Singleton

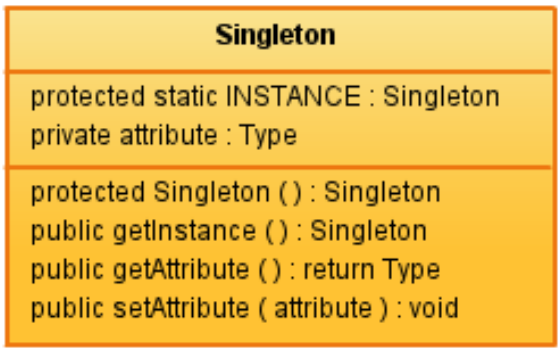
Ahogy a neve is mutatja, a ***singleton*** (magyarul *egyke*) egy példányban létezik a szkebb világunkban. A szkebb világ itt elssorban a virtuális gépet jelenti, de alkalmazás szerverek környékén akár alkalmazás szint is lehet ez a világ, ezért körültekinten használjuk: nem biztos, hogy két alkalmazásunk a ***singleton*** ugyanazon példányát használja.

A probléma

Sokszor fontos, hogy egy-egy osztályból csak egyetlen példány létezzen, gondoljunk például a programunk konfigurációs állományának beolvasására: tartalmát általában egyszer kell beolvasnunk, majd ezt fel kell dolgoznunk, aztán a program sok egyéb helyén használjuk a feldolgozott paramétereket. Ezen felül még több tucatnyi esetet lehet felsorolni, amikor egy osztályból csak egy példány létrehozása szükséges: különféle várakozósorok kezelése, egy példányból álló erforrások kezelése, stb.

A megoldás

Megoldásképpen létrehozhatunk globális (*public static*) változókat valahol a programunkban, amelyekbe a program indulásakor értékül adjuk egy-egy példányát a szükséges osztályoknak - ez azonban lehetvé teszi, hogy más is példányosíthassa a felhasznált osztályokat.



Sokkal szebb megoldás erre a feladatra ***singleton*** osztály használata, amely annyiban különbözik a fenti esettl, hogy önmaga tartalmazza saját egyetlen példányát egy globális változóban, a konstruktora pedig nem érhet el küls osztály számára, így az els és egyetlen példány létrehozását is maga a ***singleton*** végzi.

Ha a felhasználók kérnek egy példányt a ***singleton*** osztályunk, akkor nem feltétlen kell tudniuk, hogy másokkal osztoznak ezen a példányon, ezért ha felmerül a gyanú, hogy több szálból is hívhatják a visszaadott példány metódusait, akkor úgy kell megírnunk ezeket, hogy azok szál biztosan mködjenek.

Általánosságban a ***singleton*** osztályunkban biztosítani kell egy *nem publikus* konstruktort, amely létrehozza az egyetlen példányt, illetve léteznie kell egy *publikus és statikus* változónak/metódusnak, amely hordozza/visszaadja ezt az egyetlen példányt - vagyis önmagát.

Az implementáció

Felmerülhet a kérdés, hogy miért kell a ***singleton*** , amikor statikus osztályt is használhatnánk. A statikus osztály és a ***singleton*** között alapveten az a különbség, hogy a statikus osztály minden statikus tagja létrejön akkor, amikor a *classloader* betölti az osztályt, a ***singleton*** csak attól a pillanattól foglal memóriát, amikor *elször* használjuk azt. További különbség, hogy a statikus osztály nem implementálhat interfészt, míg a ***singleton*** igen, hiszen a Java szempontjából egy teljesen közönséges osztály. Nézzünk pár implementációt, amelyek más-más irányból közelítik meg a problémát: ugyan a végs eredmény azonos lesz, de az els példány létrehozása máskor és máshogy történik.

A klasszikus megoldás

Mivel a ***singleton*** lehetsége a Java egész korai verzióiban is rendelkezésre állt (hiszen ez - mint a legtöbb tervezési minta - egy OOP minta, nem a Java sajátja), ezért eleinte a klasszikus megoldást használhattuk:

Singleton.java

```
public class Singleton
{
  public static final Singleton INSTANCE = new Singleton();

  protected Singleton()
  {
    // ...
  }
}
```

Mint láthatjuk, itt a példány publikus, azaz nincs egy olyan metódus, amelyen keresztül elkérhetjük; ezen túl statikus, amely biztosítja az egyetlen példányt; s végül *final*, vagyis a tartalma nem módosítható többet. Az egyetlen probléma a korai példányosodás, ugyanis ez az implementáció a *Singleton* osztály betöltésekor létrehozza a példányát, ami nem mindig a leghasznosabb megoldás.

A lusta (lazy) Singleton

Kismértékben javíthatunk a kifacsarható teljesítményen, ha a ***singleton*** példány akkor jön létre, amikor elször használnák azt fel:

LazySingleton.java

```
public class LazySingleton
{
  protected static LazySingleton INSTANCE;

  protected LazySingleton()
  {
    // ...
  }

  public static synchonized LazySingleton getInstance()
  {
    if (INSTANCE == null)
    {
      INSTANCE = new LazySingleton();
    }

    return INSTANCE;
  }
}
```

Ennek a megoldásnak a hátránya, hogy a *getInstance* metódust meg kell védenünk attól, hogy egyszerre több szál is benne tartózkodjon, amely kissé lassít az elst követ használatokon, hiszen ekkor már a létrehozott példányt szeretnénk megkapni, amely mveletet nem kellene szinkronizálni.

A javasolt megoldás

Bill Pugh felfedezett egy olyan megoldást, amely a lehet legkésbb hozza létre az egyetlen példányt, és teljes mértékben szálbiztos:

OptimalSingleton.java

```
public class OptimalSingleton
{
  protected OptimalSingleton()
  {
    // ...
  }

  private static class SingletonHolder
  {
    private final static OptimalSingleton INSTANCE = new OptimalSingleton();
  }

  public static OptimalSingleton getInstance()
  {
    return SingletonHolder.INSTANCE;
  }
}
```

Ez a végletekig egyszer és hatékony megoldás a virtuális gép mködésére épít, ugyanis a *classloader* csak akkor tölti be a *SingletonHolder* osztályt, amikor valaki meghívja a *getInstance* metódust. A szálbiztosságot pedig az garantálja, hogy a *classloader* csak egyszer tudja betölteni a szükséges osztályt és ez JVM szint atomi mvelet.

Egyszeren csak enum

A Java 5 által behozott *enum* osztálytípus új lehetséget adott a programozók kezébe:

EnumSingleton.java

```
public enum EnumSingleton
{
  INSTANCE;

  EnumSingleton()
  {
    // ...
  }
}
```

A megoldás szálbiztos, de sajnos nem az els használatkor jön létre a példány, ellenben igen tömör megoldás. További elnye a többi megoldással szemben, hogy önmagában szerializálható, mivel az enum típusú osztályok erre automatikusan képesek.

A lusta (lazy) Singleton – Java 5 esetén

A Java 5 új memóriakezelést hozott a virtuális gépbe, ezért a *volatile* használatával lehetségünk van az alábbi megoldásra:

VolatileSingleton.java

```
public class VolatileSingleton
{

  private static volatile VolatileSingleton INSTANCE;

  protected VolatileSingleton()
  {
    // ...
  }

  public static VolatileSingleton getInstance()
  {
    if (INSTANCE == null)
    {
      synchronized (VolatileSingleton.class)
      {
        if (INSTANCE == null)
        {
          INSTANCE = new VolatileSingleton();
        }
      }
    }
    return INSTANCE;
  }
}
```

A kétszeres ellenrzés a példány lekérdezésének gyorsaságát biztosítja, a *volatile* pedig ügyel arra, hogy csak akkor legyen az *INSTANCE* értéke *nem null*, ha a konstruktor már lefutott.

Mérjünk!

Az öt implementáció különbözik egymástól lehetségekben és futási idkben, különböztessünk meg futási idket els használatra és további használatra. A ***sin gleton*** konstruktorába tegyünk egy 1000ms idej várakozást, és nézzük meg mennyi id alatt hajtódik végre az els ***singleton*** példány elkérése, illetve mennyi példányt tudunk elkérni a további alkalmak során:

Módszer	Els példány	További példányok
ClassicSingleton	~1000ms	> 10 millió / ms
LazySingleton	~1000ms	~20 ezer / ms
HolderSingleton	~1000ms	> 10 millió / ms
EnumSingleton	~1000ms	> 10 millió / ms

DoubleLazySingleton	~1000ms	~200 ezer / ms

Mint látszik, az els példány elkérése mindenhol közel 1000ms körül van, a különbség akkor ütközne ki, ha a *classloader* az els felhasználás eltt töltené be a ***singleton*** osztályt, de nehézségbe ütközik elbb használni az osztályt, minthogy használnánk azt... :)

A további példányok elkérése során látszik, hogy azok a *getInstance* metódusok igen rosszul teljesítenek, amelyekben szinkronizációt használunk, ezért lehetleg kerüljük ezeket, és próbáljuk meg a *HolderSingleton* mintát felhasználni, amely egyszer, szálbiztos és gyors.

(forrás: wikipedia)