most a napot) 5-re, azaz péntekre állítjuk. Amennyiben az input *d* nap péntek előtti, úgy ezen a héten, különben jövő héten (*plusWeeks(1)*) lesz a következő péntek.

```
public class DateTimeTest
{
...
  public static LocalDate calcNextFriday(LocalDate d)
  {
    LocalDate friday = d.dayOfWeek().setCopy(5);
    if (d.isBefore(friday))
    {
      return d.dayOfWeek().setCopy(5);
    } else
    {
      return d.plusWeeks(1).dayOfWeek().setCopy(5);
    }
  }
...
}
```

Felhívjuk a figyelmet az *isBefore()* metódusra, ami akkor ad vissza igaza, ha a paraméterül kapott időpont előtt vagyok. Ehhez hasonló segítő metódus nagyon sok van, itt nem is érdemes tételesen felsorolni őket.

Befejezésül egy klasszikus példát mutatunk be. A kérdés úgy szól, hogy egy megadott évben mikor lesz húsvét vasárnapja. Itt ragad-

juk meg az alkalmat, hogy megemlítsük a *Jollyday* könyvtárat (webhely: `http://jollyday.sourceforge.net/`). Ez arra lett tervezve, hogy együttműködjön a Joda-Time-mal és azt felkészítse olyan további tudással, ami az ünnepnapok kezelését profi szinten ismeri. Országfüggően tudja az egyes ünnepet, most azonban csak a húsvét napjának lekérdezésére szeretnénk használni, ahogy a következő példa bemutatja:

```
public class DateTimeTest
{
...
public static void testHusvet(int ev)
  {
    LocalDate lc = null;
    de.jollyday.util.CalendarUtil cu = new de.jollyday➥
        .util.CalendarUtil();
    lc = cu.getEasterSunday(ev);
    // Vasárnap: 2013-03-31
    System.out.println(lc);
  }
...
}
```

A *CalendarUtil cu* példányára meghívott *getEasterSunday()* metódus kap egy évet és visszaadja azt a Joda-Time *LocalDate* objektumot, ami a húsvét vasárnapot reprezentálja.

Részleges dátum és időpontok

A helyi nap és időpont megadása az ismert módon lehetséges, ahogy a *test3()* első 2 sora is mutatja. A következő sor a *partialDate* objektumot kiegészíti a *partialTime* résszel.

```
public class DateTimeTest
{
...
  public static void test3()
  {
    LocalDate partialDate = new LocalDate(2012, 12, 3)➥
        ;
    LocalTime partialTime = new LocalTime(12, 50);

    DateTime dateTime = partialDate.toDateTime(➥
        partialTime,
            DateTimeZone.UTC);

    // 12:50:00.000
    System.out.println( partialTime );
    // 2012-12-03T12:50:00.000Z
    System.out.println( dateTime );
  }
...
}
```

Amikor csak az év, hónapot akarjuk reprezentálni, akkor erre a *YearMonth* class kiválóan alkalmas:

```
YearMonth expirationDate = new YearMonth(2014, 12);
```

A használatára pedig ez egy releváns példa:

```
String expirationDay = expirationDate.toDateTime(new ➥
    DateTime(DateTimeZone.UTC).dayOfMonth().➥
    withMinimumValue()).dayOfWeek().getAsText();

System.out.println(expirationDay);
```

Mit csinál ez a kis kód? A *expirationDate* változó *toDateTime()* metódusa visszaadja a december 1-et, amit szövegesen megjelenítünk: *Monday*. Persze ezt a dátumot könnyebben is kitalálhattuk volna, hiszen minden hónap elsejével kezdődik. Most adjunk meg egy születésnapot reprezentáló hónap, nap objektumot (június 25.):

```
MonthDay birthday = new MonthDay(6, 25);
```

Vajon milyen napra esik a következő évben? A választ a *dayOfWeek* szöveg adja meg, amit az eddigiek alapján remélhetőleg már mindenki megért.

```
String dayOfWeek = birthday.toDateTime(new DateTime(➥
    DateTimeZone.UTC).plusYears(1)).dayOfWeek().➥
    getAsText();
```

Az idő intervallumokról bővebben

Amikor egy napi feljegyzések vagy meeting szervező alkalmazást készítünk nagy segítségre jöhet az intervallumok használatának az a módja, ami óra, perc-re van kihegyezve. Az alábbiakban a mai napra felvettünk néhány időintervallumot:

```
DateTime today = new DateTime(DateTimeZone.UTC);

Interval firstMeeting = new Interval(
        new LocalTime(9, 0).toDateTime(today),
        new LocalTime(10, 0).toDateTime(today));

Interval coffeeBreak = new Interval(
        new LocalTime(10, 0).toDateTime(today),
        new LocalTime(10, 30).toDateTime(today));

Interval secondMeeting = new Interval(
        new LocalTime(10, 30).toDateTime(today),
        new LocalTime(12, 0).toDateTime(today));

Interval lunchTime = new Interval(
        new LocalTime(13, 0).toDateTime(today),
        new LocalTime(14, 0).toDateTime(today));

Interval thirdMeeting = new Interval(
        new LocalTime(16, 0).toDateTime(today),
        new LocalTime(18, 0).toDateTime(today));

Interval sportEvent = new Interval(
        new LocalTime(17, 0).toDateTime(today),
        new LocalTime(19, 0).toDateTime(today));
```

Az első művelet, amit bemutatunk az 2 intervallumunk közötti szünet (GAP), ami szintén egy intervallum. Esetünkben a 3. meeting és az

ebéd közötti időt számítottuk ki és tettük bele a *gap* változóba:

```
Interval gap = thirdMeeting.gap(lunchTime);
```

Érdekes az is, hogy mekkora az átfedés, amit az *overlap()* metódus számít ki nekünk:

```
Interval overlap = thirdMeeting.overlap(sportEvent);
```

Ez is intervallum, esetünkben 17-18 óráig.

Kifinomult formázott kiírás

A *DateTimeFormatter* class biztosítja a rugalmas, szöveges dátum előállítást. Néhány példa a formázó objektumok előállítására:

```
DateTimeFormatter fmt = DateTimeFormat.forPattern("➥
    yyyy/MMM/dd");
DateTimeFormatter frenchFmt = fmt.withLocale(Locale.➥
    FRENCH);
DateTimeFormatter germanFmt = fmt.withLocale(Locale.➥
    GERMAN);
```

Egy dátumhoz így használjuk ezt a formázót:

```
DateTime dateTime = new DateTime(2012, 12, 1, 12, 15, ➥
    DateTimeZone.UTC);
System.out.println(germanFmt.print(dateTime));
```

Hatékonyan állíthatunk elő formázó objektumokat a *DateTimeFormatterBuilder* osztály bevetésével:

```
DateTimeFormatter formatter = new ➥
    DateTimeFormatterBuilder()
        .appendDayOfWeekShortText()
        .appendLiteral(",␣")
        .appendDayOfMonth(2)
        .appendLiteral('-')
        .appendMonthOfYearShortText()
        .appendLiteral('-')
        .appendYear(4, 4)
        .appendLiteral(",␣")
        .appendEraText()
        .toFormatter();
```

Az időtartamok összehasonlítása

Egy *Interval* példánynak van időtartama, ezért eszerint összehasonlíthatjuk az intervallumokat. Az alábbiakban *i1* és *i2* egy-egy intervallum:

```
DateTime today = new DateTime(DateTimeZone.UTC);

Interval i1 = new Interval(
        new LocalTime(9, 0).toDateTime(today),
        new LocalTime(11, 0).toDateTime(today)));

Interval i2 = new Interval(
        new LocalTime(20, 0).toDateTime(today),
        new LocalTime(23, 0).toDateTime(today)));
```

Ezek hosszának összehasonlítása:

```
int i = o1.toDuration().compareTo(o2.toDuration());
```

A periódusok használata

Az alábbi példa 2012.02.01 időpontra havi periódust használva 1 egységgel (azaz 1 hónappal) növeli meg a dátumot:

```
public class DateTimeTest
{
...
  public static void testPeriodus()
  {
    DateMidnight firstOfFebruary = new DateMidnight➥
        (2012, 2, 1, DateTimeZone.UTC);
    DateTimeFormatter formatter = ISODateTimeFormat.➥
        yearMonthDay();

    // Eredmény: 2012-03-01
    System.out.println(formatter.print(firstOfFebruary➥
        .plus(Months.ONE)));
  }
...
}
```

A következő példa egy nagyon általános periódus létrehozását mutatja be:

```
DateTimeFormatter formatter = ISODateTimeFormat.➥
    yearMonthDay();

DateMidnight start = new DateMidnight(2013, 1, 1, ➥
    DateTimeZone.UTC);
DateMidnight end = new DateMidnight(2014, 8, 1, ➥
    DateTimeZone.UTC);

Period period = new Period(start, end);

DateMidnight startMinusPeriod = start.minus(period);
DateMidnight endPlusPeriod = end.plus(period);

System.out.printf("before␣start:␣%s\n", formatter
        .print(startMinusPeriod));
System.out.printf("period␣start:␣%s\n", formatter.➥
    print(start));
System.out.printf("period␣end:␣␣␣%s\n", formatter.➥
    print(end));
System.out.printf("after␣␣end:␣␣␣%s\n", formatter
        .print(endPlusPeriod));
```

Az eredmény:

```
before start: 2011-06-01
period start: 2013-01-01
period end:   2014-08-01
after  end:   2016-03-01
```

A periódus hossza: 1 év és 7 hónap. Ennek megfelelően az előző periódus 2011.06.01-vel kezdődik, a következő pedig 2016.03.01-én fejeződik be.

6. Apache Commons - Virtual File System

Az Apache Virtual File System jelentőségét az adja, hogy a különféle fájlrendszereket elfedve biztosítja azok egységes kezelését. Eközben megvalósítja a legfontosabb shell műveleteket is (másolás, mozgatás, könyvtár listázás, törlés). Bevezet egy eseménykezelő alrendszert is, amivel hatékony fájl alapú integrációt valósíthatunk meg. A project webhelye: `http://commons.apache.org/proper/commons-vfs/`.

A *VFS* a következő fájlrendszereket támogatja:

- Helyi fájlok és mappák (*file://*)
- Tömörített fájlok: zip (*zip://*), jar (*jar://*), tar, tgz, bz2, gzip, bzip2
- Windows share (*smb://*)
- FTP (*ftp://*)
- HTTP, HTTPS (*http://*)
- SSH vagy SCP (*sftp://*)
- Webdav (*webdav://*)
- Erőforrás (a class loader használatával) fájlok (*res://*)
- Átmeneti fájlok (*tmp://*)

Fájlműveletek

Egy fájlrendszernek természetesen a fájlok a legfontosabb objektumai, amiket a VFS a *FileObject* osztállyal reprezentál. A dokumentáció tartalmaz egy *Shell.java* forrásprogramot ennek a bemutatására, amit most mi is felhasználunk az elemi műveletek bemutatásához. A *Shell* osztály kódja 3 *private* adattagot tartalmaz, amit a példánkban mi is használunk:

- *private final FileSystemManager mgr;* → a VFS fájlrendszer menedzser
- *private FileObject cwd;* → Az aktuális munkakönyvtár fájlobjektum (current working directory)
- *private BufferedReader reader;* → Egy reader textfájl kezeléshez

Ezek inicializálása így történhet:

```
private Shell() throws FileSystemException
{
    mgr = VFS.getManager();
    cwd = mgr.resolveFile(System.getProperty("user.dir"➥
        ));
    reader = new BufferedReader(new InputStreamReader(➥
        System.in));
}
```

A cd parancs

A change directory megvalósítását a következő kis részlet mutatja. A *cd()* metódus *cmd* tömbje tartalmazza a teljes parancsot. A *cmd[0]* mindig a parancs kulcsszava, utána jönnek a paraméterek. A *cd* esetén csak 1 paraméternek van értelme, ami megadja, hogy mi legyen az éppen aktuális, új munkakönyvtár útvonala. Amennyiben ez nincs megadva, akkor a lenti kód a user home directory-t állítja be a *cwd* értékeként. A manager *resolveFile()* metódusa képes a *FileObject* objektumot visszaadni, ami a beállítandó új könyvtár típusú fájlra mutat. Amennyiben az létezik, úgy erre állítjuk a *cwd*-t is.

```
public class Shell
{
    private final FileSystemManager mgr;
    private FileObject cwd;
    private BufferedReader reader;
...
    public void cd(final String[] cmd) throws ➥
        Exception
    {
        final String path;
        if (cmd.length > 1)
        {
```

```
            path = cmd[1];
        }
        else
        {
            path = System.getProperty("user.home");
        }

        // Locate and validate the folder
        FileObject tmp = mgr.resolveFile(cwd, path);
        if (tmp.exists())
        {
            cwd = tmp;
        }
        else
        {
            System.out.println("Folder does not exist:"
                + tmp.getName());
        }
        System.out.println("Current folder is " + cwd.
            getName());
    }
...
}
```

A pwd parancs

Az aktuális könyvtár kiírás, illetve lekérdezése az előző példa után már nagyon egyszerű, de azért nézzük!

```
public class Shell
{
...
  public void pwd()
    {
        System.out.println("Current folder is " + cwd.
            getName());
    }
...
}
```

A cat parancs

A fájl tartalmát írja ki a képernyőre. A példabeli *file* változó egy olyan *FileObject*-re mutat, aminek a könyvtár részét a *cwd*, a fájlnév részét pedig a *cmd[1]* String adja. A *FileUtil* a VFS könyvtár egy beépített utility osztálya.

```
public class Shell
{
...
   public void cat(final String[] cmd) throws
       Exception
    {
        if (cmd.length < 2)
        {
            throw new Exception("USAGE: cat <path>");
        }

        // Locate the file
    // Példa: cwd == mgr.resolveFile("file://home/
        inyiri");
        final FileObject file = mgr.resolveFile(cwd,
            cmd[1]);

        // Dump the contents to System.out
        FileUtil.writeContent(file, System.out);
        System.out.println();
    }
...
}
```

Az ls parancs

A könyvtári tartalom listázását azért is érdemes megérteni, mert saját programjainkban is hasonlóan szerezhetjük majd be a könyvtári bejegyzéseket. A lenti implementáció a *-R* kapcsoló esetében (6-16 sorok) rekurzívan is be tudja járni az alkönyvtárakat, amely üzemmódot a *recursive* logikai változó tartalmazza. A 18-26 sorok között megszerezzük azt a *file* nevű *FileObject* objektumot, ami a listázás gyökérpontja lesz. A 34-44 sorok között egyszerűen kiírjuk a fájl nevét, ha az nem *FileType.FOLDER* típusú, azaz maga már nem tárol más fájlokra vonatkozó listát. Mindebben egy másik VFS utility osztály, a *FileContent* segít. Amennyiben az éppen feldolgozás alatt lévő *FileObject* objektum egy folder, úgy meghívunk rá egy *listChildren()* nevű (a kódja a 49-73 sorok között van) metódust, ami akár rekurzívan is képes bejárni a directory részfát.

```
1  public class Shell
2  {
3  ...
4    public void ls(final String[] cmd) throws FileSystemException
5      {
6          int pos = 1;
7          final boolean recursive;
8          if (cmd.length > pos && cmd[pos].equals("-R"))
9          {
10             recursive = true;
11             pos++;
12         }
13         else
14         {
```

```
15              recursive = false;
16          }
17
18          final FileObject file;
19          if (cmd.length > pos)
20          {
21              file = mgr.resolveFile(cwd, cmd[pos]);
22          }
23          else
24          {
25              file = cwd;
26          }
27
28          if (file.getType() == FileType.FOLDER)
29          {
30              // List the contents
31              System.out.println("Contents of " + file.getName());
32              listChildren(file, recursive, "");
33          }
34          else
35          {
36              // Stat the file
37              System.out.println(file.getName());
38              final FileContent content = file.getContent();
39              System.out.println("Size: " + content.getSize() + " bytes.");
40              final DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, ➥
                    DateFormat.MEDIUM);
41              final String lastMod = dateFormat.format(new Date(content.getLastModifiedTime()));
42              System.out.println("Last modified: " + lastMod);
43          }
44      }
45
46      /**
47       * Lists the children of a folder.
48      */
49      private void listChildren(final FileObject dir,
50                                final boolean recursive,
51                                final String prefix)
52          throws FileSystemException
53      {
54          final FileObject[] children = dir.getChildren();
55          for (int i = 0; i < children.length; i++)
56          {
57              final FileObject child = children[i];
58              System.out.print(prefix);
59              System.out.print(child.getName().getBaseName());
60              if (child.getType() == FileType.FOLDER)
61              {
62                  System.out.println("/");
63                  if (recursive)
64                  {
65                      listChildren(child, recursive, prefix + "    ");
66                  }
67              }
68              else
69              {
70                  System.out.println();
71              }
72          }
73      }
74  ...
75  }
```

A touch parancs

Az ismert működés szerint ennek a parancsnak csak az a hatása, hogy frissíti a fájl utolsó módosításának dátumát. A működése a következő:

```
public class Shell
{
...
    public void touch(final String[] cmd) throws ➥
        Exception
    {
        if (cmd.length < 2)
        {
            throw new Exception("USAGE: touch <path>")➥
                ;
        }
        final FileObject file = mgr.resolveFile(cwd, ➥
            cmd[1]);
        if (!file.exists())
        {
            file.createFile();
        }
        file.getContent().setLastModifiedTime(System.➥
            currentTimeMillis());
    }
...
}
```

A cp parancs

A másolás az egyik legfontosabb művelet, így nézzük ennek is a megvalósítását! Az *src* változó a honnan kérdésre tárolja a választ. Az aktuális könyvtárhoz képest tárolja el a másolandó fájl nevét, ami a szokások szerint az első paraméter, azaz a *cmd[1]* tartalma. A *cmd[2]* a cél nevét reprezentálja. A *copyFrom()* metódus logikája az, hogy ezt mindig a célfájl hívja meg magára, megadva pataméterként a forrást.

```
public class Shell
{
...
    public void cp(final String[] cmd) throws ➥
        Exception
    {
        if (cmd.length < 3)
        {
            throw new Exception("USAGE: cp <src> <dest➥
                >");
        }

        final FileObject src = mgr.resolveFile(cwd, ➥
            cmd[1]);
        FileObject dest = mgr.resolveFile(cwd, cmd[2])➥
            ;
        if (dest.exists() && dest.getType() == ➥
            FileType.FOLDER)
        {
            dest = dest.resolveFile(src.getName().➥
                getBaseName());
        }

        dest.copyFrom(src, Selectors.SELECT_ALL);
    }
...
}
```

A *Selectors* osztály a működést befolyásolja, ezért érdemes röviden megnézni a lehetséges értékeit, mert más helyeken is használható:

- *SELECT_SELF*: Csak a megadott alap (base) fájlt vagy foldert szelektálja.
- *SELECT_SELF_AND_CHILDREN*: Hasonló az előzőhöz, de a közvetlen gyerekeket is szelektálja.
- *SELECT_CHILDREN*: Csak a közvetlen gyerekeket szelektálja.
- *EXCLUDE_SELF*: Az összes gyereket szelektálja, kivéve magát a base foldert
- *SELECT_FILES*: Csak a „rendes” fájlokat szelektálja.
- *SELECT_FOLDERS*: Csak a foldereket szelektálja.
- *SELECT_ALL*: A base fájlt (foldert) és az összes alkönyvtárát és fájlt is szelektál.

Az rm parancs

Ez a parancs a fájl(ok) törlésére szolgál, ez a megvalósítása:

```
public class Shell
{
...
    public void rm(final String[] cmd) throws ➥
        Exception
    {
        if (cmd.length < 2)
        {
            throw new Exception("USAGE: rm <path>");
        }

        final FileObject file = mgr.resolveFile(cwd, ➥
            cmd[1]);
        file.delete(Selectors.SELECT_SELF);
    }
...
}
```

Az mv parancs

A fájl mozgatása nem valódi másolás és törlés művelet abban az esetben, ha ugyanabban a fájlrendszerben történik. Miért? Mert ez ekkor csak

egy fájlnév átnevezést jelent. Ennek az az előnye, hogy tranzakcionális. Különböző fájlrendszerek között egy copy és delete művelettel lehet megvalósítani, de ez sajnos nem tranzakcionális. A példa az első esetet mutatja be. Míg a copy a mit akarok „magamra” másolni, addig a move a „magamat” hova szeretném másolni szemléletű.

```
public class Shell
{
...
    public void mv(final String[] cmd) throws ➥
        Exception
    {
        if (cmd.length < 3)
        {
            throw new Exception("USAGE: mv <src> <dest➥
                >");
        }

        final FileObject src = mgr.resolveFile(cwd, ➥
            cmd[1]);
        FileObject dest = mgr.resolveFile(cwd, cmd[2])➥
            ;
        src.moveTo(dest);
    }
...
}
```

Eseménykezelés

A fájlok menedzselése során nagyon hasznos szolgáltatás az eseményekre való automatikus reagálás. A rendszerek integrálása vagy egy telepíthető új fájl megjelenése, törlése vagy módosítása esetén kívánatos az ilyen működés. A VSF a következő 3 fájleseményt képes érzékelni és visszahívni a rájuk csatolt listener kódját:

- Új fájl jelent meg a figyelt könyvtárak valamelyikében (*fileCreated()*)
- Egy fájl módosult (*fileChanged()*)
- Egy fájlt töröltek (*fileDeleted()*)

Ezen 3 esemény visszahívott kódját minden olyan Java objektum képes kezelni, ami implementálja a *FileListener* VSF interfészt. A következő kód a *TestFileListener* osztály segítségével mutat be egy lehetséges implementációt, amik esetünkben csak képernyőre való kiírások. A visszahívott metódusok egy *FileChangeEvent* objektumot kapnak ajándékba, ez lehetővé teszi, hogy a kérdéses fájlt pontosan beazonosítsuk.

```
1  package org.cs.vfs;
2
3  import org.apache.commons.vfs2.FileChangeEvent;
4  import org.apache.commons.vfs2.FileListener;
5
6  public class TestFileListener implements FileListener
7  {
8      public void fileChanged(FileChangeEvent event) throws Exception
9      {
10       System.out.println( "Változott: " + event.getFile().getName());
11     }
12
13     public void fileDeleted(FileChangeEvent event) throws Exception
14     {
15       System.out.println( "Törölt: " + event.getFile().getName());
16     }
17
18     public void fileCreated(FileChangeEvent event) throws Exception
19     {
20       System.out.println( "Létrehozott: " + event.getFile().getName() );
21     }
22 }
```

A következő *FileEventProcessor* program bemutatja a VFS eseménykezelést. A *monitor* nevű *DefaultFileMonitor* objektum képes monitorozni a fájlrendszerben bekövetkező változásokat, amire most a fenti *TestFileListener* egy példányát akasztottuk rá eseménykezelő-

ként. A *monitor.addFile()* képes egy listát felvenni a megfigyelt könyvtárakról. Esetünkben most csak 1 db ilyen van, ami a */home/xxx* mappa. A monitorozás a *start()* metódussal indítható el, a *stop()*-pal pedig leállítható.

```
1  package org.cs.vfs;
2
3  import java.io.BufferedReader;
4  ...
5  public class FileEventProcessor
6  {
7
8    public static void test() throws Exception
9    {
10     FileSystemManager fsManager = VFS.getManager();
11     File testFile = null;
12
13     DefaultFileMonitor monitor = new DefaultFileMonitor( new TestFileListener() );
14
15     FileObject fileObj = fsManager.resolveFile("/home/xxx");
16
17     monitor.setDelay(100);
18     monitor.addFile(fileObj);
19     monitor.start();
20
21     Thread.sleep(60000);
22
23     monitor.stop();
24   }
25
26   public static void main(String[] args) throws Exception
27   {
28     test();
29   }
30 }
```

Az FTP protokoll

Eddig főleg a lokális fájlrendszerrel foglalkoztun pedig a VFS nagy előnye, hogy eltakarja a programozó elől azokat a különbségeket, amiket értelmesen érdemes, így a fájlok kezelését megpróbálja egységesíteni. Ez azt jelenti, hogy egy *FileObject* példány műveletei már függetlenek attól, hogy az ő fizikai fájlja hol van. Nézzük meg első példaként az FTP fájlrendszert, aminek a providerét az ismert Apache Common Net csomaggal (webhely: `http://commons.apache.org/proper/commons-net/`) implementálja a VFS.

```
1  package org.cs.vfs;
2
3  import java.io.File;
4  import org.apache.commons.vfs2.AllFileSelector;
5  import org.apache.commons.vfs2.FileObject;
6  import org.apache.commons.vfs2.FileSelector;
7  import org.apache.commons.vfs2.FileSystemException;
8  import org.apache.commons.vfs2.FileSystemManager;
9  import org.apache.commons.vfs2.FileSystemOptions;
10 import org.apache.commons.vfs2.VFS;
11 import org.apache.commons.vfs2.auth.StaticUserAuthenticator;
12 import org.apache.commons.vfs2.impl.DefaultFileMonitor;
13 import org.apache.commons.vfs2.impl.DefaultFileSystemConfigBuilder;
14 import org.apache.commons.vfs2.impl.DefaultFileSystemManager;
15 import org.apache.commons.vfs2.provider.sftp.SftpFileProvider;
```

```
16 import org.apache.commons.vfs2.provider.sftp.SftpFileSystemConfigBuilder;
17
18 public class TestFTP
19 {
20 ...
21   public static void testFtp() throws Exception
22   {
23     DefaultFileSystemManager manager = new DefaultFileSystemManager();
24     manager.addProvider("ftp", new FtpFileProvider());
25     manager.init();
26
27     String url = "ftp://infnav:ppppp@ftp.atw.hu/";
28     FileSystemOptions opts = new FileSystemOptions();
29     FtpFileSystemConfigBuilder.getInstance().setPassiveMode(opts, true);
30
31     FileObject ftpFile = manager.resolveFile(url, opts);
32     FileSelector fs = new AllFileSelector();
33     FileObject[] children = ftpFile.findFiles(fs);
34
35     System.out.println("Children of " + ftpFile.getName().getURI());
36
37     for (int i = 0; i < children.length; i++)
38     {
39       System.out.println(children[i].getName().getBaseName());
40     }
41     manager.close();
42   }
43 ...
44 } // end class
```

A fenti példa 23. sorában saját *manager* példányt készítünk, majd beregisztráljuk ide az *ftp* protokollt. Az *url* szerkezete világos. Nem kötelező, de mi a passive módot is beállítottuk, mert sok helyen szükséges a használata. A 33. sorban lekértük a „/” mappa összes bejegyzését, majd kilistáztuk azokat. Soha ne felejtsük el a 25. sor *init()* és a 41 sor *close()* metódusát meghívni, amikor saját manager-t használunk.

Az SFTP protokoll

Az SFTP használata hasonló, de itt az *SftpFileProvider* osztály implementációját a *JSch* csomaggal (webhelye: `http://www.jcraft.com/jsch/`) valósítja meg a VFS. A 27. sort azért érdemes megemlíteni, mert az bemutatja hogyan kell kikapcsolni a host, azaz a fájlszerver tanúsítvány ellenőrzését.

```
1  package org.cs.vfs;
2
3  import java.io.File;
4  import org.apache.commons.vfs2.AllFileSelector;
5  import org.apache.commons.vfs2.FileObject;
6  import org.apache.commons.vfs2.FileSelector;
7  import org.apache.commons.vfs2.FileSystemException;
8  import org.apache.commons.vfs2.FileSystemManager;
9  import org.apache.commons.vfs2.FileSystemOptions;
10 import org.apache.commons.vfs2.VFS;
11 import org.apache.commons.vfs2.auth.StaticUserAuthenticator;
12 import org.apache.commons.vfs2.impl.DefaultFileMonitor;
13 import org.apache.commons.vfs2.impl.DefaultFileSystemConfigBuilder;
14 import org.apache.commons.vfs2.impl.DefaultFileSystemManager;
15 import org.apache.commons.vfs2.provider.sftp.SftpFileProvider;
16 import org.apache.commons.vfs2.provider.sftp.SftpFileSystemConfigBuilder;
17
18 public class TestSFTP
```

```
19 {
20 ...
21   public static void testSFtp() throws Exception
22   {
23     String url = "sftp://inyiri:inyiri12@eaidevserv/svn-repo-co";
24     DefaultFileSystemManager manager = new DefaultFileSystemManager();
25     manager.addProvider("sftp", new SftpFileProvider());
26     FileSystemOptions opts = new FileSystemOptions();
27     SftpFileSystemConfigBuilder.getInstance().setStrictHostKeyChecking(opts, "no");
28     manager.init();
29
30     FileObject ftpFile = manager.resolveFile(url);
31     System.out.println(ftpFile.exists());
32
33     FileSelector fs = new AllFileSelector();
34     FileObject[] children = ftpFile.findFiles(fs);
35
36     for (int i = 0; i < children.length; i++)
37     {
38       System.out.println(children[i].getName().getBaseName());
39     }
40     manager.close();
41   }
42 ...
43 } // end class
```

HTTP protokoll

A következő példaként tekintsük a HTTP protokollt, aminek *HttpFileProvider* osztályát az Apache Commons HTTP client 3.1 és a Commons Codec segítségével implementálja a környezet (webhelyek: `http://hc.apache.org/httpclient-3.x/` és `http://commons.apache.org/proper/commons-codec/`).

```
1  package org.cs.vfs;
2
3  import java.io.File;
4  import org.apache.commons.vfs2.AllFileSelector;
5  import org.apache.commons.vfs2.FileObject;
6  import org.apache.commons.vfs2.FileSelector;
7  import org.apache.commons.vfs2.FileSystemException;
8  import org.apache.commons.vfs2.FileSystemManager;
9  import org.apache.commons.vfs2.FileSystemOptions;
10 import org.apache.commons.vfs2.VFS;
11 import org.apache.commons.vfs2.auth.StaticUserAuthenticator;
12 import org.apache.commons.vfs2.impl.DefaultFileMonitor;
13 import org.apache.commons.vfs2.impl.DefaultFileSystemConfigBuilder;
14 import org.apache.commons.vfs2.impl.DefaultFileSystemManager;
15 import org.apache.commons.vfs2.provider.sftp.SftpFileProvider;
16 import org.apache.commons.vfs2.provider.sftp.SftpFileSystemConfigBuilder;
17
18 public class TestHTTP
19 {
20 ...
21 public static void testHTTP() throws Exception
22   {
23     // http://[ username[: password]@] hostname[: port][ absolute-path]
24     String url = "http://svn.gep.hu/info2html.css";
25     // String url = "https://sapkapu";
26
27     DefaultFileSystemManager manager = new DefaultFileSystemManager();
28
29     manager.addProvider("http", new HttpFileProvider());
```

```
30      //manager.addProvider("https", new HttpFileProvider());
31      manager.init();
32
33      FileObject httpFile = manager.resolveFile(url);
34
35      byte[] bs = FileUtil.getContent(httpFile);
36      InputStream is = new ByteArrayInputStream(bs);
37
38      BufferedReader br = new BufferedReader(new InputStreamReader(is));
39
40      System.out.println(br.readLine());
41      System.out.println(br.readLine());
42
43      br.close();
44
45      manager.close();
46    }
47  ...
48  } // end class
```

A fenti kódban semmi nehéz nincs, a 29. sorban hozzáadjuk a manager-hez a *http* protokollt. A 35. sorban a *FileUtil* segítségével egy mozdulattal áttesszük a fájl tartalmát egy *bs* nevű byte tömbbe, majd erre egy *InputStream* példányt, utána az ezt transzformáló *BufferedReader* objektumot kreálunk. Ez utóbbival már soronként tudjuk olvasni az áthozott *info2html.css* fájlt. A VFS a webdav protokollt az Apache Jackrabbit csomaggal (webhelye: `http://jackrabbit.apache.org/`) implementálja.

A JAR fájl olvasása

Példaként tekintsük ezt a kis tesztprogramot:

```
public static void testJar() throws Exception
{
  FileSystemManager fsManager = VFS.getManager();

  FileObject jarFile = fsManager.resolveFile("jar:///➥
      home/xxx/test.jar");
  FileObject[] children = jarFile.getChildren();
  System.out.println("Children of " + jarFile.getName➥
      ().getURI());

  for (int i = 0; i < children.length; i++)
  {
    System.out.println(children[i].getName().➥
        getBaseName());
  }
}
```

Ez egy nagyon kellemes lehetőség, mert minden csomagolt fájltípust (*zip*, *jar*, *tar*, *tgz*, *tbz2*, *gzip*, *bzip2*) ezzel a módszerrel kezelni tudunk.

Összefoglalás

Amikor sokféle és esetleg távoli fájlokat kezelünk vagy kell egy eseménykezelő, akkor érdemes mindig a VFS-re gondolni és megvizsgálni, hogy segítségével támogatható-e a megoldás. Ez a példa például egy *s3* protokollt használ, amivel az (webhely: `https://code.google.com/p/vfs-s3/`) Amazonnal kommunikálhatunk. A protokollról többet itt olvashatunk: `http://aws.amazon.com/s3/`.

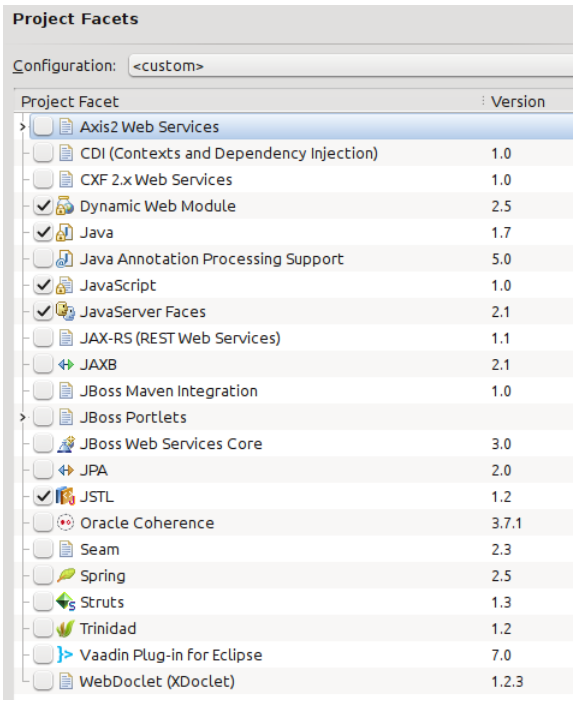
```
// Create bucket
FileSystemManager fsManager = VFS.getManager();
FileObject dir = fsManager.resolveFile("s3://simpe➥
    bucket");
dir.createFolder();

// Upload file to S3
FileObject dest = fsManager.resolveFile("s3://test➥
    bucket/backup.zip");
FileObject src = fsManager.resolveFile(new File("/path➥
    /to/local/file.zip").getAbsolutePath());
dest.copyFrom(src, Selectors.SELECT_SELF);
```

Az *s3* provider (*vfs-s3-bin.jar*) pedig innen tölthető le: `https://code.google.com/p/vfs-s3/downloads/list`. Tekintettel arra, hogy a forrásprogram is elérhető, innen is tanulmányozhatjuk, hogy egy új provider-t miképpen érdemes elkészíteni.

7. Apache Commons - FileUpload

Számos webalkalmazásban igény merül fel arra, hogy egy fájlt tölthessünk fel a böngésző segítségével, amit majd a későbbiekben fel kell dolgozni. Jó példa erre egy browser alapú e-mail kliens, amikor a mellékelt dokumentumokat is ilyen módszerrel töltjük fel. Az itt felmerülő technikai feladatok könnyítésére szolgál a *FileUpload* könyvtár.

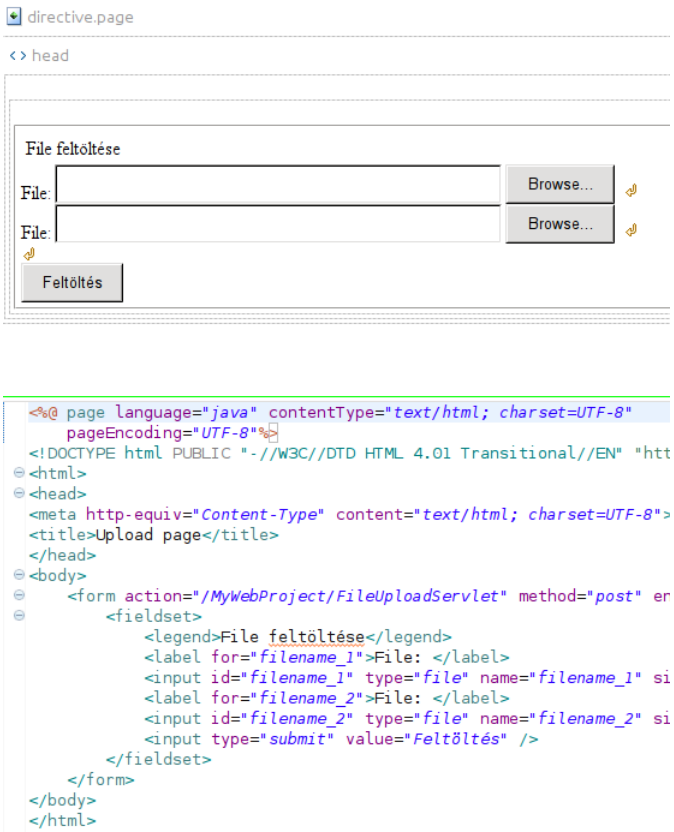
Ebben a cikkben létrehozunk egy webalkalmazást (a neve *MyWebProject* lesz) és ezen keresztül bemutatjuk a *FileUpload* könyvtár használatát. Az Eclipse fejlesztői környezetet fogjuk használni, amiben egy új *Dynamic Web Project*-et hozunk létre. Runtime környezetként az *Apache Tomcat 6.x* lett kiválasztva. A 7.1. ábra a projekthez bepipált komponenseket, azaz *Project Facet*-eket mutatja.



7.1. ábra: Project Facets

A példa alkalmazásunkhoz még a következő 2 *jar* hozzáadása is szükséges:

- *Apache Commons FileUpload* könyvtár (webhely: `commons.apache.org/proper/commons-fileupload`)
- *Apache Commons IO* könyvtár (webhely: `http://commons.apache.org/proper/commons-io/`)



7.2. ábra: Upload.jsp az editorban

Adjunk hozzá egy JSP lapot a projekthez, amit *Upload.jsp* néven mentettünk el (7-1. Programlista). Ahogy azt a 7.2. ábráról is sejthetjük, az Eclipse meglehetősen fejlett eszközökkel

rendelkezik a *html/jsp* lap szerkesztést illetően. Egyszerre láthatjuk és szerkeszthetjük a vizuális és forráskód nézetet is.Az *Upload.jsp* kódja egyszerű, tartalmaz 2 fájlfeltöltő html vezérlőt, amivel egyszerre 2 fájl feltöltését is kezdeményezhetjük. Ezenfelül természetesen van egy submit gomb is. A címléket a *for* kulcsszóval rendeltük a megfelelő *input* komponenshez. A 11. sorban lévő *form* tag szerepe több szempontból is fontos:

- Az *action* attribútumnál megadtuk a nemsokára ismertetésre kerülő *FileUploadServlet* feltöltő servletet, mint azt az URL-t, ahova a *form*ot el kel *post*-olni.
- Megadtuk az *enctype* értékét helyesen, hiszen fájlt szeretnénk feltölteni.

```
1  // 7-1. Programlista: Upload.jsp
2
3  <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
4  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8  <title>Upload page</title>
9  </head>
10 <body>
11   <form action="/MyWebProject/FileUploadServlet" method="post" enctype="multipart/form-data">
12     <fieldset>
13       <legend>File feltöltése</legend>
14       <label for="filename_1">File: </label>
15       <input id="filename_1" type="file" name="filename_1" size="50" value="Böngészés"/> <br/>
16       <label for="filename_2">File: </label>
17       <input id="filename_2" type="file" name="filename_2" size="50" value="Böngészés"/> <br /> <br/>
18       <input type="submit" value="Feltöltés" />
19     </fieldset>
20   </form>
21 </body>
22 </html>
```

A formon a *FileUploadServlet* class (7-2. Programlista) kódjára hivatkoztunk, így most nézzük meg alaposan. A forrásprogram több olyan részt is tartalmaz, amire nem megy a vezérlés, de oktatási céllal betettük ezeket is, hogy lássa az olvasó a feldolgozás lehetőségeit is. A kód jelenleg ezeket a részeket nem használja, azaz nem megy rá soha a vezérlés:

- 36-51. sorok, azaz a *newDiskFileItemFactory()* metódus
- 134-142. sorok: *processUploadedFile()* metódus
- 150-155. sorok: *processFileAsStream()* metódus
- 157-161. sorok: *processFileInMemory()* metódus

A fenti 4 programrészlet bármikor felhasználható, azonban mi most csak a fájl feltöltés a lokális fájlrendszerbe funkciót aktivizáltuk (144-148. sorok *saveFile()* metódusa). A submit gomb megnyomására a 62. sornál kezdődő *process()* metódus indul el. A 65-81 sorok között egy *progressListener* nevű változót hozunk létre, ami nem szükséges egy fájl feltöltéshez, de lehetővé teszi, hogy annak előrehaladását megjeleníthessük. Ez úgy valósul majd meg, hogy a 102. sorban regisztráltuk ezt a listener-t, így az időnként visszahívódik, amellyel az ekkor lefutó kód a monitorozás lehet. Mindez – ahogy már

említettük – csak opcionális elem. A 84. sorban létrehozott *factory* fogja előkészíteni a feltöltési folyamat feldolgozását. A 89-90. sorokban a feltöltéshez használt átmeneti lokális könyvtárat adtuk meg, amit most a servlet környezeti paraméteréből nyerünk ki (a változó és az értéke a *web.xml* fájlban van definiálva). A 96. sor *upload* változója fogja vezérelni magát a feltöltés feldolgozást, így ennek a szerepe kulcsfontosságú. A 105. sorból pedig azt tanulhatjuk meg, hogy az egyes *FileItem* objektumok bekerülnek egy listába, amit utána már könnyen fel tudunk dolgozni. Ez a feldolgozás a 110-125 sorok között történik, ami azt jelenti, hogy végigmegyünk az *items* listán és ha az nem *form* elem, hanem egy küldött fájlt reprezentál, akkor elmentjük a lokális fájlrendszerbe. Itt jegyezzük meg, hogy a kódban az *item.getName()* hívás a fájl eredeti nevét adja vissza, így ezzel is mentjük el.

```
1  // 7-2. Programlista: FileUploadServlet.java
2
3  package org.cs.fileupload;
4
5  import java.io.File;
6  import java.io.IOException;
7  import java.io.InputStream;
8  import java.util.Iterator;
9  import java.util.List;
10 import javax.servlet.ServletContext;
11 import javax.servlet.ServletException;
12 import javax.servlet.http.HttpServlet;
13 import javax.servlet.http.HttpServletRequest;
14 import javax.servlet.http.HttpServletResponse;
15 import org.apache.commons.fileupload.FileItem;
16 import org.apache.commons.fileupload.FileItemFactory;
17 import org.apache.commons.fileupload.FileUploadException;
18 import org.apache.commons.fileupload.ProgressListener;
19 import org.apache.commons.fileupload.disk.DiskFileItemFactory;
20 import org.apache.commons.fileupload.servlet.FileCleanerCleanup;
21 import org.apache.commons.fileupload.servlet.ServletFileUpload;
22 import org.apache.commons.io.FileCleaningTracker;
23
24 public class FileUploadServlet extends HttpServlet
25 {
26   private static final long serialVersionUID = 1L;
27
28   public static int KB = 1024;
29   public static int MB = 1024 * 1024;
30
31   public FileUploadServlet()
32   {
33     super();
34   }
35
36   /**
37    *
38    * @param context
39    * @param repository
40    * @return
41    */
42   public static DiskFileItemFactory newDiskFileItemFactory(
43       ServletContext context, File repository)
44   {
45     FileCleaningTracker fileCleaningTracker = FileCleanerCleanup
46         .getFileCleaningTracker(context);
47     DiskFileItemFactory factory = new DiskFileItemFactory(
48         DiskFileItemFactory.DEFAULT_SIZE_THRESHOLD, repository);
49     factory.setFileCleaningTracker(fileCleaningTracker);
```

```
50      return factory;
51    }
52
53
54    /**
55     *
56     * @param request
57     * @param response
58     * @throws ServletException
59     * @throws IOException
60     * @throws FileUploadException
61     */
62    public void process(HttpServletRequest request, HttpServletResponse response)
63        throws Exception
64    {
65      // Egy új progress listener
66      ProgressListener progressListener = new ProgressListener()
67      {
68        public void update(long pBytesRead, long pContentLength, int pItems)
69        {
70          //System.out.println("We are currently reading item " + pItems);
71          if (pContentLength == -1)
72          {
73            //System.out.println("So far, " + pBytesRead
74            //    + " bytes have been read.");
75          } else
76          {
77            //System.out.println("So far, " + pBytesRead + " of "
78            //    + pContentLength + " bytes have been read.");
79          }
80        }
81      };
82
83      // Egy disk-based file items factory létrehozása
84      DiskFileItemFactory factory = new DiskFileItemFactory();
85
86      // Egy repository, ami biztosítja a biztonságos átmeneti helyet
87      ServletContext servletContext = this.getServletConfig().getServletContext();
88
89      File repository = (File) servletContext.getAttribute("javax.servlet.context.tempdir");
90      factory.setRepository(repository);
91
92      // Memóriahasználat korlát
93      factory.setSizeThreshold(4 * KB);
94
95      // Kezeli a feltöltést
96      ServletFileUpload upload = new ServletFileUpload(factory);
97
98      // Max. ekkore lehet a fájl (nem kötelező)
99      upload.setSizeMax(10 * MB);
100
101     // Beállítani a progress listener-t
102     upload.setProgressListener(progressListener);
103
104     // A request elemzése
105     List<FileItem> items = upload.parseRequest(request);
106
107     // ------------------------------------
108     // A feltöltött tartalom feldolgozása
109     // ------------------------------------
110     Iterator<FileItem> iter = items.iterator();
111     while (iter.hasNext())
112     {
113       FileItem item = iter.next();
114
```

```
115         if (item.isFormField())
116         {
117           processFormField(item);
118 
119         } else
120         {
121           File to = new File("/home/temp/"+item.getName());
122           saveFile(item, to);
123         }
124       }
125     }
126 
127     // A form mezői
128     private void processFormField(FileItem item)
129     {
130       String name = item.getFieldName();
131       String value = item.getString();
132     }
133 
134     // Néhány adathoz való hozzájutás demója
135     private void processUploadedFile(FileItem item)
136     {
137       String fieldName = item.getFieldName();
138       String fileName = item.getName();
139       String contentType = item.getContentType();
140       boolean isInMemory = item.isInMemory();
141       long sizeInBytes = item.getSize();
142     }
143 
144     // Elmentjük a fájlrendszerben
145     private void saveFile(FileItem item, File to) throws Exception
146     {
147       item.write(to);
148     }
149 
150     // Amennyiben stream-ként akarjuk feldolgozni
151     private void processFileAsStream(FileItem item) throws Exception
152     {
153       InputStream uploadedStream = item.getInputStream();
154       uploadedStream.close();
155     }
156 
157     // Memóriában való feldolgozás módja
158     private void processFileInMemory(FileItem item) throws Exception
159     {
160       byte[] data = item.get();
161     }
162 
163     protected void doGet(HttpServletRequest request,
164         HttpServletResponse response) throws ServletException, IOException
165     {
166       try
167       {
168         process(request, response);
169       } catch (Exception e)
170       {
171         // TODO Auto-generated catch block
172         e.printStackTrace();
173       }
174     }
175 
176     protected void doPost(HttpServletRequest request,
177         HttpServletResponse response) throws ServletException, IOException
178     {
179       try
```

```
180     {
181       process(request, response);
182     } catch (Exception e)
183     {
184       // TODO Auto-generated catch block
185       e.printStackTrace();
186     }
187   }
188 }
```

Most, hogy megértettük a programot, nézzük meg a jelenleg nem használt, de bármikor bevethető további feldolgozási lehetőségeket is. Ezeket a *saveFile()* helyéről, ahelyett hívhatnánk meg. A 150-155 sorok között lévő *processFileAsStream()* metódus azt mutatja meg, hogy a feltöltött fájl tartalmát miképpen kaphatjuk meg egy *InputStream* objektumként. Ez sokszor fontos, például gondoljunk arra, hogy egy excel fájlt töltünk fel és HTML táblázatként akarjuk utána megjeleníteni. Ekkor a teljes fájlt így célszerű átvenni és az excel formátum kezelésére alkalmas objektumnak átadni. A 157-161 sorok közötti *processFileInMemory()* metódus azt mutatja be, hogy akár egyből egy byte tömbbe is berakhatjuk a feltöltött fájlt a további feldolgozáshoz. A 7-3. Programlista az alkalmazás *web.xml* fájlját tartalmazza.

```
1  // 7-3. Programlista: web.xml
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-➥
         app_2_5.xsd"
6    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-➥
         app_2_5.xsd"
7    id="WebApp_ID" version="2.5">
8    <display-name>MyWebProject</display-name>
9
10   <welcome-file-list>
11     <welcome-file>index.jsp</welcome-file>
12   </welcome-file-list>
13
14   <servlet>
15     <servlet-name>Faces Servlet</servlet-name>
16     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
17     <load-on-startup>1</load-on-startup>
18   </servlet>
19
20   <servlet-mapping>
21     <servlet-name>Faces Servlet</servlet-name>
22     <url-pattern>/faces/*</url-pattern>
23   </servlet-mapping>
24
25   <context-param>
26     <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
27     <param-value>resources.application</param-value>
28   </context-param>
29
30   <context-param>
31     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
32     <param-value>client</param-value>
33   </context-param>
34
35   <context-param>
36     <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-name>
```

```
37      <param-value>true</param-value>
38    </context-param>
39
40    <context-param>
41      <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
42      <param-value>true</param-value>
43    </context-param>
44
45    <context-param>
46      <param-name>org.apache.myfaces.DETECT_JAVASCRIPT</param-name>
47      <param-value>false</param-value>
48    </context-param>
49
50    <context-param>
51      <param-name>org.apache.myfaces.AUTO_SCROLL</param-name>
52      <param-value>true</param-value>
53    </context-param>
54
55    <listener>
56      <listener-class>org.apache.myfaces.webapp.StartupServletContextListener</listener-class>
57    </listener>
58
59    <!-- listener>
60      <listener-class>org.apache.commons.fileupload.servlet.FileCleanerCleanup</listener-class>
61    </listener -->
62
63    <servlet>
64      <description></description>
65      <display-name>FileUploadServlet</display-name>
66      <servlet-name>FileUploadServlet</servlet-name>
67      <servlet-class>org.cs.fileupload.FileUploadServlet</servlet-class>
68      <init-param>
69        <param-name>javax.servlet.context.tempdir</param-name>
70        <param-value>/home/temp</param-value>
71      </init-param>
72    </servlet>
73
74    <servlet-mapping>
75      <servlet-name>FileUploadServlet</servlet-name>
76      <url-pattern>/FileUploadServlet</url-pattern>
77    </servlet-mapping>
78
79  </web-app>
```

A *web.xml* 59-61 sorában most megjegyzésbe tettük a *FileCleanerCleanup* listener osztály használatát. Erre csak akkor van szükség, ha a *DiskFileItem* osztályt is használjuk, ami az alapértelmezett implementációja a *FileItem* class-nak. Ilyenkor célszerű a *newDiskFileItemFactory()* metódussal megszerezni a *DiskFileItemFactory* objektumot. Itt használható a *org.apache.commons.io.FileCleaningTracker*, amit a következő pontban jobban meg is érthetjünk majd. Befejezésül még megadjuk a webalkalmazás induláskor generált *indes.jsp* faces lapját, aminek az URL-je: `http://localhost:8080/MyWebProject/faces/index.jsp`.

```
1  <%@ page language="java" contentType="text/html;charset=UTF-8"%>
2  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
3  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
4
5  <f:view>
6  <html><head><title><h:outputText value="Alma" /></title>
7  </head><body></body></html></f:view>
```

8. Apache Commons - IO

Az Apache Commons IO (webhely: `http://commons.apache.org/proper/commons-io/index.html`) azért született meg, hogy a Java programokban az IO kezeléséhez eszközöket adjon. Tartalmaz néhány statikus utility osztályt, támogatja a különféle input és output műveleteket, amikhez szűrőket is a rendelkezésünkre bocsát. Itt is található egy eseménykezelő alrendszer, ami lokális esetben jobb alternatíva lehet a VFS megoldásához képest is.

Az IO könyvtár 6 fő részből áll:

- Utility osztályok: Statikus metódusokkal támogat néhány közhasznú feladatot.
- Input: Hasznos *InputStream* és *Reader* implementációk.
- Output: Hasznos *OutputStream* és *Writer* implementációk.
- Filterek: Különféle fájlszűrők.
- *Comparator*-ok: *java.util.Comparator* implementációk a fájlokhoz.
- File Monitor: A fájlrendszer események monitorozása

A FileUtils osztály

A fájl megérintése (touch)

Bemelegítésként nézzük meg azt az egyszerű és ismert műveletet, amit a fájl megérintésének (touch) hívunk és az a célja, hogy az utolsó módosítás dátumát a jelen pillanatra állítsa. A következő példa *touch()* metódusa mindezt megteszi, ha a fájl nem létezik, akkor üresen létrehozza azt.

```
package org.cs.io;

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;

public class TestTouch
{
    public static void main(String[] args)
    {
      try
      {
            File file = new File("Touch.dat");
            FileUtils.touch(file);
      } catch (IOException e) {
            e.printStackTrace();
      }
    }
}
```

Egy fájl tartalmának Stringbe olvasása

Tegyük fel, hogy adott egy szövegfájl, aminek a tartalmát 1 mozdulattal szeretnénk a memóriába tölteni, célszerűen egy String objektumba. A következő példa *readFileToString()* metódusa mindezt megteszi, alapértelmezetten feltételezve, hogy a fájl tartalma UTF-8-ban van kódolva. Ez utóbbi nem mindig teljesül, ezért opcionálisan a kódolást is megadhatjuk a Javaban szokásos módon.

```
package org.cs.io;

import org.apache.commons.io.FileUtils;
import java.io.File;
import java.io.IOException;

...
File file = new File("sample.txt");
String content = FileUtils.readFileToString(file);

String ENCODING="UTF-8";
String content = FileUtils.readFileToString(file , ➥
    ENCODING);
...
```

Egy String kiírása fájlba

Az előző feladat ellenkezője ugyanolyan gyakori eset, ekkor egy String objektumot kell fájlba menteni, amit a *writeStringToFile()* old meg. Ennek is létezik olyan változata, amely egy megfelelő kódkészlettel kódolva menti le a Stringet, bár a modern környezetekben ennek már talán nem sok értelme van. A lehetőséget azonban még támogatni kell, mert a régebbi rendszerek

még nem unicode alapon működnek és a következő években sem várható itt jelentős előrelépés.

```
package org.cs.io;

import org.apache.commons.io.FileUtils;
import java.io.File;
import java.io.IOException;

...
String data = "Alma a fa alatt";
FileUtils.writeStringToFile(file, data);

String ENCODING="UTF-8";
FileUtils.writeStringToFile(file, data, ENCODING);
```

Soronkénti írás-olvasás

Az előzőekben a szövegfájl teljes tartalmát egy egységként kezeltük, pedig azt leggyakrabbak sorokból álló adatszerkezetnek képzeljük el. A *readLines()* képes arra, hogy ennek megfelelően egy *List<String>* objektumba töltse a sorokat, amit a következő példa be is mutat.

```
package org.cs.io;

import java.io.File;
import java.io.IOException;
import java.util.List;
import org.apache.commons.io.FileUtils;

public class TestFileUtils
{
...
  public static void readLines()
  {
    File file = new File("sample.txt");

    try
    {
      List<String> content=FileUtils.readLines(file);

      for (String line : content)
      {
      System.out.println(line);
      }
    } catch (IOException e)
    {
      e.printStackTrace();
    }
  }
...
}
```

Gyakorlatilag ennek az ellentétét lehet elérni a *writeStringToFile()* metódussal, ami egy új sort fűz a textfájlhoz.

```
package org.cs.io;

import java.io.File;
import java.io.IOException;
import java.util.List;
import org.apache.commons.io.FileUtils;

public class TestFileUtils
{
...
  public static void writeLines()
  {
    try
    {
      File file = new File("sample.txt");
      String data = "Learning Java Programming";

      FileUtils.writeStringToFile(file, data);
    } catch (IOException e)
    {
      e.printStackTrace();
    }
  }
...
}
```

Fájlok másolása

Gyakori jelenség, hogy a fájlmásolást a programozók újra és újra elkészítik, pedig az ilyen feladatokat érdemes olyan könyvtári rutinokra bízni, amik ezt már jól letesztelt módon tartalmazzák. Esetünkben a *copyFile()* metódus is megvalósítja, nézzük meg a használatát! A *source* a másolandó fájl, a *target* pedig a célfájl. Ez utóbbi persze nem létezik feltétlenül, hiszen alapvetően most akarjuk létrehozni. A *targetDir* egy könyvtárat reprezentál most, esetünkben ez az *OS* temp könyvtár, de ennek nincs semmi jelentősége a példa szempontjából. A *copyFile()* hívás után elkészül a fájl másolata. A *copyFileToDirectory()* pedig a *source* fájlt bemásolja ugyanilyen néven a *targetDir* könyvtárba.

```
public class TestFileUtils
{
...
    public static void testFileCopy()
    {
        File source = new File("january.doc");
        File target = new File("january-backup.doc");
        File targetDir = new File(System.getProperty("➥
            java.io.tmpdir"));

        try
        {
            System.out.println("Copying " + source + "➥
                 file to " + target);
            FileUtils.copyFile(source, target);

            System.out.println("Copying " + source + "➥
                 file to " + targetDir);
            FileUtils.copyFileToDirectory(source, ➥
                targetDir);
        } catch (IOException e)
        {
            // Errors will be reported here if any ➥
                error occures during copying
            // the file
            e.printStackTrace();
        }
    }
...
}
```

Teljes könyvtár másolása

Egy speciálisabb eset – például archiválás esetén – egy teljes könyvtár másolása. Példánkban az

srcDir a másolandó, a *destDir* könyvtár pedig a célhely, ami létre lesz hozva. Magát a másolást a *copyDirectory()* metódus végzi, amiről jegyezzük meg az alábbiakat:

- a gyerek könyvtárakat is másolja
- amikor nem létezik a célkönyvtár, azt létrehozza
- a fájl dátuma megmarad

```
package example.commons.io;

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;

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

        String source = "/alma/source";
        File srcDir = new File(source);
        String destination = "/alma/target";
        File destDir = new File(destination);

        try {
            FileUtils.copyDirectory(srcDir, destDir);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Teljes könyvtár mozgatása

A könyvtárak mozgatása valószínűleg a másolásnál is gyakoribb művelet, amikor az a cél, hogy az archivált adatok az eredeti helyen már ne jelenjenek meg. Ilyenkor fontos, hogy a *destDir* nem lehet már létező könyvtár.

```
package example.commons.io;

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;

public class DirectoryMove {
    public static void main(String[] args) {
        String source = "/alma/source";
        File srcDir = new File(source);

        String destination = "/alma/target";
        File destDir = new File(destination);

        try {
            FileUtils.moveDirectory(srcDir, destDir);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Másolás URL-ről

Egy hasznos lehetőség a HTTP URL-ről való másolás közvetlen támogatása, ami a *copyURLToFile()* funkció személyében valósul meg. A forrás egy HTTP protokollon elérhető erőforrás, a cél pedig egy helyi fájl, ahova az URL-ről letöltött byte-okat másoljuk.

```
package example.commons.io;

import org.apache.commons.io.FileUtils;

import java.io.File;
import java.io.IOException;
import java.net.URL;

public class URLToFile {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://index.hu");
            File destination = new File("index.html");

            FileUtils.copyURLToFile(url, destination);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Könyvtárak törlése

Az alábbiakban az egyik példában bemutatjuk a teljes könyvtár, míg a másikban csak a könyvtár tartalmának törlését. Az alábbi példa az elsőt szemlélteti. A törlés rekurzív, azaz az alkönyvtárak is törlődni fognak, hiszen annak nincs értelme, hogy csak a szülő szűnjön meg. Mi lenne a gyerekekkel? Hiba esetén (például nincs jogunk a törlésre) *IOException* dobódik.

```
public class TestFileUtils
{
...
  public static void testDeleteFile()
  {
        try
        {
            File directory = new File("/home/Temp/Data"➥
                );
            FileUtils.deleteDirectory(directory);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
...
}
```

A *cleanDirectory()* metódus nem törli a könyvtárat, azonban annak tartalmát igen. A következő példában a */home/xxx* mappában létrehozunk 2 fájlt, majd letöröljük őket. Az egyik fájl ponttal kezdődik, ami UNIX esetén a rejtett fájl, erre is remekül működik az eljárás.

```
public class TestFileUtils
{
...
  public static void cleanDirectory() throws ➥
      IOException
  {
    File path = new File("/home/xxx");

      FileUtils.touch(new File(path, "alma"));
      FileUtils.touch(new File(path, ".alma"));

    FileUtils.cleanDirectory(path);
  }
...
}
```

Fájlok keresése

A fájlok keresése általában azt jelenti, hogy egy megadott mappától indulva megadott tulajdonságokkal rendelkező fájlokat gyűjtünk össze, természetesen majd valamilyen későbbi feldolgozási céllal. Minderre a legegyszerűbb módszert talán az alábbi példában alkalmazott *listFiles()* metódus valósítja meg. Esetünkben a keresés kiinduló pontja (gyökere, azaz *root*) a */home/tanulas* könyvtár. Az *extensions* tömbben a keresett fájlkiterjesztéseket adtuk meg, azaz word, excel és powerpoint fájlokat szeretnénk találni. Amennyiben a *recursive* értékét *true*-val használjuk, úgy az almappákban is keresni fog a *listFiles()*, aminek az eseményét egy *File* gyűjteményben adja majd vissza.

```
public class TestFileUtils
{
...
    public static void searchFiles()
  {
    File root = new File("/home/tanulas");

    try
    {
      String[] extensions =
      {
        "doc", "xls", "ppt"
      };
      boolean recursive = true;

      Collection<File> files = FileUtils.listFiles(➥
          root, extensions, recursive);

      for (File file : files)
      {
        System.out.println("File = " + file.➥
            getAbsolutePath());
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
...
}
```

A fenti módon használt *listFiles()* persze nem tud túl sokat, általában ennél sokat többet szeretnénk, ezért létezik egy másik alakja is:

```
Collection<File> listFiles(File directory,IOFileFilter➥
    fileFilter, IOFileFilter dirFilter);
```

Az IOFileFilter az ismert Java szabványos interface-ek kiterjesztése, ezért ezeket mindenütt használhatjuk, ahol azokat is:

- *java.io.FileFilter*
- *java.io.FilenameFilter*

A fenti metódus úgy működik, hogy a keresés a *directory* paraméterben specifikált könyvtártól indul. Azok a fájlok lesznek legyűjtve, amik a *fileFilter* szűrőn átmennek. Ilyen szűrő sok van, nemsokára részletezzük őket. A *dirFilter* paraméter opcionális. Amennyiben *null* értéket adunk ide, úgy a keresés nem lesz rekurzív és csak a *directory* mappára korlátozódik. Ennek az az alternatívája, hogy az almappákban is keres a metódus, ekkor itt valamilyen könyvtár szűrőt kell megadnunk, amit szintén lentebb részletezünk.

A filterek használata

A Commons IO egy nagy *IOFileFilter* interfészt megvalósító gyűjteménnyel rendelkezik, mindenképpen fontos áttekinteni őket. A most következő példák célja, hogy bemutassa a szűrő objektumok konstruálásának tipikus módjait, azonban nem cél minden szűrő részletes bemutatása. Nézzük meg először azt a szűrőt, ami a mappákon kívül mindent kiszűr, azaz a könyvtárakat engedi tovább. A neve a *DirectoryFileFilter*, aminek egy példányához a kódban bemutatott módon férhetünk hozzá. A *listFiles()* 3. paraméterében ilyet lehet megadni.

```
public static void testDirectoryFilter()
{
  File dir = new File(".");
  String[] files = dir.list(DirectoryFileFilter.➥
      INSTANCE);
  for (int i = 0; i < files.length; i++)
  {
    System.out.println(files[i]);
  }
}
```

A következő kis forráskód a fájl utolsó módosítsa szerinti életkora alapján szűr, ezért *AgeFileFilter* a neve. A mintakódban azokat a fájlokat kérjük le, amelyek 1 napnál öregebbek.

```
public static void testAgeFilter()
{
  File dir = new File("/home/xxx");
  long cutoff = System.currentTimeMillis() - (24 * 60 * 60 * 1000);
  String[] files = dir.list(new AgeFileFilter(cutoff));

  for (int i = 0; i < files.length; i++)
  {
    System.out.println(files[i]);
  }
}
```

A *WildcardFileFilter* a fájl nevének mintázata alapján szűr.

```
public static void testWildCardFileter()
{
  File dir = new File(".");
  FileFilter fileFilter = new WildcardFileFilter("*test*.java~*~");
  File[] files = dir.listFiles(fileFilter);
  for (int i = 0; i < files.length; i++)
  {
    System.out.println(files[i]);
  }
}
```

Ez már sokkal hatékonyabb feladatmegoldó filter, mint a bevezetőben megadott lehetőség, amit a *SuffixFileFilter* segítségével tudnánk legkönyebben megvalósítani.

```
FileFilter fileFilter1 = new SuffixFileFilter(".class");
FileFilter fileFilter2 = new SuffixFileFilter(".java");
```

Az olvasó mire idáig ért bizonyára elgondolkodott azon, hogy a gyakorlatban általában több szűrő kompozíciója ad ki egy feladathoz illeszkedő szűrőt. Erre van megoldás, ugyanis létetik 3 speciális filer, aminek pont ez a feladata:

- *AndFileFilter* → Két szűrő *ÉS* logikai kapcsolata
- *OrFileFilter* → Két szűrő *VAGY* logikai kapcsolata
- *NotFileFilter* → Egy szűrő tagadás

Mindezek jobb megértése kedvéért tekintsük az ezt bemutató példánkat!

```
public static void testAndOrFilter()
{
  File dir = new File(".");
    String[] files = dir.list(
        new AndFileFilter(
          new AndFileFilter(
            new PrefixFileFilter("A"),
            new OrFileFilter(
              new SuffixFileFilter(".class"),
              new SuffixFileFilter(".java")
            )
          ),
          new NotFileFilter(
            DirectoryFileFilter.INSTANCE
          )
        )
    );
  for ( int i=0; i<files.length; i++ ) {
    System.out.println(files[i]);
  }
}
```

A további fontos filterek a itt csak egy felsorolásban adjuk meg:

- *NameFileFilter* → a fájl neve alapján szűr
- *RegexFileFilter* → reguláris kifejezést adhatunk meg a fájl nevére (példa: *new RegexFileFilter("^.*[tT]est(-\\d+)?\\.java$");*)
- *SizeFileFilter* → A fájl mérete legyen nagyobb a megadottnál. (példa: *new SizeFileFilter(1024 * 1024);* Az 1MB-nál nagyobb fájlokat engedi át)
- *CanReadFileFilter* → a fájl olvasható
- *CanWriteFileFilter* → a fájl írható
- *FalseFileFilter* → csak a rendes fájlokat engedi tovább, a mappákat nem
- *EmptyFileFilter* → üres könyvtár vagy fájl
- *HiddenFileFilter* → a rejtett fájlok
- *FalseFileFilter* → semmit sem enged át a szűrőn
- *TrueFileFilter* → mindent átenged a szűrőn

A FileUtils egyéb lehetőségei

Befejezésül áttekintjük a *FileUtils* class még nem említett, de hasznos további metódusait. Egy könyvtár teles mérete így kérdezhető le:

```
long size=FileUtils.sizeOfDirectory(new File("/opt"));
```

Egy fájl mérete pedig így:

```
File file = new File("/home/valami.bin");
long size = file.length();
String display=FileUtils.byteCountToDisplaySize(size);

System.out.println("Name____=_" + file.getName());
System.out.println("size____=_" + size);
System.out.println("Display_=_" + display);
```

Sokszor jól jön, ha a fájlról egy CRC számot tudunk elmenteni és időnként azt visszaellenőrizni, amit a *checksumCRC32()* metódus tesz meg:

```
File file = new File("/home/alma.txt"));
long crc = FileUtils.checksumCRC32( file );
```

Két fájl byte-ról byte-ra való egyezésének vizsgálatát a *contentEquals()* szolgáltatja:

```
boolean contentEquals(File file1,File file2) throws ➥
    IOException
```

A *contentEqualsIgnoreEOL()* szöveg alapon teszi mindezt, azaz soronként teszi meg az összehasonlítás. Emiatt a kódlap paramétert is meg kell neki adni. A *convertFileCollectionToFileArray()* metódus egy *File[]* tömböt ad vissza egy *File* kollekcióból. Az *isFileNewer()* egy család, ahol többféleképpen tudunk időpontot megadni és az eredmény akkor lesz *true*, ha a fájl fiatalabb a megadott szempontnál. Az *isFileOlder()* hasonló, de itt a fájlnak öregebbnek kell lennie. Unix alatt létezik a szimbolikus link, aminek igaz voltát az *isSymlink()* logikai metódussal lehet lekérdezni. A *moveFile()* fájlt tud mozgatni, ilyen a paraméterezése:

```
void moveFile(File srcFile, File destFile) throws ➥
    IOException
```

Van 2 érdekes metódus:

```
FileInputStream openInputStream(File file) throws ➥
    IOException
FileOutputStream openOutputStream(File file) throws ➥
    IOException
```

Az egyik egy input, a másik egy output stream byte csatornát képes nyitni az adott fájlra. A *readFileToByteArray()* egy *byte[]* objektumba másolja a fájl tartalmát. A *FileUtils* class ezenkívül rendelkezik még számos, különféle helyzetekben jól használható *write...()* metódussal.

IOUtils

Az *IOUtils* class a fájl↔memória változó közötti adatmozgatás támogató statikus metódusok gyűjteménye. A következőkben 2 példát adunk erre.

Egy fájl tömbbe olvasása

Egy fájl alacsonyabb szinten byte-ok rendezett sorozata, emiatt az egyik legtermészetesebb művelet az, hogy ezeket a byte-okat egy *byte[]* tömbbe beolvassuk. A Java *InputStream* egy olyan *interface*, ami minden olyan objektum absztrakciója, amit byte-onként le lehet olvasni ezen a csatornán. Persze a fájl is ilyen, így a *toByteArray()* metódus ilyenből fogja olvasni azokat, egyetlen hívással betéve a példában mutatott *bytes* változó által referált objektumba.

```
File file = new File("/test/resources/Hello.txt");
try
{
   InputStream is = new FileInputStream(file);
   byte[] bytes = IOUtils.toByteArray(is);

   System.out.println("Byte_array_size:_" + bytes.➥
       length);
} catch (IOException e)
{
            e.printStackTrace();
}
```

InputStream alakítása Stringgé

A példában *is* egy olyan *InputStream*, amit egy fájlból származtattunk és olvasunk. A *toString()* metódus ezt egy hívással beteszi nekünk egy Stringbe, persze a kódolás megadása elengedhetetlen.

```
package example.commons.io;

import org.apache.commons.io.IOUtils;

import java.io.InputStream;
import java.io.FileInputStream;
import java.io.File;

public class InputStreamToString
{
    public static void main(String[] args) {
        InputStream is = new FileInputStream(new File(➥
            "data.txt"));
        String contents = IOUtils.toString(is, "UTF-8"➥
            );
        System.out.println(contents);
        IOUtils.closeQuietly(is);
    }
}
```

Comparator implementációk

A *java.util.Comparator* egy interface, amit utána sok algoritmus képes használni, ahol 2 objektumot össze kell hasonlítani. Két fájl esetén sokféle értelme van az összehasonlításnak, emiatt az IO csomag implementál néhányat. Előtte azonban nézzünk 2 kis példát! Az első a fájlok mérete alapján rendez:

```
List filesList = ... // Obtain a list of files from ➥
    somewhere
Collections.sort(filesList, SizeFileComparator.➥
    SIZE_COMPARATOR);
```

A második pedig a könyvtárak tartalmának összmérete alapján a könyvtárakat.

```
List directoriesList = ... // Obtain a list of ➥
    directories from somewhere
Collections.sort(directoriesList, SizeFileComparator.➥
    SIZE_SUMDIR_COMPARATOR);
```

A fájlok világában milyen összehasonlítás értelmes még? Nézzük meg ezt, hogy gondolja az IO csomag! A *LastModifiedFileComparator()* 2 fájlt az utolsó módosítása szerint hasonlít össze. A *NameFileComparator* pedig a fájlok nevei alapján állít fel sorrendet, természetesen a lexikografikus rendezést használva. A *PathFileComparator* is hasonló, de a fájl path-t veszi alapul. A *ExtensionFileComparator* a fájlnév kiterjesztése alapján teszi mindezt. A *CompositeFileComparator* azért jó, mert segítségével sorrendben egymás után több összehasonlítási szempont kombinálását is támogatja.

Séta a mappák között

A *DirectoryWalker<T>* egy nagyon hasznos eszköz, mert segítségével olyan algoritmusokat implementálhatunk, amik végigmennek egy könyvtárhierarchián és annak minden elemén csinálhatnak valami szükséges műveletet.

```
1  public class FileCleaner extends DirectoryWalker
2  {
3      public FileCleaner() {
4        super();
5      }
6
7      public List clean(File startDirectory) {
8        List results = new ArrayList();
9        walk(startDirectory, results);
10       return results;
11     }
12
13     protected boolean handleDirectory(File directory, int depth, Collection results) {
14       // delete svn directories and then skip
15       if (".svn".equals(directory.getName())) {
16         directory.delete();
17         return false;
18       } else {
19         return true;
20       }
21
22     }
23
24     protected void handleFile(File file, int depth, Collection results) {
25       // delete file and add to list of deleted
26       file.delete();
27       results.add(file);
28     }
29 }
```

A *FileCleaner* osztály a használatot mutatja be, azaz mindig kell készítenünk egy utód osztályt és abban kell megvalósítani a funkcionalitást. A *walk()* metódus indítja el a sétát a *startDirectory* ponttól, a bejárt helyeket pedig a *results* listában kapjuk vissza. A séta során könyvtárakkal, fájlokkal találkozik a walker, miközben ezekre visszahívja az itt implementált, az osztályra speciális célú *handleDirectory()* és *handleFile()* fájl metódusokat.

FileSystemUtils

A fájlrendszer egészére vonatkozó rutinok kerültek ide. A példa kiírja a lemez szabad helyének méretét:

```
package org.example.commons.io;

import org.apache.commons.io.FileSystemUtils;
import org.apache.commons.io.FileUtils;

import java.io.IOException;

public class DiskFreeSpace {
    public static void main(String[] args) {
        try {
            String path = "C:";
            long freeSpaceKB = FileSystemUtils.➥
                freeSpaceKb(path);
            long freeSpaceMB = freeSpaceKB / FileUtils➥
                .ONE_KB;
            long freeSpaceGB = freeSpaceKB / FileUtils➥
                .ONE_MB;

            System.out.println("Size␣of␣" + path + "␣=➥
                ␣" + freeSpaceKB + "␣KB");
            System.out.println("Size␣of␣" + path + "␣=➥
                ␣" + freeSpaceMB + "␣MB");
            System.out.println("Size␣of␣" + path + "␣=➥
                ␣" + freeSpaceGB + "␣GB");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Eseménykezelés

Az IO könyvtár is biztosít egy fájlmonitort, amivel a különféle fájlrendszer eseményeket észlelni lehet és azokra visszahívható metódusok tehetők, amik az események kezelői. A lenti *TestIOEvent* class ennek a programozását mutatja be egy nagyon egyszerű példán, ugyanis az *onXXX()* visszahívható metódusok implementáció csak kiírják azt, hogy egy esemény bekövetkezett.

```
1  package org.cs.io;
2
3  import java.io.File;
4  import org.apache.commons.io.monitor.FileAlterationListener;
5  import org.apache.commons.io.monitor.FileAlterationMonitor;
6  import org.apache.commons.io.monitor.FileAlterationObserver;
7
8  public class TestIOEvent
9  {
10   public static void main(String[] args) throws Exception
11   {
12     File directory = new File("/home/xxx");
13     FileAlterationObserver observer = new FileAlterationObserver(directory);
14     FileAlterationListener listener = new FileAlterationListener()
15     {
16
17       public void onStart(FileAlterationObserver fao)
18       {
19         System.out.println("onStart:" + fao.toString());
20       }
21
22       public void onDirectoryCreate(File file)
23       {
24         System.out.println("onDirectoryCreate:" + file.getName());
25       }
26
27       public void onDirectoryChange(File file)
28       {
29         System.out.println("onDirectoryChange:" + file.getName());
30       }
31
```

```
32        public void onDirectoryDelete(File file)
33        {
34          System.out.println("onDirectoryDelete:" + file.getName());
35        }
36
37        public void onFileCreate(File file)
38        {
39          System.out.println("onFileCreate:" + file.getName());
40          long GAP = 1000 * 10;
41          long lastTime = file.lastModified();
42          while ( file.lastModified() - lastTime <= GAP )
43          {
44            try
45            {
46              Thread.sleep(GAP);
47            } catch (InterruptedException e)
48            {
49              e.printStackTrace();
50            }
51            lastTime  = file.lastModified();
52          }
53        }
54
55        public void onFileChange(File file)
56        {
57          System.out.println("onFileChange:" + file.getName());
58        }
59
60        public void onFileDelete(File file)
61        {
62          System.out.println("onFileDelete:" + file.getName());
63        }
64
65        public void onStop(FileAlterationObserver fao)
66        {
67          System.out.println("onStop:" + fao.toString());
68        }
69      };
70
71      observer.addListener(listener);
72
73      FileAlterationMonitor monitor = new FileAlterationMonitor(1000);
74      monitor.addObserver(observer);
75      monitor.start();
76      Thread.sleep(1000 * 60);
77      monitor.stop();
78    }
79  }
```

Az *onFileCreate()* azért lett egy kicsit több sorból álló, mert azt is lekezeli, hogy egy fájl legalább 10 másodperc öreg legyen. Erre azért lehet szükség, mert az *onFileCreate()* bejelenti egy új fájl megjelenését, de azzal már nem törődik, hogy módosul-e még. A példa egyébként a */home/xxx* könyvtárban történteket figyeli, mert a monitort arra állítottuk. Az ismert observer design pattern valósítja meg a működést, amihez több listener-t is tehetnék, de a példában most csak 1 van.

Stream és Reader/Writer osztályok

Az IO csomag tartalmaz néhány közhasznú

- *InputStream*
- *Reader*

- *OutputStream* és
- *Writer*

implementációt olyan esetekre, amik gyakran előfordulnak a programozás során. Érdemes ezeket röviden áttekinteni, mert nagyon jól jönnek, amikor éppen szükségünk van valamelyikükre. Mi itt példaképpen csak egyet, a *LockableFileWriter* osztályt mutatjuk be röviden, ami ezért jó, mert úgy működik, mint az ismert *FileWriter*, de létrehoz egy lock fájlt. A legegyszerűbb konstruktora így néz ki:

```
public LockableFileWriter(File file) throws ➥
    IOException
```

Amikor meghívjuk a *close()* metódust, akkor a zároló fájl is törlődik.

Egyéb statikus osztályok

HexDump

Ez a kis osztály egy *byte[]* adatot hexadecimális formában küld el egy *OutputStream* csatornára. Egyetlen metódusa a *dump()*.

LineIterator

A *FileUtils* osztállyal való közös használatra lett tervezve, ahogy a következő példa is mutatja:

```
LineIterator it = FileUtils.lineIterator(file, "UTF-8"➥
    );
try {
   while (it.hasNext()) {
     String line = it.nextLine();
     // do something with line
   }
} finally {
   it.close();
}
```

A *lineIterator()* első paramétere egy *File* objektum.

EndianUtils

A különféle unicode rendszerek közötti átjárást segíti, amennyiben ilyenre szükségünk van, érdemes ezt az osztályt is megnézni.

IOCase

A különféle rendszerek (Unix, Windows) eltérő érzékenységűek a kis és nagybetű különbségekre. Ezeknek az univerzálisabb kezeléséhez ad segítséget ez az osztály.

Charsets

Szabványos *java.nio.charset.Charset* objektumokat képes szolgáltatni.

FileCleaningTracker

A fájlok törlésre jelölését és annak törlését menedzseli egy hozzárendelt handler objektum segítségével. Amikor ez a szemétgyűjtés során felszabadul, az így megjelölt fájl törlésre kerül.

FileDeleteStrategy

A fájlok törlése során alkalmazott stratégiát reprezentáló példányokat létrehozó osztály. Például mondhatjuk azt, hogy csak üres könyvtárat törölhetünk, de azt is, hogy erőltetjük a nem üres könyvtárak törlését.

FilenameUtils

Segítségével a fájlnevek és teljes útvonalak manipulálását könnyíthetjük meg. Windows esetén egy név ilyen: *C:\dev\project\file.txt*. A *C:* a prefix. A *dev\project* a path. Az osztály metódusai ezen név elemeknek a lekérdezését, illeszkedés vizsgálatát támogatják. Ilyen például a *String getPath(String filename)*, ami *a/b/c.txt* esetén a *a/b/* értéket adja vissza.

9. Az Active Directory elérése

Ebben a fejezetben Active Directory (AD) címtár adatbázis olvasását és írását tekintjük át. Régebben külön Java könyvtárak használatára volt szükség ehhez, de az Java 1.6 óta ezen funkcionalitás része a Java SDK-nak is. Szeretnénk megköszönni kollégánknak, Zilahy Zoltánnak, hogy felkeltette az érdeklődést ezen cikk megírására, amihez néhány jó ötletet is adott.

Az LDAP rövid áttekintése

Az *LDAP*[3] protokoll egy alkalmazás rétegben lévő API, ami lehetővé teszi az elosztott címtár szerverek elérését és karbantartását. Az LDAP az *IETF*[4] szervezet szabványa, aminek az utolsó változata az *RFC 4511*. Egy Directory Service bármilyen rekordot képes a címtár fa struktúrájában eltárolni, ezeknek a rekordoknak az összességét a címtár sémájának nevezzük. Egy címtár alapja lehet egy vállalati SSO megoldásnak.

Az LDAP rövid története

Talán elsőként a távközlési vállalatok ismerték fel a címtárak jelentőségét, gondoljunk csak a telefonkönyvekre. Kialakult a címtárak koncepciója, amit végül az *X.500* szabványban rögzítettek, amely folyamatot az *ITU*[5] menedzselte az 1980-as években. Ezt a *DAP* protokollal lehetett elérni, de hamarosan megjelent ennek alternatívájaként az LDAP, aminek jelenlegi LDAPv3 változata először 1997-ben volt publikálva.

Az LDAP áttekintése

Az LDAP kliens/szerver felépítésű, azaz a kliensek kapcsolódnak az LDAP szerverhez és utána hálózaton keresztül elérik annak szolgáltatásait. Érdemes megjegyezni, hogy a szerverek elterjedt alapértelmezett TCP portja a 389. A kliens különféle műveletek elvégzését kérheti a szervertől, amire az elvégzi a kért feladatot és valamilyen válasz üzenettel tudatja mindezt a hívó felé. Milyenek is lehetnek ezek a műveletek? Nézzük meg röviden:

- *Search* → Ez talán a leggyakoribb művelet típus, így az LDAP szerverek is erre optimalizáltak. A cél megtalálni egy directory bejegyzést (directory entry) és természetesen azt visszaadni a kliens részére. Ez lehet egy egész lista is, hiszen a keresési feltételnek több rekord is eleget tehet.
- *Compare* → Azt lehet ezzel a művelettel tesztelni, hogy egy megnevezett rekord tartalmazza-e megadott attribútum értéket.
- *Add* → Egy új rekord (entry) hozzáadása a címtárhoz.

[3] LDAP=Lightweight Directory Access Protocol
[4] Internet Engineering Task Force
[5] International Telecommunication Union

- *Delete* → Egy rekord (entry) törlése a címtárból.
- *Modify* → Egy rekord (entry) módosítása a címtárban.
- *DN Modify* → A *Distinguished Name* (*DN*, azaz megkülönböztető név) módosítása a címtárban. Ez lehet mozgatás vagy átnevezés.
- *Abandon* → Egy előző kliens műveleti kérés megszakítása.
- *Bind*→ Bejelentkezés, ami megadja a használni kívánt LDAP protokoll verziót is .
- *Unbind* → A kapcsolat lezárása.
- *StartTLS* → Az LDAPv3 biztonságos TLS hálózati rétegen való használatának kérése.

A directory struktúra kinézete

A directory rekordokra (továbbiakban: *Entry*) igazak a következő kijelentések:

- Egy Entry az attribútumok egy halmazából áll. Ezek a faszerkezet csomópontjai.
- Minden attribútum rendelkezik névvel és egy vagy több ehhez rendelt értékkel. Ezen attribútumok pontosan definiáltak, amit a címtár sémája (vagy sémái) rögzítenek.
- Mindegyik Entry rendelkezik egy egyedi azonosítóval, ami a már említett *DN*. A *DN* része a *Relative Distinguished Name* (*RDN*), ugyanis hozzá a directory faszerkezetének egy útvonalán (PATH) tudunk eljutni. Ez az útvonal a csomópont rekordok sorozata, aminek a végén – mint levél – az RDN áll.

Általában egy Entry mozgatható a fában és rendelkeznek egy *UUID* azonosítóval is, amit ettől teljesen független, de végső soron ez a bejegyzés teljes élettartama alatt biztosít az Entry számára egy egyedi azonosítót. Egy Entry megadható az un. *LDIF*[6] text formátumban is, ami így néz ki:

```
dn: cn=John Doe,dc=example,dc=com
cn: John Doe
givenName: John
sn: Doe
telephoneNumber: +1 888 555 6789
telephoneNumber: +1 888 555 1232
mail: john@example.com
manager: cn=Barbara Doe,dc=example,dc=com
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: person
objectClass: top
```

A fenti adatszerkezet csak példa, mert a bejegyzés típusától függően más és más attribútumok írhatják le a rekordot. A példában a *dn:* sor az Entry objektum megkülönböztető neve. Ez nem része természetesen az Entry adatainak, célja csak az, hogy megadja azt a path-t, ahogy az objektumhoz el lehet jutni a címtár fában. A *cn:* az Entry RDN neve (*cn*=common name). A

[6]LDIF=LDAP Data Interchange Format

dc=example,dc=com pedig az objektumunk szülő Entry-je, ahol a *dc* jelentése *Domain Component*. Ez egy tároló is egyben, hiszen az ilyen típusú objektumok halmaza közelíthető meg rajta keresztül. Láthatjuk azt is, hogy az attribútum nevek valamilyen emlékeztető nevek, esetünkben néhány példa:

- *givenName* → keresztnév
- *sn* → vezetéknév (surname)

Egy LDAP szerver mindig egy címtár részfát is jelent, a fenti esetben ez a *dc=example,dc=com* részfa. Ez egy kezdőpontja minden további bejegyzésnek. Egy Entry lehet referencia egy másik szerverre is, például az *ou* (organisational unit=szervezeti egység) Entry által jelképezett részfa lehet egy másik szerveren is egy ilyen név esetén: *ou=hr,dc=example,dc=com*.

Műveletek a címtárban

Ebben a pontban áttekintjük a címtáron végezhető LDAP műveleteket. A legtöbb program igénye a *Search and Compare* műveletekre korlátozódik, mert nem akarják a címtár tartalmát megváltoztatni. Itt meg kell tanulnunk néhány alapfogalmat, amit az LDAP kliens programok lépten-nyomon használnak.

9.1. Meghatározás (baseObject). *A bázis objektum (Entry) neve, ami akár a root (azaz a fa gyökére) is lehet. A keresés innen fog kezdődni a fában, amiatt* base DN *néven is szoktuk emlegetni.*

9.2. Meghatározás (scope). *Azt határozza meg, hogy a* baseObject *alatt elindított keresés milyen körben történjen. Ez lehet maga a* baseObject. *A másik eset a* singleLevel, *amikor közvetlenül a* baseObject *alatt keres, míg a legáltalánosabb lehetőség a* wholeSubtree, *ugyanis ekkor a teljes részfa átvizsgálásra kerül.*

9.3. Meghatározás (filter). *A filter egy keresési feltétel, ami a megadott scope-on belüli objektumokra kerül leellenőrzésre. Tekintsünk egy példát:*

```
(&(objectClass=person)(|(givenName=Imre)(mail=inyiri*)))
```

Ez a szűrő csak azon objektumokat fogja visszaadni, amik person *típusúak, azaz személyek. A filter egy & jellel kezdődik, ez utal arra, hogy itt több szempont* ÉS *kapcsolata lesz. Ez volt az első, nézzük a másodikat. A pipe (|) jel a* VAGY *kapcsolat művelete, azaz az* ÉS *második operandusa egy* VAGY *szerkezet:*

- *A* givenName *Imre VAGY*
- *a* mail *jellemző inyiri*-ra illeszkedik.*

Ezenfelül létezik még a tagadás, aminek „!” a jele. Fontos, hogy a filter kifejezés alapesetben kisbetű-nagybetű érzékeny. Egy szűrőben használhatjuk a =, <= és >= relációs jeleket is.

Amennyiben valakit részletesebben is érdekel az LDAP szűrő kifejezések, úgy ajánljuk ezt a helyet tanulmányozás céljából: `http://www.ldapguru.info/ldap/mastering-ldap-search-filters.html` . A *derefAliases* kereső kulcsszó a referenciák követésével kapcsolatos. Ez az az eset, amikor az egyik Entry objektum hivatkozik a másikra. A keresés *attributes* kulcsszó azokat a jellemzőket adja meg, amiket vissza kell adni az eredmény objektumokból. A *sizeLimit* és *timeLimit* a keresés azon feltételeit specifikálja, hogy maximum mennyi darab Entry objektum érkezhet vissza, illetve a keresés meddig folytatható. A *typesOnly* megadása esetén csak a jellemzők típusa adódik vissza, értéke nem. A továbbiakban nézzük meg az egyéb LDAP műveleteket röviden:

- *Add*: Az alábbi LDIF formátumú objektumot akarjuk beszúrni:

```
dn: uid=user,ou=people,dc=example,dc=com
changetype: add
objectClass: top
objectClass: person
uid: user
sn: last-name
cn: common-name
userPassword: password
```

  Ekkor a beszúrandó *uid=user,ou=people,dc=example,dc=com* objektum nem létezhet, de a szülő *ou=people,dc=example,dc=com* objektumnak léteznie kell.

- *Bind*: Az LDAP címtárhoz való kapcsolódás, aminek a hitelesítés (authentication) is része. Tekintettel arra, hogy egy címtárban minden objektumnak egy egyedi *DN* neve van, ezért az lesz a hagyományos értelemben vett username, ami mellett egy jelszót is meg kell adnunk. Hitelesítés után a létrejött session bizonyos attribútumokat is tartalmaz, ezért is nevezzük ezt a folyamatot bind-nak.

- *Delete*: Kitöröl egy objektumot.

- *Modify*: Egy objektum módosítása.

- *Modify DN*: Egy objektum mozgatása vagy átnevezése.

- *Abandon*: Az éppen futó művelet megszakítása.

- *Unbind*: Lekapcsolódás az LDAP szerverről.

Az LDAP URL általános kinézete

Az LDAP URL általános alakja a következő, de ezek egy része opcionális:

```
ldap://host:port/DN?attributes?scope?filter?extensions
```

Az URL egyes részeinek jelentése a következő:

- *host:port*: Az LDAP szerviz itt érhető el.

- *DN*: a DN név

- *attributes*: vesszővel elválasztva itt adhatjuk meg, hogy a válaszban mely attribútumokat szeretnénk visszakapni.
- *scope*: A már ismertetett scope adható itt meg.
- *filter*: Az bemutatott filter kifejezés helye.
- *extensions*: Az LDAP kiterjesztések megadási helye.

Az LDAP Java API

A *javax.naming.directory* csomag tartalmazza azokat az osztályokat és interface-eket, amik egy LDAP szerverrel (így az AD is) való együttműködéshez szükséges. Létezik még sok olyan külső *jar* is, ami ugyanezt a feladatot látja el, de azokat csak szükség esetén érdemes használni. Szeretnénk felhívni a figyelmet az *Apache Directory Projectre* (webhelye: `http://directory.apache.org/`), aminek 3 fontos szoftver eleme van:

- *ApacheDS*: Egy teljes értékű címtár szerver.
- *Apache Directory Studio*: Egy Eclipse alapú LDAP kliens.
- *Apache Directory LDAP API*: Egy Java könyvtár, ami implementálja az LDAP API-t. Ez is használható lenne a JDK beépített API-ja helyett.

Első példa egyszerű lekérdezésre

Az alábbiakban látható példánk forráskódja:

```
1  import java.util.Hashtable;
2  import javax.naming.ldap.*;
3  import javax.naming.directory.*;
4  import javax.naming.*;
5
6
7  public class ADTest
8  {
9   public static void main (String[] args)
10  {
11
12    Hashtable env = new Hashtable();
13
14    String userName = "CN=inyiri,CN=CegUsers,DC=ceg,DC=com";
15    String userPassword = "XXXXXXX";
16    String ldapURL = "ldap://mydc.ceg.com:389";
17
18    env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
19
20    //set security credentials
21    env.put(Context.SECURITY_AUTHENTICATION,"simple");
22    env.put(Context.SECURITY_PRINCIPAL,userName);
23    env.put(Context.SECURITY_CREDENTIALS,adminPassword);
24
25    //specify use of ssl
26    env.put(Context.SECURITY_PROTOCOL,"ssl");
27
```

```
28    //connect to my domain controller
29    env.put(Context.PROVIDER_URL,ldapURL);
30
31    try {
32
33     // Create the initial directory context
34     DirContext ctx = new InitialLdapContext(env,null);
35
36     //Create the search controls
37     SearchControls searchCtls = new SearchControls();
38
39     //Specify the attributes to return
40     String returnedAtts[]={"sn","mail","cn","telephonenumber"};
41     searchCtls.setReturningAttributes(returnedAtts);
42
43     //Specify the search scope
44     searchCtls.setSearchScope(SearchControls.SUBTREE_SCOPE);
45
46     //specify the LDAP search filter
47     String searchFilter = "(&(objectClass=user)(mail=*))";
48
49     //Specify the Base for the search
50     String searchBase = "DC=ceg,DC=com";
51
52     //initialize counter to total the results
53     int totalResults = 0;
54
55     // Search for objects using the filter
56     NamingEnumeration answer = ctx.search(searchBase, searchFilter, searchCtls);
57
58     //Loop through the search results
59     while (answer.hasMoreElements()) {
60          SearchResult sr = (SearchResult)answer.next();
61
62      totalResults++;
63
64      System.out.println(">>>" + sr.getName());
65
66      // Print out some of the attributes, catch the exception if the attributes have no values
67      Attributes attrs = sr.getAttributes();
68      if (attrs != null) {
69       try {
70         System.out.println("   surname: " + attrs.get("sn").get());
71         System.out.println("   firstname: " + attrs.get("givenName").get());
72         System.out.println("   mail: " + attrs.get("mail").get());
73       }
74       catch (NullPointerException e) {
75         System.out.println("Errors listing attributes: " + e);
76       }
77      }
78     }
79
80      System.out.println("Total results: " + totalResults);
81      ctx.close();
82    }
83    catch (NamingException e) {
84      System.err.println("Problem searching directory: " + e);
85    }
86   }
87  }
```

Tekintsük át a fontosabb részek jelentését! A 14-16 sorokban egy-egy stringbe mentjük a bejelentkezéshez szükséges felhasználói nevet, jelszót és az LDAP connection stringet. Az *env* egy gyűjteménye azoknak a paramétereknek, amik a BIND igényel, azaz a kapcsolódáshoz szükséges.

Mindez a 18-29 sorok között van megadva, a program megjegyzéseiből kiolvashatóak a szerepek. A 34. sor *ctx* változója egy kontextust kér és tárol az LDAP szerver felé. A 40-41 sorok *returnedAtts* tömbje és annak beállítása fogja majd specifikálni, hogy mely jellemzőkre van szükségünk a keresés eredményeképpen kapott válaszban. A 44. sorban egy keresési vezérlési szabályt deklarálunk, azaz azt kérjük, hogy majd a *base DN* alatti teljes részfában történjen a keresés. A 47. sor egy kereső filter kifejezés, ami egy *ÉS* kapcsolatban azt mondja, hogy kérünk minden *user* típusú objektumot, ahol a *mail* bármi. Az 50. sor szöveges változója pedig a keresés base DN-jét tárolja. Az 56. sorban lefuttatjuk magát a keresést, aminek a válasza az *answer* változó által referált *NamingEnumeration* típusú objektumba kerül. Az 59-78 sorok közötti ciklus a választ dolgozza fel, azaz jelen esetben kiírja a képernyőre a kapott értékeket. Végiglépdelünk az iteráción, ahol az egyes iterálandó elemek *SearchResult* típusúak. Az értékek lekérése egyszerűen megérthető a kód 64-72 sorok közötti részéből. Befejezésként fontos lépés a 81. sor lezáró művelete.

Az LDAPManager példakód

A most következő példakód *Brett McLaughlin*től származik, a Java LDAP API sok vonatkozását megismerhetjük belőle. A kódot teljes egészében közöljük, de kiemeljük, hogy ez a www.oreilly.com terméke és szabadon felhasználható. Az egyes alpontokban metódusonként mutatjuk be az *LDAPManager* lehetőségeit. A kód elején konstansokat adtunk meg, amik a következő dolgokat specifikálják:

- A userek tárolása melyik DN alatt lesz (*USERS_OU*)
- A csoportok tárolása melyik DN alatt lesz (*GROUPS_OU*)
- A jogok tárolásának base objektuma (*PERMISSIONS_OU*)
- Az LDAP szerver alapértelmezett portja (*DEFAULT_PORT*)

A getInstance() gyártó és néhány kisegítő metódus

Az *LDAPManager* alapstruktúráját – aminek további részleteit a későbbi pontok tárgyalják – a következő kód mutatja:

```
1  package test;
2
3  import java.util.HashMap;
4  import java.util.Properties;
5  import java.util.LinkedList;
6  import java.util.List;
7  import java.util.Map;
8  import javax.naming.Context;
9  import javax.naming.NameNotFoundException;
10 import javax.naming.NamingEnumeration;
11 import javax.naming.NamingException;
12 import javax.naming.directory.Attribute;
13 import javax.naming.directory.AttributeInUseException;
14 import javax.naming.directory.Attributes;
15 import javax.naming.directory.BasicAttribute;
16 import javax.naming.directory.BasicAttributes;
17 import javax.naming.directory.DirContext;
```

```
18 import javax.naming.directory.InitialDirContext;
19 import javax.naming.directory.ModificationItem;
20 import javax.naming.directory.NoSuchAttributeException;
21 import javax.naming.directory.SearchControls;
22 import javax.naming.directory.SearchResult;
23
24 public class LDAPManager {
25
26     /** The OU (organizational unit) to add users to */
27     private static final String USERS_OU =
28         "ou=People,o=forethought.com";
29
30     /** The OU (organizational unit) to add groups to */
31     private static final String GROUPS_OU =
32         "ou=Groups,o=forethought.com";
33
34     /** The OU (organizational unit) to add permissions to */
35     private static final String PERMISSIONS_OU =
36         "ou=Permissions,o=forethought.com";
37
38     /** The default LDAP port */
39     private static final int DEFAULT_PORT = 389;
40
41     /** The LDAPManager instance object */
42     private static Map instances = new HashMap();
43     /** The connection, through a <code>DirContext</code>, to LDAP */
44     private DirContext context;
45
46     /** The hostname connected to */
47     private String hostname;
48
49     /** The port connected to */
50     private int port;
51
52     protected LDAPManager(String hostname, int port,
53                           String username, String password)
54         throws NamingException {
55
56         context = getInitialContext(hostname, port, username, password);
57
58         // Only save data if we got connected
59         this.hostname = hostname;
60         this.port = port;
61     }
62
63     public static LDAPManager getInstance(String hostname,
64                                           int port,
65                                           String username,
66                                           String password)
67         throws NamingException {
68
69         // Construct the key for the supplied information
70         String key = new StringBuffer()
71             .append(hostname)
72             .append(":")
73             .append(port)
74             .append("|")
75             .append((username == null ? "" : username))
76             .append("|")
77             .append((password == null ? "" : password))
78             .toString();
79
80
81         if (!instances.containsKey(key)) {
82             synchronized (LDAPManager.class) {
```

```
83                  if (!instances.containsKey(key)) {
84                      LDAPManager instance =
85                          new LDAPManager(hostname, port,
86                                          username, password);
87                      instances.put(key, instance);
88                      return instance;
89                  }
90              }
91          }
92
93          return (LDAPManager)instances.get(key);
94      }
95
96      public static LDAPManager getInstance(String hostname, int port)
97          throws NamingException {
98
99          return getInstance(hostname, port, null, null);
100     }
101
102     public static LDAPManager getInstance(String hostname)
103         throws NamingException {
104
105         return getInstance(hostname, DEFAULT_PORT, null, null);
106     }
107 ...
108     private String getUserDN(String username) {
109         return new StringBuffer()
110                 .append("uid=")
111                 .append(username)
112                 .append(",")
113                 .append(USERS_OU)
114                 .toString();
115     }
116
117     private String getUserUID(String userDN) {
118         int start = userDN.indexOf("=");
119         int end = userDN.indexOf(",");
120
121         if (end == -1) {
122             end = userDN.length();
123         }
124
125         return userDN.substring(start+1, end);
126     }
127
128     private String getGroupDN(String name) {
129         return new StringBuffer()
130                 .append("cn=")
131                 .append(name)
132                 .append(",")
133                 .append(GROUPS_OU)
134                 .toString();
135     }
136
137     private String getGroupCN(String groupDN) {
138         int start = groupDN.indexOf("=");
139         int end = groupDN.indexOf(",");
140
141         if (end == -1) {
142             end = groupDN.length();
143         }
144
145         return groupDN.substring(start+1, end);
146     }
147
```

```
148     private String getPermissionDN(String name) {
149         return new StringBuffer()
150                 .append("cn=")
151                 .append(name)
152                 .append(",")
153                 .append(PERMISSIONS_OU)
154                 .toString();
155     }
156
157     private String getPermissionCN(String permissionDN) {
158         int start = permissionDN.indexOf("=");
159         int end = permissionDN.indexOf(",");
160
161         if (end == -1) {
162             end = permissionDN.length();
163         }
164
165         return permissionDN.substring(start+1, end);
166     }
167
168     private DirContext getInitialContext(String hostname, int port,
169                                          String username, String password)
170         throws NamingException {
171
172         String providerURL =
173             new StringBuffer("ldap://")
174                 .append(hostname)
175                 .append(":")
176                 .append(port)
177                 .toString();
178
179         Properties props = new Properties();
180         props.put(Context.INITIAL_CONTEXT_FACTORY,
181                   "com.sun.jndi.ldap.LdapCtxFactory");
182         props.put(Context.PROVIDER_URL, providerURL);
183
184         if ((username != null) && (!username.equals(""))) {
185             props.put(Context.SECURITY_AUTHENTICATION, "simple");
186             props.put(Context.SECURITY_PRINCIPAL, username);
187             props.put(Context.SECURITY_CREDENTIALS,
188                 ((password == null) ? "" : password));
189         }
190
191         return new InitialDirContext(props);
192     }
193 ...
194 } // end class
```

Az első, amit talán észreveszünk, hogy az *LDAPManager* konstruktora *protected*, emögött az a cél húzódik meg, ebből egy példányt valamelyik *getInstance()* gyártómetódussal hozzunk létre, amik persze belül ezzel hoznak létre új példányt. A *getInstance()* legteljesebb paraméterezésű változata ezeket az argumentumokat képes átvenni:

- hostname, port
- username, password

A 70-78 sorok között egy string kerül összeállításra erről a 4 darab paraméterről. Az *instances* egy *HashMap*, amit még a 42. sorban hoztunk létre. Kliensünk képes több LDAP szerver session-t egyszerre fenntartani, de ha már van ilyen, akkor azt használja. Emiatt kellett az előző

string is, ugyanis ez azonosítja ezeket a kapcsolatokat a *Map*-ben. Ennek megfelelően működik a 81-91 sorok közötti kód. Amennyiben nincs a megadott paraméterekre illeszkedő session az *instances* tárolóban, úgy azt a konstruktorral létrehozza, majd beteszi oda. A végeredmény mindig az lesz, hogy egy *LDAPManager* példányt kapunk. A használt konstruktor fő feladata a *context* adattag inicializálása (azaz a BIND művelet), ami az 56. sorban történik meg. Itt érdemes gyorsan átnézni a privát metódusokat is, mert azokat a későbbiekben majd használni fogjuk. Az első és legfontosabb ilyen a *getInitialContext()*, ugyanis itt van megvalósítva a konstruktor lényegi része. A metódus egy *DirContext* objektumot ad vissza. A 172-177 sorokban felépítjük az LDAP URL (*ldap://...*) stringet, amit a *providerURL* változóhoz rendeltünk. Az összes kapcsolódáshoz szükséges paraméter egy *props* nevű *Properties* változóba kerül, amivel aztán előállítjuk a visszaadandó *InitialDirContext* objektumot. Mindezzel azt értük el, hogy van egy új session az LDAP (vagy AD) szerver felé. A *getUserDN()* metódus feladata az, hogy egy *uid=**user**,ou=People,o=forethought.com* alakú stringet szolgáltasson, ahol a *user* egy bemenő paraméter. Ez a felhasználói *DN*. A *getGroupDN()* szerepe hasonló, neki egy csoport neve alapján a *cn=**group**,ou=Groups,o=forethought.com* alakú szöveget kell szolgáltatnia. A *getUserUID()* a fenti *uid*-del kezdődő *DN* névből kiveszi az első „=” és „,” jelek közötti részt, azaz a user nevét. A *getGroupCN()* hasonlóan ugyanezt teszi, de a csoport *cn*, azaz *RDN* nevét adja vissza. A *getPermissionDN()* egy *cn=***permission***,ou=Permissions,o=forethought.com* alakú stringet gyárt le, ami egy-egy jog *DN* neve. A *getPermissionCN()* pedig ennek – hasonlóan a fentiekhez – a jog *RDN* nevét vágja ki.

Az addUser() metódus

A korábbiakban már említettük, hogy az LDAP címtár egy séma alapján írja le a saját adatszerkezetét, amiben az Entry objektumok is ott vannak. Ez általában úgy van megvalósítva, hogy több sémafájl együtt írja le a teljes sémát, amit utána a szerver indulásakor ismer és használ. Itt van például a *core.schema* fájl egy kicsi részlete:

```
1 // core.schema
2 ...
3 objectclass ( 2.5.6.6 NAME 'person' SUP top STRUCTURAL
4   MUST ( sn $ cn )
5   MAY ( userPassword $ telephoneNumber $ seeAlso $ description ) )
6 ...
```

Láthatjuk, hogy itt most a *person* típust megadó sémarészletet látjuk. Már itt az elején érdemes észrevenni, hogy a megadott attribútumok egy része kötelező (MUST), másik része pedig csak lehetőség (MAY). Ennyi előzmény után már jobban érthető lesz az *addUser()* metódus működése, aminek az a feladata, hogy egy új DN Entry-t, pontosabban egy felhasználót szúrjon be a címtárba. Tekintettel arra, hogy 4 darab paraméterünk van, ezek lesznek majd a bejegyzésben szereplő jellemzők. A 9. sorban egy *container* változót készítünk a jellemzők tárolására, ez standard módon az *Attributes* típus lesz. Az *objClasses* egy másik ugyanilyen konténer, de ennek kifejezett célja az lesz, hogy a sémának megfelelő attribútumok majd benne legyenek (12. sor). A 13-16 sorok között alakítjuk ki a végső sémát a *top*, *person*, *organizationalPerson* és *inetOrgPerson* séma osztályok alapján. Mindegyik hozzáadja a saját maga által leírt mezőket, a *person*-t például bemutattuk. A 18-22 sorok között csak egy olyan string került előállításra, ami a személy teljes

nevét tartalmazza. A 23-30 sorok között létrehozzuk a feltöltendő értékeket reprezentáló 5 darab *Attribute* objektumot. A 33-38 sorok között felépítjük a *container* objektumot, előbb a tárolót, aztán hozzáadjuk a jellemzőket. A 41. sor az így előkészített *container* tartalmát elhelyezi a címtárba oda, amit a már ismert *getUserDN()* metódus kijelöl számára.

```
1  ...
2  public class LDAPManager {
3  ...
4      public void addUser(String username, String firstName,
5                          String lastName, String password)
6          throws NamingException {
7
8          // Create a container set of attributes
9          Attributes container = new BasicAttributes();
10
11         // Create the objectclass to add
12         Attribute objClasses = new BasicAttribute("objectClass");
13         objClasses.add("top");
14         objClasses.add("person");
15         objClasses.add("organizationalPerson");
16         objClasses.add("inetOrgPerson");
17
18         // Assign the username, first name, and last name
19         String cnValue = new StringBuffer(firstName)
20             .append(" ")
21             .append(lastName)
22             .toString();
23         Attribute cn = new BasicAttribute("cn", cnValue);
24         Attribute givenName = new BasicAttribute("givenName", firstName);
25         Attribute sn = new BasicAttribute("sn", lastName);
26         Attribute uid = new BasicAttribute("uid", username);
27
28         // Add password
29         Attribute userPassword =
30             new BasicAttribute("userpassword", password);
31
32         // Add these to the container
33         container.put(objClasses);
34         container.put(cn);
35         container.put(sn);
36         container.put(givenName);
37         container.put(uid);
38         container.put(userPassword);
39
40         // Create the entry
41         context.createSubcontext(getUserDN(username), container);
42     }
43 ...
44 } // end class
```

A deleteUser() metódus

Ezen metódus feladata az, hogy kitörölje a paraméterben megadott felhasználót a címtárból, aminek DN nevét szintén a *getUserDN()* segítségével állítja elő. Maga a megvalósítás nem túl nehéz, mindössze egyetlen sor, mert semmi újat nem kell előállítani, a megoldás mindössze a *context* objektumra meghívott *destroySubcontext()* metódus, ami értelemszerűen a törlendő DN-t kapja.

```
1  ...
2  public class LDAPManager {
3  ...
```

```
4       public void deleteUser(String username) throws NamingException {
5           try {
6               context.destroySubcontext(getUserDN(username));
7           } catch (NameNotFoundException e) {
8               // If the user is not found, ignore the error
9           }
10      }
11  ...
12  } // end class
```

Az isValidUser() metódus

Ezzel a logikai metódussal le tudjuk azt ellenőrizni, hogy egy megadott user/password birtokában be tudunk-e lépni a címtárba. Ehhez mindössze annyit ellenőriz le, hogy a *context* objektumot meg tudjuk-e szerezni.

```
1   ...
2   public class LDAPManager {
3   ...
4   public boolean isValidUser(String username, String password)
5           throws NameNotFoundException {
6           try {
7               DirContext context =
8                   getInitialContext(hostname, port, getUserDN(username),
9                                     password);
10              return true;
11          } catch (javax.naming.NameNotFoundException e) {
12              throw new NameNotFoundException(username);
13          } catch (NamingException e) {
14              // Any other error indicates couldn't log user in
15              return false;
16          }
17      }
18  ...
19  } // end class
```

Az addGroup() metódus

Az *addGroup()* egy új csoportot helyez el a *getGroupDN()* által visszaadott helyre. Működése az *addUser()*-hez hasonló, de a használt *objClass*-ok egy része természetesen más, lévén csoport leírásról van szó.

```
1   ...
2   public class LDAPManager {
3   ...
4       public void addGroup(String name, String description)
5           throws NamingException {
6
7           // Create a container set of attributes
8           Attributes container = new BasicAttributes();
9
10          // Create the objectclass to add
11          Attribute objClasses = new BasicAttribute("objectClass");
12          objClasses.add("top");
13          objClasses.add("groupOfUniqueNames");
14          objClasses.add("groupOfForethoughtNames");
15
16          // Assign the name and description to the group
```

```
17          Attribute cn = new BasicAttribute("cn", name);
18          Attribute desc = new BasicAttribute("description", description);
19
20          // Add these to the container
21          container.put(objClasses);
22          container.put(cn);
23          container.put(desc);
24
25          // Create the entry
26          context.createSubcontext(getGroupDN(name), container);
27      }
28  ...
29  } // end class
```

A deleteGroup() metódus

A csoport törlését végző *deleteGroup()* logikája teljesen hasonló, mint a *deleteUser()* volt, de a törlés szülőhelyét természetszerűleg a *getGroupDN()* által visszaadott kontextus string adja meg.

```
1   ...
2   public class LDAPManager {
3   ...
4   public void deleteGroup(String name) throws NamingException {
5           try {
6               context.destroySubcontext(getGroupDN(name));
7           } catch (NameNotFoundException e) {
8               // If the group is not found, ignore the error
9           }
10      }
11  ...
12  } // end class
```

Az addPermission() metódus

Egy jogelem hozzáadási logikája sem különbözik a többitől, de itt az erre kitalált *forethought-Permission* sémarész objektumot is használjuk, illetve a beszúrás helyét a *getPermissionDN()* kalkulálja ki nekünk.

```
1   ...
2   public class LDAPManager {
3   ...
4     public void addPermission(String name, String description)
5           throws NamingException {
6
7           // Create a container set of attributes
8           Attributes container = new BasicAttributes();
9
10          // Create the objectclass to add
11          Attribute objClasses = new BasicAttribute("objectClass");
12          objClasses.add("top");
13          objClasses.add("forethoughtPermission");
14
15          // Assign the name and description to the group
16          Attribute cn = new BasicAttribute("cn", name);
17          Attribute desc = new BasicAttribute("description", description);
18
19          // Add these to the container
20          container.put(objClasses);
21          container.put(cn);
```

```
22          container.put(desc);
23
24          // Create the entry
25          context.createSubcontext(getPermissionDN(name), container);
26      }
27  ...
28  } // end class
```

A deletePermission() metódus

A *deletePermission()* a már ismer módszer szerint töröl egy jogelemet.

```
1   ...
2   public class LDAPManager {
3   ...
4       public void deletePermission(String name) throws NamingException {
5           try {
6               context.destroySubcontext(getPermissionDN(name));
7           } catch (NameNotFoundException e) {
8               // If the permission is not found, ignore the error
9           }
10      }
11  ...
12  } // end class
```

Az assignUser() metódus

Az *assignUser()* egy felhasználót betesz a specifikált csoportba. Ehhez egy 1 elemű *ModificationItem* tömb szükséges. A 10-12 sorokban egy *mod* attribútumot hozunk létre, ahol a *uniqueMember* jellemzőhöz a felhasználónk DN-jét rendeljük hozzá. A 13-14 sorok ezt azt 1 módosítási művelet kérést helyezik el a *mods* tömbbe, majd a 15. sorban végrehajtatjuk a kérést a *getGroupDN()* helyen lévő csoportra.

```
1   ...
2   public class LDAPManager {
3   ...
4       public void assignUser(String username, String groupName)
5           throws NamingException {
6
7           try {
8               ModificationItem[] mods = new ModificationItem[1];
9
10              Attribute mod =
11                  new BasicAttribute("uniqueMember",
12                                     getUserDN(username));
13              mods[0] =
14                  new ModificationItem(DirContext.ADD_ATTRIBUTE, mod);
15              context.modifyAttributes(getGroupDN(groupName), mods);
16          } catch (AttributeInUseException e) {
17              // If user is already added, ignore exception
18          }
19      }
20  ...
21  } // end class
```

A removeUser() metódus

A *removeUser()* az előző művelet fordítottja, azaz eltávolítja a felhasználót a csoportból.

```
1  ...
2  public class LDAPManager {
3  ...
4      public void removeUser(String username, String groupName)
5          throws NamingException {
6
7          try {
8              ModificationItem[] mods = new ModificationItem[1];
9
10             Attribute mod =
11                 new BasicAttribute("uniqueMember",
12                                    getUserDN(username));
13             mods[0] =
14                 new ModificationItem(DirContext.REMOVE_ATTRIBUTE, mod);
15             context.modifyAttributes(getGroupDN(groupName), mods);
16         } catch (NoSuchAttributeException e) {
17             // If user is not assigned, ignore the error
18         }
19     }
20 ...
21 } // end class
```

A userInGroup() metódus

Ez a *userInGroup()* logikai metódus azért fontos, mert egy ilyen vizsgálat lényeges része az alkalmazás jogosultságkezelési rendszerének. A csoport részéről továbbra is a *uniqueMember* mező lesz a kulcs attribútum, amit a *searchAttributes* tömbben el is helyeztünk. A 11-13 sorokban a csoport DN és ezen tömb alapján megalkotjuk az *attributes* változót. A 15. sorban látható módon lekérhetjük a *memberAtts* által mutatott objektumot, amit végig tudunk járni és fel tudunk dolgozni, mint *NamingEnumeration* objektum. Amennyiben ezek között van a mi user-ünk, úgy benne van a csoportban.

```
1  ...
2  public class LDAPManager {
3  ...
4      public boolean userInGroup(String username, String groupName)
5          throws NamingException {
6
7          // Set up attributes to search for
8          String[] searchAttributes = new String[1];
9          searchAttributes[0] = "uniqueMember";
10
11         Attributes attributes =
12             context.getAttributes(getGroupDN(groupName),
13                                   searchAttributes);
14         if (attributes != null) {
15             Attribute memberAtts = attributes.get("uniqueMember");
16             if (memberAtts != null) {
17                 for (NamingEnumeration vals = memberAtts.getAll();
18                      vals.hasMoreElements();
19                      ) {
20                     if (username.equalsIgnoreCase(
21                         getUserUID((String)vals.nextElement()))) {
22                         return true;
23                     }
24                 }
```

```
25              }
26          }
27
28          return false;
29      }
30  ...
31  } // end class
```

A getMembers() metódus

A *getMembers()* működése hasonlít a *userInGroup()*-hoz, de ahelyett, hogy vizsgálná a user tartalmazását, visszaadja a csoport összes felhasználóját.

```
1   ...
2   public class LDAPManager {
3   ...
4       public List getMembers(String groupName) throws NamingException {
5           List members = new LinkedList();
6
7           // Set up attributes to search for
8           String[] searchAttributes = new String[1];
9           searchAttributes[0] = "uniqueMember";
10
11          Attributes attributes =
12              context.getAttributes(getGroupDN(groupName),
13                                    searchAttributes);
14          if (attributes != null) {
15              Attribute memberAtts = attributes.get("uniqueMember");
16              if (memberAtts != null) {
17                  for (NamingEnumeration vals = memberAtts.getAll();
18                       vals.hasMoreElements();
19                       members.add(
20                           getUserUID((String)vals.nextElement()))) ;
21              }
22          }
23
24          return members;
25      }
26  ...
27  } // end class
```

A getGroups() metódus

A *getGroups()* visszaadja a megadott user összes csoportjának a nevét. Ez lehetővé teszi például annak a technikának a megvalósítását, hogy hitelesítés után egy olyan *User* objektum kerüljön a session-re, ami már a csoportjainak listáját is tartalmazza. A megoldás kulcsa egy jól megfogalmazott filter, ami azt kéri, hogy csak azok az objektumok kerüljenek legyűjtésre, ahol az *objectClass=groupOfForethoughtNames* ÉS a *uniqueMember* értéke pedig a mi felhasználói nevünk. A 18-19 sorok mutatják, hogy most 1 szintű scope lesz használva, azaz nem kell keresnünk a részfában. Ennek az az oka, hogy a csoportok közvetlenül a *GROUPS_OU* alatt vannak.

```
1   ...
2   public class LDAPManager {
3   ...
4       public List getGroups(String username) throws NamingException {
5           List groups = new LinkedList();
6
```

```
7           // Set up criteria to search on
8           String filter = new StringBuffer()
9               .append("(&")
10              .append("(objectClass=groupOfForethoughtNames)")
11              .append("(uniqueMember=")
12              .append(getUserDN(username))
13              .append(")")
14              .append(")")
15              .toString();
16
17          // Set up search constraints
18          SearchControls cons = new SearchControls();
19          cons.setSearchScope(SearchControls.ONELEVEL_SCOPE);
20
21          NamingEnumeration results =
22              context.search(GROUPS_OU, filter, cons);
23
24          while (results.hasMore()) {
25              SearchResult result = (SearchResult)results.next();
26              groups.add(getGroupCN(result.getName()));
27          }
28
29          return groups;
30      }
31  ...
32  } // end class
```

Az assignPermission() metódus

Az *assignPermission()* célja, hogy egy jogelemet egy csoporthoz rendeljen. A megvalósítás módja analóg a *assignUser()* metódussal.

```
1   ...
2   public class LDAPManager {
3   ...
4       public void assignPermission(String groupName, String permissionName)
5           throws NamingException {
6
7           try {
8               ModificationItem[] mods = new ModificationItem[1];
9
10              Attribute mod =
11                  new BasicAttribute("uniquePermission",
12                                     getPermissionDN(permissionName));
13              mods[0] =
14                  new ModificationItem(DirContext.ADD_ATTRIBUTE, mod);
15              context.modifyAttributes(getGroupDN(groupName), mods);
16          } catch (AttributeInUseException e) {
17              // Ignore the attribute if it is already assigned
18          }
19      }
20  ...
21  } // end class
```

A revokePermission() metódus

Egy jogelemet visszavesz a megadott csoportból, aminek működése olyan, mint a *removeUser()* volt.

```
...
public class LDAPManager {
...
    public void revokePermission(String groupName, String permissionName)
        throws NamingException {

        try {
            ModificationItem[] mods = new ModificationItem[1];

            Attribute mod =
                new BasicAttribute("uniquePermission",
                                   getPermissionDN(permissionName));
            mods[0] =
                new ModificationItem(DirContext.REMOVE_ATTRIBUTE, mod);
            context.modifyAttributes(getGroupDN(groupName), mods);
        } catch (NoSuchAttributeException e) {
            // Ignore errors if the attribute doesn't exist
        }
    }
...
} // end class
```

A hasPermission() metódus

A *hasPermission()* logikai metódus azt vizsgálja meg, hogy egy csoport rendelkezik-e egy joggal. Az implementáció logikája a *userInGroup()*-hoz hasonló.

```
...
public class LDAPManager {
...
  public boolean hasPermission(String groupName, String permissionName)
        throws NamingException {

        // Set up attributes to search for
        String[] searchAttributes = new String[1];
        searchAttributes[0] = "uniquePermission";

        Attributes attributes =
            context.getAttributes(getGroupDN(groupName),
                                  searchAttributes);
        if (attributes != null) {
            Attribute permAtts = attributes.get("uniquePermission");
            if (permAtts != null) {
                for (NamingEnumeration vals = permAtts.getAll();
                     vals.hasMoreElements();
                     ) {
                    if (permissionName.equalsIgnoreCase(
                        getPermissionCN((String)vals.nextElement()))) {
                        return true;
                    }
                }
            }
        }

        return false;
    }
...
} // end class
```

A getPermissions() metódus

A *getPermissions()* egy listát ad vissza mindazon jogelemekről, ami a megadott csoporthoz van éppen rendelve. A működés a *getMembers()*-hez hasonló.

```
...
public class LDAPManager {
...
   public List getPermissions(String groupName) throws NamingException {
        List permissions = new LinkedList();

        // Set up attributes to search for
        String[] searchAttributes = new String[1];
        searchAttributes[0] = "uniquePermission";

        Attributes attributes =
            context.getAttributes(getGroupDN(groupName),
                                  searchAttributes);
        if (attributes != null) {
            Attribute permAtts = attributes.get("uniquePermission");
            if (permAtts != null) {
                for (NamingEnumeration vals = permAtts.getAll();
                     vals.hasMoreElements();
                     permissions.add(
                         getPermissionCN((String)vals.nextElement()))) ;
            }
        }

        return permissions;
    }
...
} // end class
```

Egy valós példa

Befejezésül egy éles rendszerből vett AD (LDAP) példával szeretnénk zárni ezt a fejezetet. A lent látható *ModifyFields* class egy POJO, ami egy indexet és egy stringet képes eltárolni.

```
package test.ad;

public class ModifyFields {
    private int index;
    private String adAttribute;

    public int getIndex() {
        return index;
    }

    public void setIndex(int index) {
        this.index = index;
    }

    public String getAdAttribute() {
        return adAttribute;
    }

    public void setAdAttribute(String adAttribute) {
        this.adAttribute = adAttribute;
    }
}
```

A fenti osztályt használja az *LdapUtils* class, aminek a feladata egy AD programozási felületet adni az alkalmazás felé. Az AD elérhető a szokásos LDAP porton keresztül (általában 389), de létezik egy Global Catalog szolgáltatás is, aminek segítségével egyszerre láthatjuk az összes Windows domain-t. Ez utóbbi alapértelmezett portja a 3268. A konstruktor elkészíti az LDAP (*ctx*) és a Global Catalog (*ctxGC*) kontextust is. A *modifyUser()* metódus feladata a megadott user DN specifikált attribútumainak módosítása. A 61-63 sorokban a Global Catalog-ban (GC) történik egy keresés, ami csak *person* ÉS *user* lehet, valamint a megadott filter jellemző. A keresés eredménye a *result* lista, amin egy ciklussal (65-84 sorok) megyünk végig és feldolgozzuk. Ez a szükséges jellemzők módosítását jelenti, amely műveletet már a normál *ctx* LDAP kontextus segítségével végezzük, mert a GC nem alkalmas írásra.

```
1  package test.ad;
2
3  import java.util.ArrayList;
4  import java.util.Hashtable;
5  import javax.naming.Context;
6  import javax.naming.NamingEnumeration;
7  import javax.naming.directory.*;
8
9  import org.apache.log4j.Logger;
10
11 public class LdapUtils {
12
13         DirContext ctx = null;
14         DirContext ctxGC = null;
15         SearchControls ctlsGC = null;
16         Logger log;
17
18         public LdapUtils(ConnectType ldapConnect, Logger logger) throws Exception {
19                 log = logger;
20                 log.info("START LdapUtils constructor...");
21                 Hashtable<String, String> env = new Hashtable<String, String>();
22
23                 env.put(Context.INITIAL_CONTEXT_FACTORY,
24                                 ldapConnect.getInitialContextFactory());
25                 env.put(Context.PROVIDER_URL, ldapConnect.getProviderUrl());
26                 env.put(Context.SECURITY_AUTHENTICATION,
27                                 ldapConnect.getSecurityAuthentication());
28                 env.put(Context.SECURITY_PRINCIPAL, ldapConnect.getSecurityPrincipal());
29                 env.put(Context.SECURITY_CREDENTIALS,
30                                 ldapConnect.getSecurityCredentials());
31
32                 ctx = new InitialDirContext(env);
33
34                 // for Global Catalog
35                 Hashtable<String, String> envGC = new Hashtable<String, String>();
36
37                 envGC.put(Context.INITIAL_CONTEXT_FACTORY,
38                                 ldapConnect.getInitialContextFactory());
39                 envGC.put(Context.PROVIDER_URL, ldapConnect.getGCProviderUrl());
40                 envGC.put(Context.SECURITY_AUTHENTICATION,
41                                 ldapConnect.getSecurityAuthentication());
42                 envGC.put(Context.SECURITY_PRINCIPAL,
43                                 ldapConnect.getSecurityPrincipal());
44                 envGC.put(Context.SECURITY_CREDENTIALS,
45                                 ldapConnect.getSecurityCredentials());
46
47                 ctxGC = new InitialDirContext(envGC);
48                 String[] attrIDs = { "distinguishedName" };
49                 ctlsGC = new SearchControls();
50                 ctlsGC.setReturningAttributes(attrIDs);
```

```
51                  ctlsGC.setSearchScope(SearchControls.SUBTREE_SCOPE);
52                  log.info("END LdapUtils constructor.");
53          }
54
55          public void modifyUser(ConnectType ldapConnect, String[] row,
56                          int filterFileColumnIndex, ArrayList<ModifyFields> modifyFields)
57                          throws Exception {
58                  log.info("START modifyUser...");
59                  log.info("Email: " + row[filterFileColumnIndex]);
60
61                  NamingEnumeration<SearchResult> result = ctxGC.search("", "(&(objectCategory=➥
                        person)(objectClass=user)("
62                                  + ldapConnect.getFilterAttribute() + "="
63                                  + row[filterFileColumnIndex] + "))", ctlsGC);
64
65                  while (result.hasMoreElements()) {
66                          SearchResult searchResult = result.next();
67                          String distinguishedName = searchResult.getNameInNamespace();
68                          log.info("DistinguishedName: " + distinguishedName);
69
70                          if (distinguishedName != null) {
71                                  ModificationItem[] mods = new ModificationItem[modifyFields
72                                                  .size()];
73                                  for (int i = 0; i < modifyFields.size(); i++) {
74                                          ModifyFields mf = modifyFields.get(i);
75                                          mods[i] = new ModificationItem(
76                                                          DirContext.REPLACE_ATTRIBUTE, new ➥
                                                              BasicAttribute(
77                                                                      mf.getAdAttribute(), row➥
                                                                          [mf.getIndex()]));
78                                  }
79
80                                  ctx.modifyAttributes(distinguishedName, mods);
81                          } else {
82                                  log.info("Unsuccessful update! There is not distinguishedName!")➥
                                        ;
83                          }
84                  }
85                  log.info("END modifyUser.");
86          }
87
88          public void releaseAll() {
89                  try {
90                          if (ctx != null) {
91                                  ctx.close();
92                          }
93                          if (ctx != null) {
94                                  ctxGC.close();
95                          }
96                  } catch (Exception e) {
97                          log.info("Error releasing resources!",e);
98                  }
99          }
100 }
```

A Java és AD kapcsolatáról javasoljuk ezt a 2 webhelyet is áttanulmányozni:

- JCIFS: `http://jcifs.samba.org/`
- `http://www.javaactivedirectory.com/`