

# JEGYZŐKÖNYV

WebXML

Könyvtár

Készítette: **Péter Balázs**

Neptun-kód: **BUM37G**

Dátum: 2025.November

1.Bevezetés.....	2
2. Feladat leírása .....	2
1. Feladat.....	3
1.1 Az adatbázis ER modell tervezése .....	3
1.2 Az adatbázis konvertálása XDM modellre .....	4
1.3 XDM modell alapján XML dokumentum .....	5
1.4 XML dokumentum alapján XMLSchema .....	7
2.feladat.....	9
2.1 Adatolvasás .....	9
2.2 Adat-lekérdezés .....	11
2.3 Adatmódosítás .....	13

## 1.Bevezetés

A beadandó feladat célja egy könyvtári adatkezelő rendszer modellezése különböző XML-alapú technológiák segítségével. A feladat során először elkészítettem az ER és XDM diagramokat, majd ezek alapján létrehoztam a rendszerhez tartozó XML dokumentumot, az XML Sémát (XSD), valamint a DOM-alapú Java programokat a dokumentum beolvasására, lekérdezésére és módosítására. A munka során az volt a fő szempont, hogy az egyes modellek és programok logikailag következetesek legyenek, és a valós könyvtári működést minél pontosabban tükrözzék.

## 2. Feladat leírása

A beadandó fő célja egy könyvtári adatnyilvántartó rendszer megtervezése és XML formátumban történő megvalósítása. Első lépésként elkészítettem a rendszer ER modelljét, amelyben az alapvető entitásokat (Könyv, Olvasó, Könyvtáros, Kölcsönzés, Olvasókártya és Kategória) valamint azok kapcsolatait határoztam meg. Az ER modell segítségével áttekinthetővé vált a rendszer logikai felépítése és a különböző adatelemek összefüggései.

A modell következő szintre emeléséhez létrehoztam az XDM diagramot, amely az ER modell struktúráját XML-kompatibilis formában jeleníti meg. Itt már figyelni kellett arra, hogy a relációs kapcsolatok helyett fa alapú hierarchiát alakítsak ki, ezért a gyökérellem a Könyvtár lett, és a hozzá tartozó csoportosított elemek (pl. Könyvek, Olvasók, Kölcsönzések) ennek

gyermekelemeiként szerepelnek. A modell célja az volt, hogy közvetlenül lefordítható legyen egy érvényes XML dokumentummá.

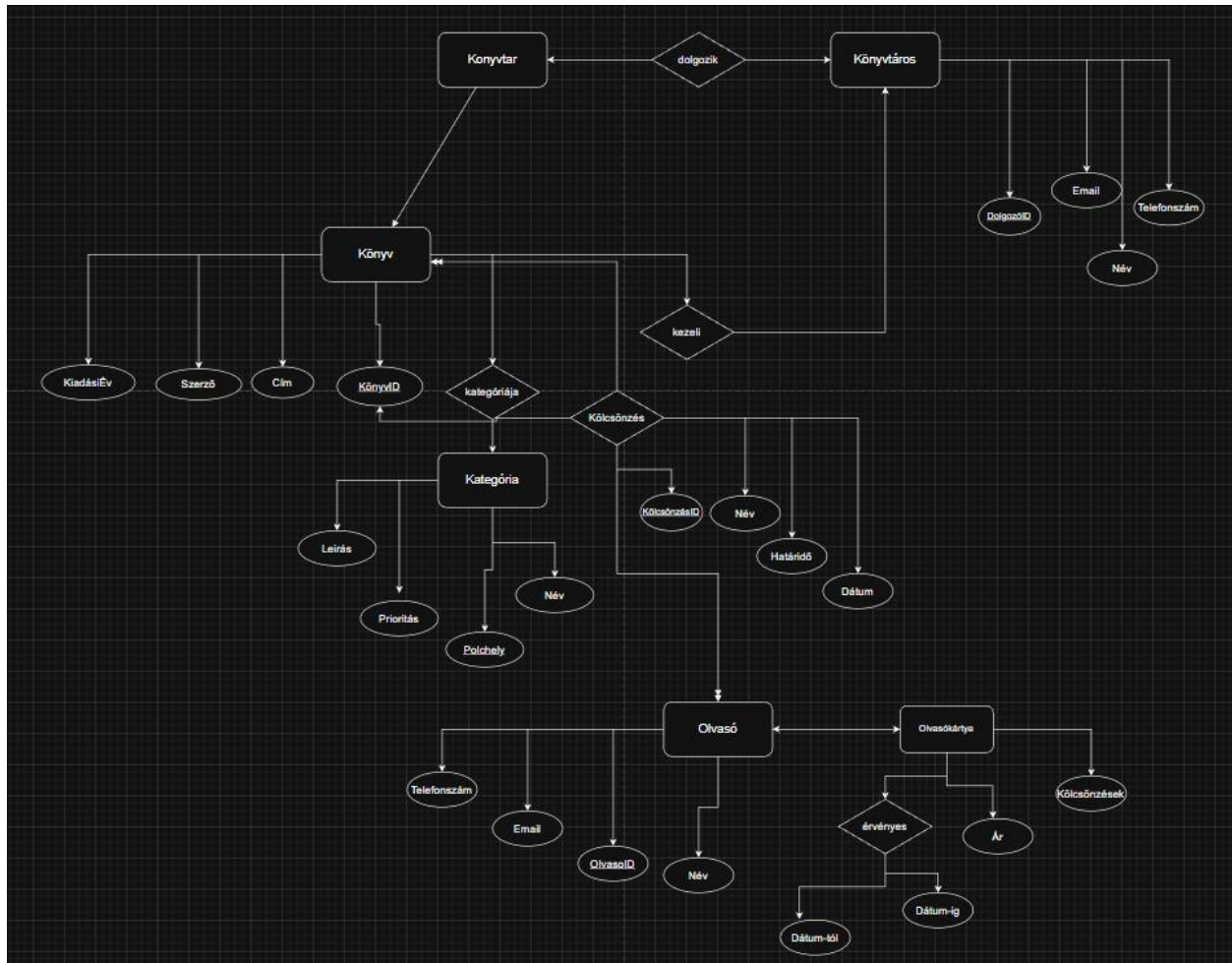
A következő lépésben elkészítettem magát az XML állományt, amely a könyvtári rendszer mintapéldányait tartalmazza. A feladat előírása szerint minden ismétlődő elemcsoportból legalább két példányt kellett megadni, valamint néhány megjegyzést is el kellett helyezni a dokumentumban. Az XML alapján elkészült a hozzá tartozó XML séma is (XSD), amely definiálja az adatok felépítését, típusaikat és az elemek egymáshoz való viszonyát.

A feladat végső részében három Java programot készítettem DOM parser használatával. A `bun37gDOMRead` osztály feladata a teljes XML dokumentum beolvasása és a konzolra történő strukturált kiírása. A `bun37gDOMQuery` osztály több lekérdezést valósít meg DOM bejárással, például könyvek szűrését és olvasókhöz tartozó kölcsönzések megjelenítését. A `bun37gDOMModify` osztály pedig különféle módosításokat hajt végre az XML állományon, mint például új elem hozzáadása, adatok átírása vagy törlése, majd a változtatott dokumentumot új fájlba menti.

Összességében a feladat jól szemlélteti, hogyan lehet egy adatbázis-logikájú rendszert XML formátumban megjeleníteni, majd azt programozott módon feldolgozni és módosítani.

## 1. Feladat

### 1.1 Az adatbázis ER modell tervezése



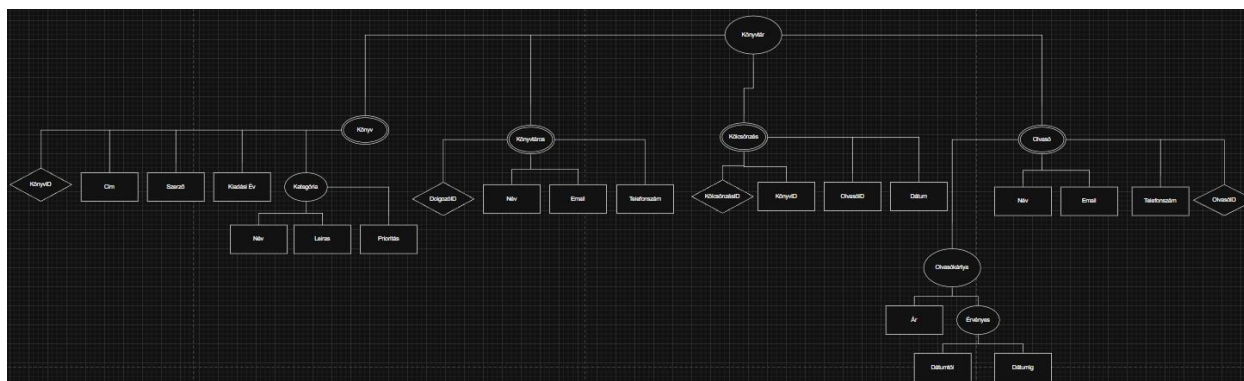
1. Ábra: ER Modell

Az ER modellt a könyvtári rendszer logikai felépítésének ábrázolására készítettem. A fő entitások: **Könyv**, **Olvasó**, **Könyvtáros**, **Kölcsönzés**, **Olvasókártya** és **Kategória**.

Mindegyikhez megadtam a szükséges attribútumokat és az egyedi azonosítókat (pl. KönyvID, OlvasóiID), mivel ezek biztosítják az egyértelmű hivatkozást.

A kapcsolatok meghatározásánál a valódi könyvtári működést vettem alapul – például egy olvasó több kölcsönzést is létrehozhat, ezért a Kölcsönzés köztes entitásként működik. A cél az volt, hogy egy jól áttekinthető, relációs adatbázisnak megfelelő struktúrát hozzak létre, amelyből később az XDM és XML modell is könnyen előállítható.

## 1.2 Az adatbázis konvertálása XDM modellre



2. Ábra: XDM modell

Az XDM modellben az ER modell elemeit XML-alapú hierarchiába rendeztem. A **Könyvtár** lett a gyökérel, alatta külön csoportokba tettem a könyveket, olvasókat, könyvtárosokat, kölcsönzéseket és olvasókártyákat.

Az attribútumokat téglalapokkal, az entitásokat ellipszisekkel jelöltem. A relációs idegenkulcsokat egyszerű XML tagek képviselik (pl. <KönyvID>, <OlvasóID>), mivel XML-ben nincs hagyományos idegenkulcs-kezelés.

A cél egy olyan XDM struktúra volt, amely 1 az 1-ben átfordítható konkrét XML dokumentummá, és jól követi az ER modellben szereplő szerkezeti logikát.

### 1.3 XDM modell alapján XML dokumentum

Az XML dokumentumot az elkészített XDM modell alapján hoztam létre. A cél az volt, hogy a könyvtári rendszer adatait egy fa-struktúrában, áttekinthető és könnyen feldolgozható formában jelenítsem meg. Gyökérelként a **Könyvtár** szerepel, amely alá logikusan csoportosítottam a könyveket, olvasókat, könyvtárosokat, kölcsönzéseket és olvasókártyákat. A feladat előírásának megfelelően minden ismétlődő elemcsoportból legalább két példányt hoztam létre.

A tervezés során arra törekedtem, hogy az XML szerkezete kövesse a valós rendszer logikáját: a könyvekhez például egy beágyazott Kategória elem tartozik, az olvasókártyán belül pedig az Ervenyes elem adja meg az érvényességi tartományt. A kapcsolatok megjelenítése a relációs modellben megszokott idegenkulcsok helyett egyszerű tagekkel történt (pl. <KönyvID>, <OlvasóID>), hiszen XML-ben nincs hagyományos idegenkulcs mechanizmus. A dokumentumban minimális kommenteket is elhelyeztem, ezzel is segítve az egyes részek értelmezését.

A végleges XML egy struktúrált, jól olvasható és a DOM parserrel könnyen feldolgozható dokumentum lett, amely közvetlenül az XSD és a Java programok alapját adja.

<Könyvtár>

```

<!-- Konyvek listaja -->
<Konyvek>

  <!-- Elso konyv -->
  <Konyv>
    <KonyvID>1</KonyvID>
    <Cim>A Gyuruk Ura</Cim>
    <Szerzo>J.R.R. Tolkien</Szerzo>
    <KiadasiEv>1954</KiadasiEv>

    <!-- Kapcsolt kategoria -->
    <Kategoria>
      <Nev>Fantasy</Nev>
      <Leiras>Kozepfold tortenete</Leiras>
      <Prioritas>1</Prioritas>
    </Kategoria>
  </Konyv>

  <!-- Masodik konyv -->
  <Konyv>
    <KonyvID>2</KonyvID>
    <Cim>A Paldini Rejtely</Cim>
    <Szerzo>Mate Bence</Szerzo>
    <KiadasiEv>2021</KiadasiEv>

    <Kategoria>
      <Nev>Krimi</Nev>
      <Leiras>Detektiv tortenet</Leiras>
      <Prioritas>2</Prioritas>
    </Kategoria>
  </Konyv>

</Konyvek>

```

A megadott kódrészlet a teljes dokumentum Konyvek részét mutatja be, amelyben több könyv adatai találhatóak. Ez a rész felelős a könyvek csoportosított tárolásáért.

A Konyvtar elem alatt található a Konyvek gyűjtőelem, amely tartalmazza a többször előforduló Konyv elemeket. Minden Konyv elem egy önálló könyv adatait írja le. Az egyes könyvekhez

tartozó alapvető információk a KonyvID, a Cim, a Szerzo és a KiadasiEv elemekben jelennek meg. Ezek megfelelnek az ER modell Könyv entitásának attribútumainak.

A Konyv elemben található egy beágyazott Kategória rész is. Ez azért szükséges, mert a kategória a könyvhez tartozó, összetett adatstruktúra, amely önmagában három további elemet tartalmaz: a Nev, a Leiras és a Prioritas elemeket. Ezek a kategória nevét, rövid leírását és besorolási fontosságát adják meg. A beágyazott struktúra azt jelzi, hogy a Kategória szorosan kapcsolódik az adott könyvhöz, és logikailag annak része.

A kódrészlet végén egy második Konyv elem is szerepel, amely ugyanazt a szerkezetet követi. Erre azért van szükség, mert a feladat előírta, hogy minden ismétlődő adategységből legalább két példányt kell szerepeltetni. A második könyv más adatokat tartalmaz, de a szerkezete megegyezik az elsőével, így biztosítva a dokumentum egységességét és a mintaadatok sokszínűségét.

## 1.4 XML dokumentum alapján XMLSchema

Az XML Schema (XSD) dokumentum feladata az volt, hogy formálisan meghatározza az XML fájl szerkezetét, adattípusait és az engedélyezett előfordulások számát. A tervezés során az XDM modell és az elkészült XML dokumentum struktúráját vettem alapul, majd ezekhez igazítva hoztam létre a szükséges komplex típusokat és elemeket. A gyökérelem a Konyvtar lett, amely több összetett, belső szerkezetű elemet tartalmaz, például a Konyveket, Olvasokat, Konyvtarosokat és Kolcsonzeseket.

A komplex típusok használata lehetővé tette, hogy az ismétlődő adatstruktúrákat külön típusként definiáljam (például KonyvType, OlvasoType stb.), és ezeket később az XML dokumentum egyes részei könnyen hivatkozassák. A minOccurs és maxOccurs attribútumokat úgy állítottam be, hogy a feladat feltételét teljesítsem, vagyis minden ismétlődő adatszerkezetből legalább két példányt lehessen elhelyezni. Az XML Schema így egyértelmű szabályrendszert biztosít az XML dokumentum helyességéhez, és megakadályozza a hibás vagy hiányos adatok bevitelét.

```
<!-- ==== Gyökérelem definicio === -->
<!-- Konyvtar -->
<xsd:element name="Konyvtar">
  <xsd:complexType>
    <xsd:sequence>

      <!-- Konyvek -->
      <!-- Ketto vagy tobb peldany engedelyezve -->
      <xsd:element name="Konyvek">
        <xsd:complexType>
```

```

        <xsd:sequence>
            <xsd:element name="Konyv" type="KonyvType"
minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

    <!-- Olvasok -->
    <xsd:element name="Olvasok">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Olvaso" type="OlvasoType"
minOccurs="2" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Konyvtarosok -->
    <xsd:element name="Konyvtarosok">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Konyvtaros" type="KonyvtarosType"
minOccurs="2" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Kolcsonzesek -->
    <xsd:element name="Kolcsonzesek">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Kolcsonzes" type="KolcsonzesType"
minOccurs="2" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

```

A megadott kódrészlet az XSD dokumentum legfelső, szerkezeti felépítésért felelős részét mutatja. A Konyvtar gyökérelem komplex típusa egy sequence szerkezetet tartalmaz, amely sorban felsorolja a rendszer fő adategységeit. Ezek mind külön csoportosító elemek (Konyvek,



Olvasok, Könyvtárosok, Kolcsonzesek), és mindegyik tartalmaz egy belső sequence részt a többször előforduló összetett elemek kezelésére.

A Könyvek elem például egy olyan sequence-et tartalmaz, amelyben a Könyv elem szerepel. Ehhez hozzárendeltem a KönyvType típust, amely külön definiálja majd a könyv összetett adatait. A minOccurs="2" és maxOccurs="unbounded" azt jelenti, hogy legalább két könyvnek szerepelnie kell az XML dokumentumban, de felső határ nincs. Ugyanez a logika érvényes az Olvaso, Könyvtáros és Kolcsonzes elemekre is: mindegyikből minimum két példány kötelező, és tetszőleges számú további példány engedélyezett.

Ez a szerkezet biztosítja, hogy az XML dokumentum megfeleljen a feladat követelményeinek, és egyértelműen meghatározza a fő adatcsoportok ismétlődő szerkezetét.

## 2.feladat

### 2.1 Adatolvasás

Link:

[https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G\\_XMLTASK/bum37gDOMRead.java](https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G_XMLTASK/bum37gDOMRead.java)

A DOMRead program célja az volt, hogy az XML dokumentum tartalmát DOM alapú feldolgozással beolvassa, majd a benne található adatokat strukturáltan kiírja a konzolra. A tervezés során először a teljes XML fájl betöltő rész készült el, amelyhez a DocumentBuilderFactory és DocumentBuilder osztályok szolgáltak alapul. A Document objektum létrehozása után a program normalizálja a dokumentumot, hogy a szöveges csomópontok megfelelő módon egyesüljenek.

A kiírások szervesen, külön segédfüggvényekben történnek. Minden nagyobb adategység (könyvek, olvasók, könyvtárosok, kölcsönzések, olvasókártyák) külön metódust kapott, amely a megfelelő tageket NodeList formában kiolvassa, majd bejárja azokat. A getText segédfüggvény gondoskodik arról, hogy a program egyszerűen hozzáférjen egy adott elem gyerek-tagjának szövegéhez, így elkerülhető a kódismétlés.

A program felépítése jól tükrözi az XML dokumentum szerkezetét: minden metódus csak a neki megfelelő adatrészért felel. Ez átláthatóvá teszi a kódot, és megkönnyíti a hibakeresést, illetve további bővítést. A megoldás külön előnye, hogy a DOM feldolgozás miatt az egész dokumentum a memóriában marad, így gyorsan elérhetők a különböző részek.

## 1. Az XML dokumentum betöltése

```
DocumentBuilderFactory factory =  
DocumentBuilderFactory.newInstance();  
DocumentBuilder builder = factory.newDocumentBuilder();  
Document doc = builder.parse(xmlFile);
```

Ez a rész felelős a DOM objektum létrehozásáért. A factory és builder kombinációjával a program képes megnyitni és értelmezni az XML fájlt. A parse metódus eredménye a Document objektum, amely a teljes XML-fát memóriában tartalmazza.

## 2. Elemek kilistázása NodeList segítségével

```
NodeList konyvList = doc.getElementsByTagName("Konyv");  
for (int i = 0; i < konyvList.getLength(); i++) {  
    Node node = konyvList.item(i);  
    if (node.getNodeType() == Node.ELEMENT_NODE) {  
        Element konyv = (Element) node;
```

Ez a kódrészlet mutatja, hogyan lehet egy adott elemcsoportot kilistázni. A `getElementsByTagName` minden "Konyv" tagelemet összegyűjt, majd a ciklus bejárja őket. A `node` típusának ellenőrzése biztosítja, hogy valódi elemmel dolgozunk, nem szövegcsomóponttal.

## 3. Gyerek-elemek olvasása segédfüggvénnyel

```
private static String getText(Element parent, String tagName) {  
    NodeList list = parent.getElementsByTagName(tagName);  
    if (list.getLength() == 0) return "";  
    return list.item(0).getTextContent().trim();  
}
```

Ez a metódus azért fontos, mert egységes módon biztosítja a gyermek-elemek szövegének beolvasását. Így nem kell mindenhol külön DOM műveletekkel elérni a gyerek-csomópontokat, ami jelentősen egyszerűsíti a többi metódust.

## 4. Beágyazott elemek kezelése

```
Element erv = (Element) e.getElementsByTagName("Ervenyes").item(0);  
if (erv != null) {  
    System.out.println("    Ervenyes:");  
    System.out.println("        Datumtol: " + getText(erv, "Datumtol"));
```

```
        System.out.println("    Datumig : " + getText(erv, "Datumig"));
    }
```

Ez a rész azt mutatja, hogyan kezeli a program a hierarchikusan elhelyezkedő XML elemeket. Az Olvasokartya elemen belül az Ervenyes rész további adatokat tartalmaz, amelyeket ugyanúgy, a `getText` függvénnyel lehet kiolvasni.

## 2.2 Adat-lekérdezés

Link:

[https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G\\_XMLTASK/bum37gDOMQuery.java](https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G_XMLTASK/bum37gDOMQuery.java)

A DOMQuery program célja az XML dokumentum lekérdezése DOM alapú bejárással, XPath használata nélkül. A feladat szerint legalább négy különböző lekérdezést kellett megvalósítani, ezért a programot úgy terveztem meg, hogy minden lekérdezés külön jól elkülöníthető metódusban kapjon helyet. A kód elején ugyanaz a dokumentumbetöltő rész található, mint a DOMRead esetben, így a feldolgozás alapja egy normalizált Document objektum.

A lekérdezések mind NodeList bejárással működnek: a program végigiterál az adott elemcsoporton, majd a gyerek-elemek értékei alapján dönt arról, hogy kiírja-e az eredményt. A `getText` metódus itt is kulcsszerepet játszik, mivel egyszerű módon teszi elérhetővé a gyermek-tagok szövegét. A `findOlvasoNev` segédfüggvény lehetővé teszi, hogy egy másik adategységből (Olvaso) további információt kérjünk le (név), ezzel megvalósítva a „kapcsolat alapú” keresést DOM segítségével.

A program négy lekérdezést valósít meg:

- (1) 2000 után kiadott könyvek listázása,
- (2) egy megadott olvasó összes kölcsönzése,
- (3) érvényes olvasókártyával rendelkező olvasók egy adott napon,
- (4) könyvtárosok email címeinek kigyűjtése.

A megoldás jól mutatja, hogy DOM bejárással összetettebb, több adategységet érintő lekérdezések is megvalósíthatók.

### 1. A lekérdezések sorrendje és meghívása

```
queryKonyvek2000Utan(doc);
```

```
queryKolcsonzesekOlvasoSzerint(doc, "100");

queryErvenyesOlvasokartyak(doc, "2024-06-01");

queryKonyvtarosEmail(doc);
```

Ez a rész a program logikai felépítését mutatja. Minden lekérdezés önálló metódusként fut, így a kód jól áttekinthető marad, és a lekérdezések külön-külön is módosíthatók vagy bővíthetők.

## 2. Dátum-intervallum alapú lekérdezés

```
String tol = getText(erv, "Datumtol");
String ig = getText(erv, "Datumig");

if (datum.compareTo(tol) >= 0 && datum.compareTo(ig) <= 0) {
    String olvasoId = getText(kartya, "OlvasoID");
    String nev = findOlvasoNev(doc, olvasoId);
    System.out.println(" - OlvasoID=" + olvasoId + " (" + nev +
    ")");
}
```

Ez a rész azt mutatja be, hogyan lehet DOM segítségével dátum-intervallumot vizsgálni. A `compareTo` egyszerű, de hatékony módja annak, hogy eldöntsük: a vizsgált dátum beleesik-e az olvasókártya érvényességi tartományába.

## 3. Másik elem adatának lekérése lekérdezés közben

```
private static String findOlvasoNev(Document doc, String olvasoId) {
    NodeList list = doc.getElementsByTagName("Olvaso");
    for (int i = 0; i < list.getLength(); i++) {
        Element o = (Element) list.item(i);
        if (olvasoId.equals(getText(o, "OlvasoID"))) {
            return getText(o, "Nev");
        }
    }
    return "ismeretlen";
}
```

Ez a metódus bizonyítja, hogy DOM alapú lekérdezéssel összetettebb kapcsolatokat is kezelhetünk. A kölcsönzésből csak az `OlvasoID` érhető el, de ezzel a függvénnyel hozzáférünk az olvasó nevéhez is.

#### 4. String alapú évszűrés könyvek esetén

```
String evStr = getText(k, "KiadasEv");
if (!evStr.isEmpty()) {
    int ev = Integer.parseInt(evStr);
    if (ev > 2000) {
        System.out.println("    - " + getText(k, "Cim") + " (" + ev +
    ");");
    }
}
```

Ez a rész mutatja, hogyan lehet számadatot kinyerni az XML-ből, majd arra logikai feltételt alkalmazni. A DOM alapvetően minden adatot szöveggént ad vissza, így szükség van átalakításra és ellenőrzésre.

### 2.3 Adatmódosítás

Link:

[https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G\\_XMLTASK/bum37gDOMModify.java](https://github.com/peterbalazsbence/BUM37GWebXML/blob/84ee269af3b902e4cc39b1278ad6fe29ecf07db8/BUM37G_XMLTASK/bum37gDOMModify.java)

A DOMModify program fő célja az XML dokumentum módosítása DOM alapú műveletekkel. A tervezés során arra törekedtem, hogy a feladatban előírt legalább négy különböző módosítást (elem hozzáadása, módosítása, törlése) egyértelműen elkülönítve, külön metódusokban valósítsam meg. A program a DOMRead és DOMQuery osztályokhoz hasonlóan először betölti és normalizálja az XML dokumentumot, majd sorban végrehajtja a módosításokat.

A megvalósítás négy fő műveletet tartalmaz: egy új könyv létrehozását és hozzáadását a megfelelő elemhez, egy meglévő olvasó email címének átírását, egy kölcsönzés határidejének módosítását, valamint egy másik kölcsönzés teljes törlését. A módosítások elvégzése után a program a dokumentumot konzolra is kiírja, végül pedig egy új fájlba menti, hogy az eredeti XML érintetlen maradjon. A DOM alapú feldolgozás előnye, hogy a teljes dokumentum a memóriában van, így a node-ok közvetlenül és gyorsan módosíthatók.

A program szerkezete áttekinthető: minden módosítást külön metódus végez, és egy segédfüggvény segíti a gyermek-elemek szövegének kiolvasását. Ez megkönnyíti a fejlesztést és a későbbi bővítést is.

#### 1. Új könyv létrehozása és hozzáadása

```
Element ujKonyv = doc.createElement("Konyv");
```

```

Element id = doc.createElement("KonyvID");
id.setTextContent("3");
ujKonyv.appendChild(id);

Element cim = doc.createElement("Cim");
cim.setTextContent("XML programozas");
ujKonyv.appendChild(cim);

Element szerzo = doc.createElement("Szerzo");
szerzo.setTextContent("Bum Developer");
ujKonyv.appendChild(szerzo);

Element ev = doc.createElement("KiadasiEv");
ev.setTextContent("2024");
ujKonyv.appendChild(ev);

Element kat = doc.createElement("Kategoria");
Element nev = doc.createElement("Nev");
nev.setTextContent("Szakirodalom");
Element leiras = doc.createElement("Leiras");
leiras.setTextContent("XML es DOM");
Element prior = doc.createElement("Prioritas");
prior.setTextContent("3");
kat.appendChild(nev);
kat.appendChild(leiras);
kat.appendChild(prior);

ujKonyv.appendChild(kat);

// hozzaadas a Konyvek listahoz
konyvekElem.appendChild(ujKonyv);

```

Ez a rész mutatja be az új elem létrehozásának folyamatát. A createElement segítségével új XML elemek hozhatók létre, majd a setTextContent beállítja az adatokat. Az új könyv végül a Konyvek lista végére kerül.

## 2. Elem módosítása – email cím frissítése

```

Element emailElem = (Element)o.getElementsByTagName("Email").item(0);
emailElem.setTextContent(newEmail);

```

Ez a kódrészlet azt mutatja, hogyan módosítható egy létező XML elem tartalma. A megfelelő Email elem elérése után annak szövege egyszerűen lecserélhető az új értékre.

### 3. Elem módosítása – határidő átírása

```
Element hataridoElem = (Element)k.getElementsByTagName("Hatarido").item(0);  
hataridoElem.setTextContent(newHatarido);
```

Ez a működés nagyon hasonló az email módosításához, de itt egy kölcsönzés határideje változik meg. A kódrészlet jól szemlélteti a DOM módosítás egyszerűségét.

### 4. Elem törlése a dokumentumból

```
Node parent = k.getParentNode();  
parent.removeChild(k);
```

Ez a rész mutatja meg, hogyan távolítható el egy teljes elem a dokumentumból. Miután megtaláltuk a törlendő Kolcsonzes elemet, egyszerűen a szülő node-ot kérjük meg, hogy távolítsa el azt.

### 5. A módosított dokumentum mentése

```
Transformer transformer = tf.newTransformer();  
transformer.setOutputProperty(OutputKeys.INDENT, "yes");  
transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
```

Ez a rész a dokumentum fájlba mentéséért felel. A Transformer segítségével a módosított DOM objektum XML formátumba alakítható és elmenthető. A beállított indentálás jobb olvashatóságot biztosít az új fájlban.