Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Osztálybetöltés

A ClassLoader osztály I.

- *"Kezdetben volt a virtuális gép…"*
- Az osztályok byte kódját futtatás előtt **be kell tölteni a VM-be**.
- Az osztálybetöltés **on-demand** módon, **dinamikusan** történik, mindig csak az éppen szükséges osztályt tölti be a VM.
- Az osztálybetöltést a **java.lang.ClassLoader** osztály, illetve annak leszármazottai végzik.
- A ClassLoader osztály leszármaztatásával létrehozhatunk olyan betöltőket, amelyek **nem a file-rendszerből**, hanem például **hálózatról**, vagy **adatbázisból** töltenek be osztályokat.
- Egy VM-en belül **egyszerre több ClassLoader** is lehet.

A ClassLoader osztály II.

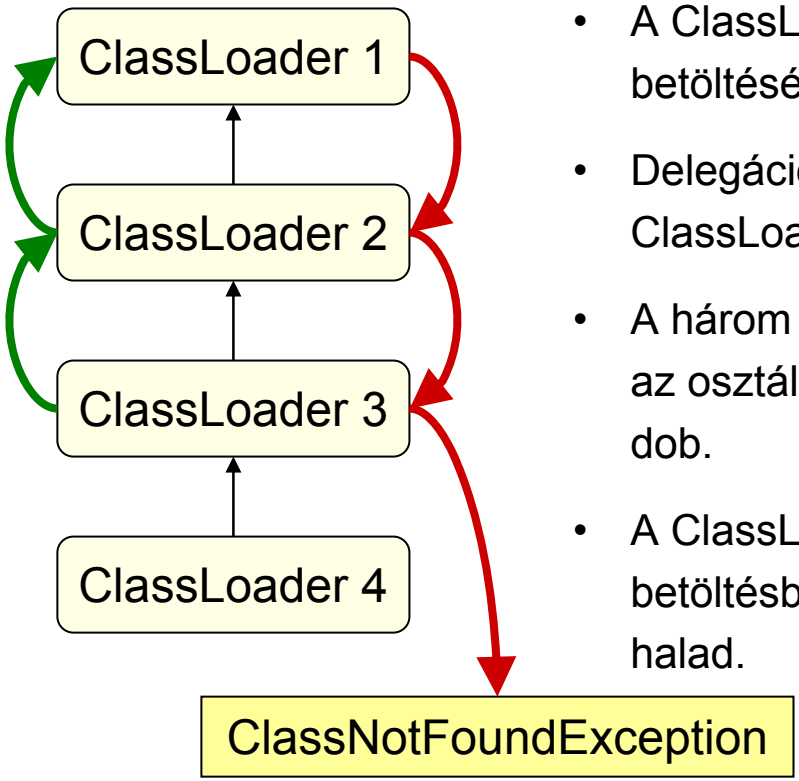
- Minden betöltőnek van (lehet) egy **szülő**je (parent).
- A gyermek-szülő viszonyok mentén a betöltők **fá**t (illetve erdőt) alkotnak.
- Egy osztály betöltése az alábbiak szerint zajlik:
  - Ha a meghívott betöltő az osztályt korábban már betöltötte, a betöltési folyamat végetér.
  - Ha nem, a betöltő először továbbadja a kérést a szülőjének.
  - Ha ez nem vezetett eredményre, a gyermek próbálkozik meg az osztály betöltésével.
  - Ha ez sikerül, a betöltés sikeresen végetért, ha nem, az eredetileg meghívott betöltő ClassNotFoundException-t dob.

Delegáció I.

- Az előzőekben vázolt mechanizmust **delegáció**nak nevezzük.
- A delegáció biztosítja azt, hogy egy speciális betöltőn keresztül (például amelyik hálózatról tölt be osztályokat) is be tudjunk tölteni olyan osztályokat, amelyeket ténylegesen csak annak szülői tudnak betölteni.
- Minden betöltött osztály tudja, hogy **őt melyik betöltő töltötte be**. Amikor ezen osztály kódja egy új osztályra hivatkozik, **a saját betöltőjét utasítja** a hivatkozott osztály betöltésére.
- Ez azt jelenti, hogy legelőször a meghívott betöltő fájának a gyökerében lévő betöltő próbálja meg betölteni az osztályt, majd pedig a meghívott betöltőig vezető ágon lévő többi betöltő.

Delegáció II.

- Az alábbi példában egy nemlétező osztályt próbálunk betölteni



- A ClassLoader 3-at utasítjuk az osztály betöltésére.
- Delegáció útján a betöltési igény eljut a ClassLoader 1-ig.
- A három betöltő közül egy sem tudja betölteni az osztályt, így végül a ClassLoader 3 kivételt dob.
- A ClassLoader 4 nem vesz részt a betöltésben, mert a fában az igény lefelé nem halad.

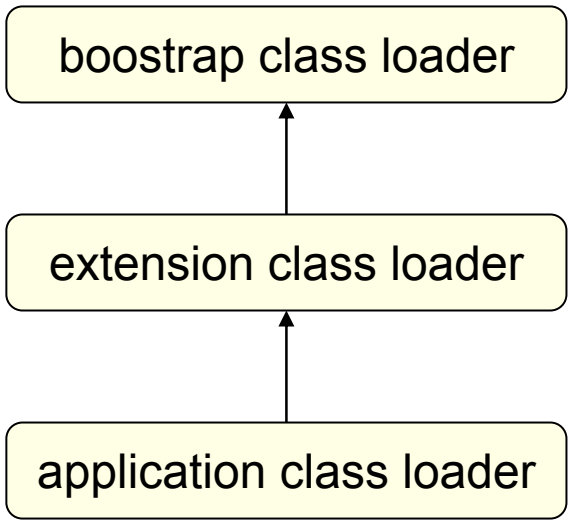
A bootstrap betöltő

- Az osztályokat ClassLoader-ek töltik be.
- Mi tölti be magát a ClassLoader osztályt?
- Az úgynevezett **bootstrap class loader** natív kódban (platformfüggően) van megvalósítva, a VM szerves részét képezi. Ez a betöltő tölti be a ClassLoader-t, és a Java futtató környezet (Java Runtime) osztályait.
- A bootstrap betöltőt a **null** referencia jelképezi, és ennek a betöltőnek nincs szülője sem.
- Ebből az következik, hogy ha egy betöltőnek a **szülőjeként a null referenciát adjuk meg**, a **szülője a bootstrap betöltő lesz**. A betöltők tehát összefüggő fát alkotnak.

A futtatókörnyezet betöltő I.

- A futtató környezet indításkor a bootstrap betöltőn kívül több betöltőt is létrehoz, melyek más és más helyekről töltenek be osztályokat.
  - A **bootstrap class loader** (elvileg) az rt.jar-ból tölt be, ez a JAR file tartalmazza a Java Runtime beépített osztályait.
  - Az **extension class loader**, mely az Extension Mechanism betöltője (lásd később), egy speciális könyvtárból tölt be.
  - Az **application class loader**, mely a classpath-ban megadott könyvtárakból, illetve JAR file-okból tölti be az alkalmazások osztályait.
- Ez a három betöltő a betöltők fájában egy ágon helyezkedik el.

A futtatókörnyezet betöltő II.



- Amikor tehát a java paranccsal elindítunk egy alkalmazást, a megfelelő osztályt először a bootstrap, aztán az extension, végül az application betöltő fogja megpróbálni betölteni.
- Általában, mivel az alkalmazás osztályai a classpath-ban megadott útvonalakon vannak, az application betöltő tölti be az alkalmazások osztályait.

- A hivatkozott osztályok betöltésére vonatkozó korábban már megismert szabály miatt, az alkalmazás összes további osztályát is az application betöltőn keresztül fogja a rendszer megpróbálni betölteni.
- Az általános gyakorlatban tehát **az alkalmazás összes osztályát az application betöltő tölti be**.

Az Extension Mechanism I.

- Az Extension Mechanism célja az, hogy a Java rendszert **opcionális csomagokkal kiterjeszthessük**. Az opcionális csomagokat (optional packages) korábban standard kiterjesztéseknek (standard extensions) nevezték.
- Az opcionális csomagok leggyakrabban **nyílt szabványok** megvalósításai (ilyen például a Java Cryptography).
- Természetesen e mechanizmus nélkül is megtehetnénk ezt, de úgy a könyvtárak JAR file-jainak nevét egyenként meg kell adnunk a classpath-ban az alkalmazás fordításakor és futtatásakor.
- Az extension betöltő az application betöltő felett helyezkedik el, és nem a classpath-ból tölt be osztályokat, hanem egy külön könyvtárból. Az ide elhelyezett osztályokat, JAR-okat az extension betöltő automatikusan beemeli, nem kell őket egyenként megadnunk.

Az Extension Mechanism II.

- Az Extension Mechanism révén betöltött osztályokat – elvileg – két csoportra oszthatjuk:
  - **installed extensions**: a rendszerben állandóan jelenlévő kiterjesztések,
  - **download extensions**: egy-egy alkalmazáshoz használt kiterjesztések.
- A kiterjesztéseket alapértelmezésben a <JAVA-HOME>/lib/ext könyvtárból tölti be a rendszer, de parancssori paraméterként egyéb könyvtárakat is megadhatunk.
- Az állandó kiterjesztéseket tipikusan az alapértelmezett könyvtárban helyezzük el.

Az Endorsed Standards Override Mechanism

- A Java platform alapvetően a **Java Community Process (JCP)** keretében definiált szabványok megvalósításait tartalmazza.
- Van azonban néhány kivétel:
  - az **org.omg** csomag és alcsomagjai, egy CORBA implementáció,
  - az **org.w3c.dom** csomag, a W3C DOM szabványának megvalósítása, illetve
  - az **org.xml.sax** csomag és alcsomagjai, egy SAX XML parser.
- Mivel ezeket a szabványokat nem a JCP-n belül definiálják, előfordulhat, hogy a szabvány, és így az implementáció megváltozik két Java verzió között.
- Az új verziók JAR-jait egy külön könyvtárból a rendszer automatikusan beemeli (alapértelmezésben a <JAVA-HOME>/lib/endorsed).

Az osztálybetöltési mechanizmus veszélyei I.

- Mit ír ki a következő program?

```
import java.net.*;

public class Test {
   public Test () {
      System.out.println ("Példányszámláló: " + ++counter);
   }
   public static void main (String[] args) {
      try {
         new Test ();                                    // első példány
         URLClassLoader ucl = URLClassLoader.newInstance (
            new URL[] { new URL ("file:///teszt/") },null);
         Class c = ucl.loadClass ("Test");
         c.newInstance ();                              // második példány
      } catch (Exception e) { e.printStackTrace (); }
   }
   private static int counter = 0;
}
```

Az osztálybetöltési mechanizmus veszélyei II.

- Mit várunk?

```
> java Test
Példányszámláló: 1
Példányszámláló: 2
```

- Mit kapunk?

```
> java Test
Példányszámláló: 1
Példányszámláló: 1
```

- A Test osztályt először az application betöltő töltötte be, majd másodszor az általunk létrehozott URLClassLoader példány, így **a Test osztályból magából is két példány létezik**!
- A statikus mezők tehát **nem az egész VM-ben, hanem csak az egy ClassLoader által betöltött osztályok között konzisztensek**!

Az osztálybetöltési mechanizmus veszélyei III.

- A bootstrap betöltő alapértelmezésben az rt.jar-ból, és még néhány könyvtárból tölt be (ez platform- és verziófüggő), az **-Xbootclasspath** opcióval azonban ezt felüldefiniálhatjuk.
- Ennek az a következménye, hogy a Java beépített osztályait felüldefiniálhatjuk.
- Ez jó, mert:
  - rendkívüli szabadságot biztosít a programozónak **kísérletezés**re,
  - a programozó **saját környezet**et alakíthat ki magának.
- Ez rossz, mert:
  - **ellenőrizhetetlen** lesz **a beépített osztályok viselkedése**,
  - **biztonsági problémák** is felmerülhetnek.