Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

java.lang

Sipos Róbert
siposr@hit.bme.hu

2014. 02. 25.

Bevezetés

- A java.lang csomag olyan osztályokat (illetve interfészeket, kivételeket, stb.) tartalmaz, amelyek a Java nyelv elemeihez kapcsolódnak, azokat egészítik ki, illetve a Java és a külvilág kapcsolatát szolgálják.
- Ezen osztályok olyan gyakran használtak Java programokban, illetve olyan alapvető fontosságúnak tekintettek, hogy az egyszerű névvel megadott osztályokat a fordító a **java.lang csomagban is keresi import deklaráció nélkül** is. (implicit import)

Object I.

- Az Object osztály minden Java osztály implicit őse, vagyis minden Java osztály örökli a tagjait.
- Nincsenek példánymezői, egyrészt mert a Java objektumoknak elvileg nincs kötelező, általános állapota, másrészt azért, mert így nem növeli minden Java osztály memória-beli méretét.
- Példánymetódusai olyan műveleteket valósítanak meg, amelyek minden Java objektumra értelmezhetők.
- Nincsenek statikus tagjai, egyrészt mert ezek nem az objektumokhoz, hanem az osztályokhoz köthetők logikailag, így a Class osztályban kaptak helyet (lásd a Reflection API-nál), másrészt mert ezek minden osztály névterét fölöslegesen “szennyeznék”.

Object II.

- Az Object metódusai három csoportra oszthatók.
  - Az objektum **értékével** kapcsolatos metódusok:
    - equals, hashCode, toString
  - Az objektummal mint **szinkronizációs primitívvel** kapcsolatos metódusok:
    - notify, notifyAll, wait
  - Az objektum **életciklusával** kapcsolatos metódusok:
    - clone, finalize
- A második csoport metódusaival itt részletesebben nem foglalkozunk, lásd a szálakról szóló előadást.

Object III.

```
public boolean equals (Object o);
```

- Az **equals** metódus az objektum egyenlőségét vizsgálja a paraméterként kapott másik objektummal.
- A Javában két egyenlőség fogalom létezik.
  - A **referenciális egyenlőség**, amelyet az == operátor valósít meg. Két referencia akkor és csak akkor referenciálisan egyenlő, ha ugyanarra az objektumra mutatnak, vagy mindkettő null.
  - Az **equals metódus szerinti egyenlőség**. Az equals metódus jelentése az, hogy a két objektum “valamilyen értelemben” egyenlő. Ez azt jelenti, hogy az Object leszármazottai az egyenlőség kritériumát szabadon meghatározhatják. Annak érdekében, hogy az equals értelmes egyenlőségfogalomként használható legyen, az equals definíciója öt kritériumot fogalmaz meg.

Object IV.

- Az equals által definiált reláció legyen:
  - **Reflexív**. ∀x != null: x.equals (x) == true.
  - **Szimmetrikus**. ∀x != null, ∀y != null: x.equals (y) == true ⇔ y.equals (x) == true.
  - **Tranzitív**. ∀x != null, ∀y != null, ∀z != null: x.equals (y) == true ∩ y.equals (z) == true, ⇒ x.equals (z) == true.
  - **Konzisztens**. Ugyanazon két x-re és y-ra x.equals (y) értéke ismételt meghívásra konzisztensen true vagy false, ha az x és y állapotának az összehasonlításban résztvevő része nem változott.
  - ∀x != null: x.equals (null) == false.
- Az Object osztály equals implementációja egy egyszerű referenciális egyenlőségvizsgálat, vagyis ∀x != null: x.equals (y) == true ⇔ x == y.

Object V.

```
public int hashCode ();
```

- A **hashCode** metódus az objektum **hash értékét** számítja ki.
- A hashCode által számított érték legyen:
  - **Konzisztens az equals-szal**. ∀x, ∀y: x.equals (y) ⇒ x.hashCode () == y.hashCode ()
  - **Konzisztens**. Ismételt meghívásra ugyanazt az értéket kell visszaadja, feltéve, hogy az objektum két állapota az equals szerint megegyezik.
- Nem feltétel, hogy ha két objektum az equals szerint nem egyezik, akkor eltérő hash értéket adjon vissza, de a hash táblák optimális működése érdekében erre nyilvánvalóan törekedni érdemes.
- Az Object hashCode megvalósítása az objektum memóriacíméből állít elő hash értéket, ami különböző objektumokra gyakorlatilag különböző értéket ad.

Object VI.

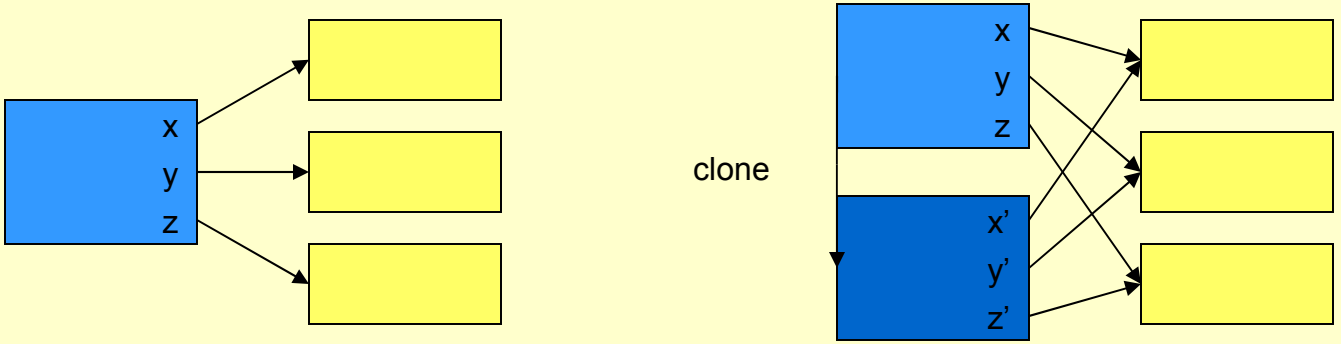
```
public String toString ();
```

- A **toString** metódus az objektum szöveges reprezentációját állítja elő.
- A cél általában az, hogy a szöveg legyen **ember által jól olvasható**, **tömör**, és a **lehető legtöbbet mondjon el az objektumról**, illetve annak állapotáról.
- Ideálisan minden Java osztályban felül kell definiálni ezt a metódust, bár ez nem mindig valósul meg.
- Az Object toString implementációja egy, az osztály nevét, egy “@”-t, és a hash érték hexadecimális reprezentációját tartalmazó szöveget állít elő.

Object VII.

```
protected Object clone ();
```

- A **clone** metódus az objektum másolatát állítja elő.
- Az Object implementációja az objektum “**shallow copy**”-ját állítja elő.
- A shallow copy az objektum gyakorlatilag bitről bitre pontos másolata, ebből következően csak egy rétegű másolat: az objektum mezőiben lévő esetleges referenciák ugyanazokra az objektumokra (példányokra) mutatnak.



Object VIII.

- A clone metódus konvencionálisan az objektum teljes, az eredetitől független másolatát állítja elő (**deep copy**).
- A másolhatónak szánt osztályokban felül kell definiálni a clone metódust úgy, hogy az a teljes másolatot előállítsa.
- A kiindulási alap a másoláshoz általában a super.clone ().
- Az Object clone megvalósítása CloneNotSupportedException kivételt dob, ha az osztály nem valósítja meg a Cloneable interfészt. A Cloneable nem tartalmaz tagokat, csak jelzésként szolgál, hogy az osztályról az Object implementációja szerinti shallow copy létrehozható.
- Az Object clone metódusa protected, ezt a leszármazottak public-ká tehetik, ha a másolást kívülről is meg akarják engedni.

Object IX.

```
protected void finalize ();
```

- A **finalize** metódust a szemétgyűjtő (garbage collector) hívja meg, mielőtt az elérhetetlenné vált objektumot felszámolná.
- Részletesebben lásd a szemétgyűjtőről szóló előadásban.

Burkoló osztályok I.

- A primitív és a referencia típusok közötti kapcsolatot a **burkoló osztályok** (wrapper class) biztosítják.
- Minden primitív típusnak megfelel egy burkoló osztály:
  - Numerikus típusok: Byte, Double, Float, Integer, Long, Short, ezek őse a Number.
  - Egyéb típusok: Boolean, Character.
- A burkoló objektumok minden esetben **immutábilisak**.
- A burkoló osztályok számos módszert biztosítanak a primitív típus, a burkoló osztály példányai, illetve az adott érték szöveges reprezentációja közötti konverzióra.
- Bizonyos típusok esetében további eszközöket biztosítanak az adott típusú értékek vizsgálatára illetve létrehozására.

Burkoló osztályok II.

- Minden burkoló osztályban megtalálhatók az alábbi metódusok:
  - Primitív érték és burkoló objektum közötti konverzió:
    - konstruktor, nnnValue
  - Szöveges reprezentáció és primitív érték közötti konverzió:
    - parseNnn, toString
  - Szöveges reprezentáció és burkoló objektum közötti konverzió:
    - konstruktor, valueOf, toString

Primitív érték –> burkoló objektum konverzió I.

- A burkoló osztályok a megfelelő primitív értékkel paraméterezett konstruktorai létrehoznak egy új példányt a burkoló osztályból, amely a megadott értéket tartalmazza:

```
Integer i = new Integer (25);
```

- A statikus **valueOf** metódus szintén egy, az adott értéket tartalmazó burkoló példányt ad vissza, azonban a gyakran kért értékekhez tartozó példányokat cache-eli.

```
Integer i = Integer.valueOf (25);
```

- A konstruktor használata tehát csak nagyon ritka esetben indokolt, akkor, ha mindenképpen új példányra van szükségünk, egyébként a valueOf használata jellemzően idő- és memóriamegtakarítást eredményez.

Primitív érték –> burkoló objektum konverzió II.

- A fentiekből egyrészt következik, hogy

```
Integer i = new Integer (25);
Integer j = new Integer (25);
i == j                                                    // false
```

- ..., az azonban már nem, hogy

```
Integer i = Integer.valueOf (25);
Integer j = Integer.valueOf (25);
i == j                                                    // may be true
```

- ..., mert a cache-elés stratégiája implementációs részlet, az objektumok referenciális egyenlősége nem követelmény.

Primitív érték <– burkoló objektum konverzió I.

- A numerikus típusok esetében az **nnnValue** metódusok a Number osztályban vannak definiálva:

```
byte byteValue ();
double doubleValue ();
float floatValue ();
int intValue ();
long longValue ();
short shortValue ();
```

- A fenti metódusokkal bármely, a Number-ből származó burkoló osztály példánya konvertálható bármely primitív típusú értékké.
- Ez a konverzió a típusoktól függően implicit (bővítő), vagy explicit (szűkítő) konverziónak felel meg, tehát az alábbi két kifejezés egyenértékű:

```
Double.valueOf (3.14).intValue ()                    // 3
(int) 3.14                                           // 3
```

Primitív érték <– burkoló objektum konverzió II.

- A Boolean és a Character esetében a fenti konverziót a **booleanValue** és a **charValue** metódusok végzik.

Szöveg –> primitív érték konverzió

- A szöveges reprezentációból a primitív értéket a statikus **parseNnn** metódusok állítják elő (a Character kivételével).

```
int i = Integer.parseInt ("8472");
```

- A Boolean kivételével, ha a megadott szöveg nem értelmezhető, a metódus NumberFormatException kivételt dob. A megengedett formátumok leírása az adott burkoló osztály dokumentációjában megtalálható.

```
int i;
try {
   i = Integer.parseInt ("hát szám ez?");
} catch (NumberFormatException e) {
   ...
}
```

- A Boolean esetében a parseBoolean értéke akkor true, ha a megadott String kisbetűssé alakítva "true".

Szöveg <– primitív érték konverzió

- A primitív érték szöveges reprezentációját a statikus **toString** metódusok állítják elő:

```
Integer.toString (42)                                    // "42"
```

- Az előállított String formátumának leírása az adott burkoló osztály dokumentációjában megtalálható.

Szöveg <–> burkoló objektum konverzió

- A szöveges reprezentációból burkoló objektumot a String paraméterű konstruktor, vagy a statikus **valueOf** metódus segítségével állíthatunk elő.

```
Integer i = new Integer ("47");
Integer j = Integer.valueOf ("30");
```

- A kettő közötti különbség ismét az, hogy míg a konstruktor mindig új példányt állít elő, a valueOf metódus ez megpróbálja elkerülni.
- Burkoló objektumból a szöveges reprezentációt a **toString** metódussal állíthatjuk elő, amelynek eredménye a statikus toString által adott eredménnyel egyezik meg.

```
Integer.valueOf (2007).toString ()                    // "2007"
```

Burkoló osztályok III.

- A numerikus típusok esetében definiált a **MIN_VALUE** és a **MAX_VALUE** konstans, amelyek az adott típus által ábrázolható minimális és maximális értéket tartalmazzák.
- A lebegőpontos típusoknál megtalálható még a **NaN**, a **NEGATIVE_INFINTY** és a **POSITIVE_INFINITY** is, amelyek a 0/0, -x/0, illetve x/0 értékét jelölik. Ezekhez kapcsolódnak még az isNaN, isInfinite metódusok is.
- A lebegőpontos típusok esetében a lebegőpontos érték és a megfelelő **egész értékként ábrázolt IEEE 754 reprezentáció** közötti konverzió is lehetséges a doubleTo(Raw)LongBits, longBitsToDouble, floatTo(Raw)IntBits, intBitsToFloat metódusok segítségével.

Burkoló osztályok IV.

- Az egész típusok burkoló osztályaiban megtalálhatók a **10-estől eltérő számrendszerben** ábrázolt szöveges reprezentációk konverziójára szolgáló parseNnn, toString, és valueOf metódusok is.

```
Long.parseLong ("cafebabe",16)                    // 3405691582
Integer.toString (42,7)                              // "60"
Long.valueOf ("alibaba",25)                       // 2653694035
```

Character I.

- A Java jelenleg a Unicode szabvány 4.0-ás verzióját támogatja.
- Az eredeti Unicode szabvány (4.0 előtti változatok) 16 biten ábrázolták a karaktereket, a kódpontok (Unicode Code Point) 0000 és FFFF közöttiek lehettek. Ezt a tartományt a Unicode **Basic Multilingual Plane**-nek (BMP) nevezi.
- A Unicode 4.0-ás változatától kezdődően a kódpontok 0000 és 10FFFF közöttiek lehetnek, az 10000 és 10FFFF közötti tartomány neve **Supplementary Characters** (kiegészítő karakterek).
- A Unicode-ban több ábrázolásmód létezik, a legfontosabbak az **UTF-8**, az **UTF-16**, illetve az **UTF-32**.

Character II.

- A BMP-beli kódpontokat az UTF-8 több karakterként ábrázolja, erre a célra a 00-FF tartományban egy úgynevezett **surrogate tartomány** van elkülönítve.
- Az UTF-16 a BMP-beli kódpontokat egyetlen karakterként ábrázolja, a Supplementary Characters tartománybelieket pedig két 16 bites értékként, az egyik a **High Surrogates** (D800-DBFF), a másik pedig a **Low Surrogates** tartományból (DC00-DFFF) való. A két érték együttes neve **surrogate pair**.
- A Java char típusa, és így a Character burkoló osztály is a 4.0 előtti Unicode szabványnak felel meg.
- Egy char érték egy Unicode kódpont UTF-16 ábrázolásban, így a BMP-beli kódpontokat egyetlen char érték írja le.
- A Supplementary Characters tartomány kódpontjait egy-egy UTF-16 surrogate pair, vagyis két char érték ír le.

Character III.

- A **kódpont** (code point) kifejezés egy, a 000000-10FFFF tartománybeli értéket jelent, a **kódegység** (code unit) pedig egy UTF-16 értéket jelent, amelyből egy ír le egy BMP-beli kódpontot, kettő pedig egy Supplementary Characters-beli kódpontot.
- A karakterekkel kapcsolatos API-kban azok a metódusok, amelyek char értéket vesznek át, nem támogatják a Supplementary Characters tartomány kódpontjait.
- Azon metódusok, amelyek int értéket vesznek át, a teljes Unicode 4.0 tartományt képesek kezelni.
- A karaktereket reprezentáló int értékek alsó 21 bitje írja le a kódpontot, a felső 11 bit pedig 0 kell legyen.
- További, Unicode-dal kapcsolatos információ: **http://www.unicode.org**.

Character IV.

- A Character osztály metódusai főleg karakterek besorolásával, konverziójával, illetve az integer reprezentációval kapcsolatos műveletekkel foglalkoznak.
- Integer reprezentáció:
  - charCount, codePointAt, codePointBefore, codePointCount, offsetByCodePoints, toChars, toCodePoint
- Besorolás:
  - isDigit, isIdentifierIgnorable, isISOControl, isJavaIdentifierPart, isJavaIdentifierStart, isLetter, isLetterOrDigit, isLowerCase, isSpaceChar, isTitleCase, isUnicodeIdentifierPart, isUnicodeIdentifierStart, isUpperCase, isWhitespace
- Egyéb Unicode vizsgálatok:
  - getDirectionality, getType, isDefined, isHighSurrogate, isLowSurrogate, isMirrored, isSupplementaryCodePoint, isSurrogatePair, isValidCodePoint

Character V.

- Konverziók:
  - digit, forDigit, getNumericValue, toLowerCase, toTitleCase, toUpperCase.
- A gyakorlatban legtöbbször BMP-beli karakterekkel dolgozunk, így a több kódegységként ábrázolt kódpontok által behozott komplexitással többnyire nem kell foglalkozni.

String I.

- A String osztály karakterfüzéreket tárol UTF-16 ábrázolásban.
- A **karakterfüzér**ek logikailag karakterek rendezett halmazai, ahol az egyes karakterek a füzérben elfoglalt pozíciójuk szerint **indexelhetők**, az első karakter indexe 0.
- A Unicode 4.0 bevezetése annyiban árnyalja a képet, hogy az UTF-16 ábrázolás miatt a Supplementary Characters tartománybeli kódpontok egy, két kódegységből álló surrogate pairként ábrázoltak.
- A String indexei kódegységekre vonatkoznak, így a Supplementary Characters tartománybeli kódpontok két indexet foglalnak el.
- A fenti probléma megoldására a String osztály számos, a surrogate pairek kezelését megvalósító metódust tartalmaz.

String II.

- A String osztály példányai **immutábilisak**, minden, a stringet módosító művelet eredménye egy új String példány.
- A String osztály két szempontból speciálisan viselkedik a többi Java osztályhoz képest, mindkettő a fordító által nyújtott "szintaktikai édesítőszer".
  - String példányt **nem csak a konstruktorai révén hozhatunk létre**, hanem string literálok leírásával is:

```
String alma = new String (new char[] { 'a','l','m','a' });// Ez megy,

String duma = "Sok a szöveg, kevés a lényeg.";        // ... de ez is.
```

  - A **+ operátor túlterhelt a stringekre nézve**, string konkatenációt és konverziót valósít meg. A legalább egy stringet tartalmazó "összeg" tagjait toString metódusuk, illetve primitív típusok esetén a burkoló osztályok toString metódusai segítségével stringgé konvertálja.

```
String s = "az érték " + 25 + ", de lehet, hogy " + 3.14 + ".";
String t = 25 + 3.14;                                  // Fordítási hiba
```

String III.

- Stringeket az alábbi módokon hozhatunk létre konstruktorok segítségével:
  - Másik string másolataként (erre nagyon ritkán lehet szükség):

```
String eredeti = "eredeti";
String másolat = new String (eredeti);
```

  - Unicode kódpontok (integerek) tömbjéből:

```
int[] codePoints = ...;
String s = new String (codePoints,0,codePoints.length);
```

  - Unicode kódegységek (karakterek) tömbjéből:

```
char[] codeUnits = ...;
String s = new String (codeUnits);
```

  - Byte-ok tömbjéből:

```
byte[] bytes = ...;
String s = new String (bytes, ...);
```

String IV.

- Fontos különbség a többi módszerhez képest, hogy amikor byte-ok tömbjéből hozunk létre stringet, nem Unicode kódpontokkal vagy kódegységekkel dolgozunk, hanem tetszőleges kódolásban értelmezendő egyszerű karakterkódokkal.
- A Javában a kódolásokat a **Charset** osztály példányai reprezentálják.
- A Charset egy **leképezés Unicode kódegységek és byteszekvenciák között**.
  - Ebből az következik, hogy nem csak egy-egy kódpontnak felel meg több kódegység, hanem egy-egy kódegységnek is több byte felelhet meg.
- A kódolásokra az IANA Charset Registry-ben megadott nevükkel hivatkozunk. Ha az általunk megadott néven nem létezik kódolás, az adott metódus UnsupportedEncodingException-t dob.

String V.

- A fenti konstruktoroknak megfelelő ellenirányú konverziókat biztosító metódusok is léteznek. Példaként az alábbi kódrészlet egy stringet először byte tömbbé, majd ismét stringgé alakít.

```
byte[] bytes;
String arvizturo;
try {
   bytes = "Árvíztűrő tükörfúrógép".getBytes ("ISO-8859-2");
   arvizturo = new String (bytes,"ISO-8859-2");
} catch (UnsupportedEncodingException e) {
   e.printStackTrace ();
   System.exit (-1);
}
```

- Ha a konverziós metódusoknak nem adunk meg kódolást, a plaform alapértelmezett kódolását használja.

String VI.

- Az **intern** metódus az adott string kanonikus példányát adja vissza.
  - Az azonos tartalmú stringeket a String osztály egy poolban tárolja. Az intern metódus meghívásakor, ha már létezik a poolban azonos string, a poolbeli string referenciáját kapjuk vissza, ha nincs, a string bekerül a poolba.
  - A string literálokra és a string értékű fordításidejű konstansokra a fordító mindig meghívja az intern metódust, így azok nem duplikálódnak.
- A String néhány általános információs és összehasonlító metódusa:

```
String s = "ez csak egy string";
s.length ()                                                    // 18
s.isEmpty ()                                                  // false
s.equals ("ez valami más")                                    // false
s.equalsIgnoreCase ("Ez csak egy String")                      // true
s.startsWith ("ez cs")                                         // true
s.endsWith ("ong")                                            // false
s.matches ("ez [a-z]+ egy .*");                                // true
```

Interning

```
package testPackage;

class Test {
     public static void main(String[] args) {
          String hello = "Hello";
          String lo = "lo";

          System.out.print((hello == "Hello") + " ");       // true
          System.out.print((Other.hello == hello) + " ");       // true
          System.out.print((other.Other.hello == hello) + " ");       // true
          System.out.print((hello == ("Hel"+"lo")) + " ");       // true
          System.out.print((hello == ("Hel"+lo)) + " ");       // false
          System.out.println(hello == ("Hel"+lo).intern());       // true
     }
}

class Other { static String hello = "Hello"; }

---


package other;

public class Other { static String hello = "Hello"; }
```

String VII.

- Stringekben lehetőség van egyes karakterekre, illetve részstringekre hivatkozni indexek alapján:

```
String s = "ez csak egy string";
s.charAt (3)                                    // 'c'
s.substring (8)                              // "egy string"
s.substring (3,7)                               // "csak"
```

- Karaktereket és részstringeket kereshetünk is:

```
String s = "ez csak egy string";
s.contains ("sak e")                            // true
s.indexOf ("ring")                               // 14
s.indexOf ("q")                                  // -1
s.lastIndexOf ("s")                              // 12
```

String VIII.

- A stringeket "szeletelhetjük", illetve egyéb módosításokat is végezhetünk rajtuk:

```
String s = "c:\\Program Files\\My Software";
String[] ss = s.split ("\\\\");                  // Reguláris kifejezés!
ss.length                                                          // 3
ss [1]                                                  // "Program Files"
"   some sparse    text ".trim ()                 // "some sparse    text"
String t = "Murder on the Orient Express";
t.toLowerCase ();
t                  // "Murder on the Orient Express" -> Immutábilis!!!
t = t.toUpperCase ();
t                                          // "MURDER ON THE ORIENT EXPRESS"
t.replace ("RIENT","UTLOOK")             // "MURDER ON THE OUTLOOK EXPRESS"
t.replaceAll ("[A-Z] ","?-")              // "MURDE? O? TH? ORIEN? EXPRES?"
t.replaceFirst ("O","I")                  // "MURDER IN THE ORIENT EXPRESS"
```

String IX.

- A String osztály statikus **valueOf** metódusa segítségével tetszőleges Java típus string reprezentációját megkaphatjuk:

```
public static String valueOf (boolean b);
public static String valueOf (char c);
public static String valueOf (double d);
public static String valueOf (float f);
public static String valueOf (int i);
public static String valueOf (long l);
public static String valueOf (Object obj);
```

- Ez azt is jelenti, hogy a String.valueOf metódusa túlterhelés révén "mindenevő".

StringBuilder I.

- A **StringBuilder** osztály a String mutábilis megfelelője.
- **append**, **delete** és **insert** metódusai révén tetszőleges stringet felépíthetünk segítségével.

```
StringBuilder sb = new StringBuilder ();
for (int i = 0;i < 100;i++) {
   sb.append (", " + i);
}
System.out.println (sb.substring (2));          // "0, 1, 2, ..., 99"
```

- A StringBuilder jelentősége a törlés és beszúrás lehetőségén kívül az, hogy lényegesen gyorsabb, mint ha a + operátort használnánk stringek összefűzésére, mivel nem jön létre minden egyes műveletnél újabb string példány.

StringBuilder II.

- Az append, illetve insert metódusok "mindenevők", vagyis minden Java típusra túl vannak terhelve.
  - Ennek az a következménye, hogy nem kell a hozzáfűzni kívánt értékeket először stringgé alakítani.

```
StringBuilder sb = new StringBuilder ();
sb.append ("Ali Baba és a  rabló.");
sb.insert (14,40);
sb.replace (0,3,"Gül");
sb.toString ()                              // "Gül Baba és a 40 rabló."
sb.reverse ();
sb.toString ()                              // ".ólbar 04 a sé abaB lüG"
```

- A **StringBuffer** a StringBuilder osztály "thread safe" változata, bővebben lásd a szálakról szóló előadást.

Math és StrictMath

- A **Math** osztály statikus metódusok gyűjteménye, amelyek gyakran használt matematikai konstansokat, műveleteket, függvényeket valósítanak meg.
- Konstansok:
  - E, PI
- Metódusok
  - abs, ceil, floor, min, max, round, rint, signum, copySign
  - sqrt, cbrt
  - sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, toDegrees, toRadians
  - exp, expm1, pow, log, log10, log1p
  - hypot, IEEERemainder, getExponent, nextAfter, nextUp, scalb, ulp, random
- A **StrictMath** a fenti műveletek a Freely Distributable Math Library **(fdlibm)** "IEEE 754 core function" verziójának megfelelő implementációja.

System

- A **System** osztály nem példányosítható, csak statikus tagjai vannak, melyek a Javát futtató operációs rendszerrel való kapcsolatot segítik.
- Alacsonyszintű VM műveletek:
  - exit, arraycopy, gc, runFinalization, load, loadLibrary, mapLibraryName
- Java Security (lásd a Java Security előadásban):
  - getSecurityManager, setSecurityManager
- Standard I/O:
  - **err, in, out mezők**, setErr, setIn, setOut, console (lásd az Java I/O előadásban).
- Környezeti változók, rendszertulajdonságok (lásd később):
  - getenv, getProperties, getProperty, setProperties, setProperty
- Idő:
  - currentTimeMillis, nanoTime

ProcessBuilder és Process I.

- A **ProcessBuilder** osztály feladata külső operációs rendszer folyamatok indítása és futásuk felügyelete.
- Az alábbi példa a ProcessBuilder használatának lépéseit mutatja be:

```
ProcessBuilder pb = new ProcessBuilder ();
pb.command ("awk 'BEGIN { c = 0 } { c = c + 1 } END { print c }'");
pb.directory ("/data/files");
pb.redirectErrorStream (false);
Process p = pb.start ();
```

- A start metódus eredménye egy Process objektum, amely a futó folyamattal való kommunikációt teszi lehetővé.

ProcessBuilder és Process II.

- A **Process** objektum segítségével a folyamatot leállíthatjuk (destroy), megvárhatjuk a futása végét (waitFor), illetve lekérdezhetjük az exit kódját (exitValue).
- A getErrorStream, getInputStream, illetve getOutputStream metódusok segítségével a folyamat standard error, input, illetve output csatornáit kérhetjük le, így kommunikálhatunk vele.