3. Egy kis szárazanyag

Minden nyelv tartalmaz olyan részeket, amelyeket egyszeren *tudni* kell - ezekre a részekre szokták mondani: *száraz az anyag*. De tudjuk, hogy nincs olyan száraz anyag, amit ne lehetne egy kis sörrel könnyebben emészthetvé tenni... készítsünk hát magunk mellé egy üveggel a jobbik fajtából, és feküdjünk neki az ivásnak... illetve az olvasásnak... :)

3.1. Azonosítók

A programozási nyelvek nyelvtana általában négy fogalommal jellemezhet:

- *változók*, amelyek a típusuknak megfelel értékeket tárolnak, például pohárban vagy üvegben folyadékokat tárolunk.
- *literálok*, amelyek önmaguk hordozzák a típusukat, ezek általában a való életbl (és a matematikából) vett fogalmak, például az egy korsó sör esetén az *"egy korsó"* egy literál, közös megegyezés alapján fél liternyi folyadékot jelent - ahogy a *"fél liter"* is egy literál, ennek definícióját most nem részlezetném... :)
- *operátorok*, amelyek mveleteket határoznak meg az elz két kategória elemei között, tehát ha egy vödörbe öntünk három üveg sört, akkor *összege ztük* a három üvegnyi sör tartalmát, amelyet *beletettünk* egy vödörbe.
- *vezérl szerkezetek*, amelyek meghatározzák a program sorainak végrehajtási módjait, vagyis ha addig akarunk sört inni, amíg menni tudunk, akkor erre például ciklust kell írnunk, amely eddig ismételgeti a *"sörivás"* mveletet amíg menni tudunk. Tapasztalt játékosok a ciklus kilépési feltételénél azt is megvizsgálják, hogy van-e még pénzük egy újabb körre... :)

A változókat és a típusokat is el kell neveznünk, hogy hivatkozni tudjunk rájuk, vagyis pontosan *azonosítani* tudjuk ket - ezért ket közös néven *azonosítónak* nevezzük. Mivel Java nyelven az osztályok egyben típusok is, ezért az osztályok, a típusok és a változók elnevezése jobbára azonos szabályok szerint kell történjen:

- Gyakorlatilag bármilyen hosszú *Unicode*bet és szám kombinációt használhatunk, de:
    - nem kezddhet számmal
    - nem tartalmazhat írásjeleket - kivéve a $ (dollár) és az _ (aláhúzás) jelet
    - nem lehet benne se szóköz, se egyéb speciális karakter ('^', '°', stb.)
- Általános irányelv, hogy a lehetség ellenére a dollár jelet nem teszünk a változó nevébe, és nem kezdjük aláhúzás jellel.

Helyes azonosítók:

Helyes azonosítók

```
sor
Sör
sÖr
ser
_sör
sör2
egy_sör
$sör
```

Hibás azonosítók:

Hibás azonosítók

```
2sör
egy sör
kis-sör
super sör
sör:
Sört!
```

A Java nyelv kulcsszavait nem használhatjuk azonosítóként, ezek a következk:

Foglalt kulcsszavak

```
abstract        boolean         break           byte            case
catch           char            class           const           continue
default         do              double          else            enum
extends         false           final           finally         float
for             goto            if              implements      import
instanceof      int             interface       long            native
new             package         private         protected       public
return          short           static          super           switch
synchronized    this            throw           throws          transient
true            try             void            volatile        while
```

A Sun elkövette azt a hibát, hogy az Java 5.0 verziójában egy új kulcsszót hozott be: ez a kulcszó az *enum*. Ez önmagában nem probléma, de ezt nem foglalta le kulcsszóként a nyelv megelz verzióiban, mint a *const* és a *goto* kulcsszót, amelyeket a nyelv **azóta sem használ**. Így az 1.5.0 JRE alatt ugyan futottak azon régebbi programok, amelyek ezt az *enum* kulcsszót azonosítóként használták, de az 1.5.0 JDK alatt nem lehetett ezeket lefordítani, mert a fordító hibát jelzett.

3.2. Típusok

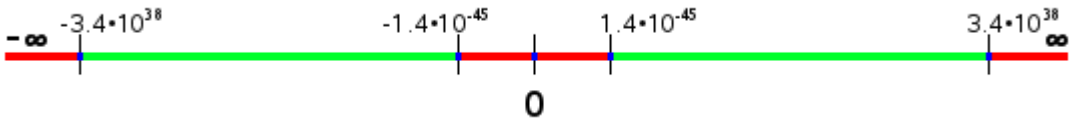
A Java nyelv - hasonlóan a Pascal és a C/C++ nyelvhez - alapveten bináris számítógépeken futó programok készítésére használatos: egyszerbb a mi fogalmainkat használva megírni egy programot, amelyet aztán lefordítunk a számítógép nyelvére (*gépi kód*), amelyet ugyan meg lehet tanulni, de hatékonyabban tudunk felhasználói igényeket kielégíteni, ha magas szinten programozunk. A bináris számítógépek hatékony kihasználásához nem feltétlen szükséges ismernünk a számítógép magját alkotó CPU által kezelhet adattípusokat, ezt megteszik helyettünk az említett programnyelvek, mégpedig a primitív típusok által.

3.2.1. Primitív típusok

Egy bináris (más néven digitális) számítógép számára minden a világon két számból épül fel: **0** és **1**, amelyeket bináris számnak (**b**inary dig**it**, vagyis bit) nevezünk. Bármi, amit a számítógépünkön találunk, az mind bináris számok különféle kombinációja és sorozata, s a kettes számrendszerírja le a kombinációk és sorozatok kódolását. A különféle hosszúságú kettes számrendszerben kódolt számoknak a szakma neveket adott, nézzük át ket (a teljesség igénye nélküli):

- ***bit (boolean)*** - *bit*: a legkisebb egység, értéke lehet **0** vagy **1**, illetve igaz (***true***) vagy hamis (***false***).
- ***byte*** - *bájt*: 8 bináris szám által reprezentált szám, értéktartománya 2^8, vagyis *0 – 255* vagy *-128 – 127*.
- ***char*** - *karakter*: egy bájt által reprezentált karakter, kódolása általában *ASCII*.
- ***word*** - *egy gépi szó*: két bájt, 16 bit által reprezentált szám, értéktartománya 2^{16}, vagyis *0 – 65535* vagy *-32768 – 32767*.
- ***integer*** - *egész szám*: egy vagy több gépi szó által reprezentált szám, értéktartománya tipikusan 2^{32}, vagyis *0 – 4 milliárd* vagy *-2 milliárd – 2 milliárd.*
- ***long (integer)*** - *nagy egész szám*: általában négy gépi szó által reprezentált szám, értéktartománya általában 2^{64}, vagyis *0 – 16 trillió* vagy *-8 trillió – 8 trillió.*
- ***real (float)*** - *valós szám*: két gépi szóból álló egyszer (lebegpontos) valós szám, értéktartománya eljel nélkül $\sim 1.4^{-45}$ – $\sim 3.4^{38}$.
- ***double*** - *nagy valós szám*: négy gépi szóból álló (lebegpontos) valós szám, értéktartománya eljel nélkül: $\sim 4.9^{-324}$ – $\sim 1.8^{308}$.

A lebegpontos számok használata során ismernünk kell a trükköt, hiszen nyilvánvaló kell legyen, hogy 32 bitbe nem fér el minden valós szám 1.4^{-45} és 3.4^{38} között. A megoldás egyszer: az egész számokat tároló megoldások a szám összes számjegyét tárolják, míg az utóbbi kett - valós szám tárolására használt számkódolási eljárás - kerekít. A megadott tartományon belül tetszlegesen nagy számot tudunk tárolni, de csak 7-8 számjegy pontossággal. További trükk, hogy a nulla értéket csak megközelíteni tudjuk, a legkisebb érték és a nulla között mindig lesz egy tartomány, amelyet nem tudunk használni. Az alábbi számegyenesre tekintve láthatjuk, hogy a kék pontok jelentik a kódolás határértékeit, a zöld részen tudunk számokat tárolni 7-8 számjegy pontossággal, és a piros területeket már nem tudjuk tárolni:



A Java nyelvben a felsorolás sorrendjében a *boolean*, a *byte*, a *char*, a *short*, az *int*, a *long*, a *float* és a *double* használatos, ezeket nevezzük ***primitív típusnak***. Fontos tudni, hogy nincs eljel nélküli típus, az összes primitív típus eljeles szám, nézzünk példát ezen típusok használatára.

Vegyük el a NetBeans környezetben a már létrehozott *Kocsma* projektet, és keressük meg benne a *kocsma* csomagot, majd azon belül a *Main*osztályt, majd az alábbi programot írjuk bele:

Main.java

```
package kocsma;

public class Main
{

  public static void main(String[] args)
  {
    // Ide fogjuk írni az alábbi programrészleteket
  }
}
```

A // kezdet sor helyére kell írni az alábbiakban megemlített programrészleteket, majd az *F6* gomb megnyomásával futtatni tudjul a (helyesen megírt) programot.

A boolean

A *boolean* típust akkor használjuk, ha egy eldönthet kérdésekre adott válasz csak **igen/igaz** vagy **nem/hamis** lehet, a *boolean* ugyanis nem képes a *talán* vagy a *nem tudom* válaszokat értelmezni. A C/C++ nyelvektl eltéren egy *boolean* típusú változó értéke csak a *true* vagy a *false*értéket veheti fel, a számszeren jelzett értékeket nem képes értelmezni.

Java

```
boolean igaz = true;
boolean hamis = false;
boolean válasz;
válasz = igaz;
```

Vannak olyan mveletek, amelyek természetszerleg *boolean*értéket adnak eredményül:

Java

```
válasz = 9 < 3;
System.out.println(válasz);
válasz = 3 < 10;
System.out.println(válasz);
```

Az els esetben *false* (hamis) eredményt kapunk, hiszen a kilenc nem kisebb, mint a három; a második esetben az eredmény *true* (igaz), hiszen a három kisebb, mint a tíz.

A példában említett *true* és *false* literál, vagyis a nyelvbe épített és értékkel bíró fogalom. Láthatunk továbbá több deklarációt és értékadásokat is - ezekrl késbb több szót ejtek.

A byte

A *byte* típust akkor célszer használnunk, ha nulla közeli egész számokkal számolunk, hiszen értéktartománya mindössze a -128 és a 127 közötti számokra érvényes:

Java

```
byte szám;
szám = 12;
szám = -12;
szám = -128;
szám = 127;
```

A példában láthatunk egy deklarációt és négy értékadást, ahol a *szám*nev változónak adunk más-más értéket, s itt már szám literálokat használunk. Fontos tudni, hogy a primitív típusok képesek a csöndes túlcsordulásra:

Java

```
byte szám;
szám = 127;
szám++;
System.out.println(szám);
```

A rövidke programrészlet eredménye *-128* lesz, nem pedig a matematikailag várható *128*. Ennek oka, hogy a változó *túlcsordult*, túllépte az értékkészlete egyik határát, és ezért értékül az értékkészletének másik határértékét veszi fel. Nagyon oda kell figyelnünk erre a tulajdonságra, mivel a programunk mindezt csöndben teszi, legtöbbször igen keserves és hosszú munkával tudjuk csak megkeresni a hiba okát.

A short, az int és a long

A *short*, az *int* és a *long* teljesen azonos módon használható, mint a *byte*, a különbség mindössze az értéktartományban van (illetve abban, hogy a *long* típus literál használatához a szám mögé egy **l**karaktert kell írnunk):

- *short*: -32768 – 32768
- *int*: –2147483648 – 2147483647
- *long*: -9223372036854775808**l** – 9223372036854775807**l**

Ha nem adunk meg egy egész szám literál után módosító karaktert, akkor a nyelv szabályai szerint a fordító megpróbálja konvertálni arra a típusra, ami a használt változó típusa, egyéb esetben a literál típusa *int*lesz.

A char

A többi nyelvhez hasonlóan a *char* egy öszvér adattípus, alapveten számokat tartalmaz, de ha az értékét kiírjuk, akkor karaktereket kapunk eredményül. A Java nyelv úttör volt a tekintetben, hogy a *char* típus alapja nem a 8 bites *byte*, hanem az 32 bites *integer*, hiszen a Java nem ASCII alapú, hanem UTF-8 az alapértelmezett karakter kódolása, mivel ismert világ összes karaktere nem fér el 8 biten.

Java

```
char bet = 64;
System.out.print(bet);
bet = '\n';
System.out.print(bet);
bet = 'a';
System.out.print(bet);
bet = '\t';
System.out.print(bet);
bet = '\\';
System.out.print(bet);
bet = '\'';
System.out.print(bet);
bet = '"';
System.out.print(bet);
bet = '\070';
System.out.print(bet);
bet = '\u003F';
System.out.print(bet);
```

Nézzük meg ezt a gyakorlatban:

Eredményül az alábbi (ránézésre értelmetlen) karakter halmazt kapjuk:

Eredmény

```
@
a       \'"8?
```

Lássuk a dekódolást:

- A @ karakter UTF-8 (és ASCII) kódja a 64 szám, ezért a @ az els kiírt karakter.
- A \n karakter az új sor kódja, ez nem látható karakter, viszont a (szintén nem látható) kurzort a következ sorba teszi.
- A következ sor elején kiírt a bet egyértelm kell legyen, hiszen ezt adtuk értékül a bet nev változónak.
- A negyedik kiírt karakter ismét egy nem látható karakter (\t), amely egy tabulátor pozícióval jobbra mozgatja a kurzort.
- Biztos felmerült, hogy a \ karaktert hogy tudjuk kiírni: egyszeren a \\ karaktert kell használnunk. A \ (*backslash* vagy *vissza-per* jel) egy speciális karakter, amely az t követ karakternek is speciális jelentést ad.

- A szimpla idézjel kiírásához a \' karaktereket kell használnunk.
- A dupla idézjel kiírható, ha szimpla idézjelek között van.
- A \070 kódolás oktális formátumú számot hordoz, a 070 oktális szám azonos az 56 decimális számmal, amely az UTF-8 kódrendszerben a '8' karakternek felel meg.
- A \u003F kódolás hexadecimális formátumú számot jelent, a 003F szám azonos a 63 decimális számmal, amelynek a '?' karakter felel meg.

Mint láthatjuk, eléggé változatos módon tudunk karaktereket megadni, azonban a *char* ketts természetét az is meghatározza, hogy tudunk vele mveleteket végezni:

Java

```
char bet = 64;
bet++;
System.out.print(bet);
bet += 0;
System.out.print(bet);
bet += '0';
System.out.print(bet);
```

Az eredmény az AAq karaktersorozat lesz, hiszen a 64 az '@' karaktert kódolja, amihez ha hozzáadunk egyet, akkor 'A' bet lesz, amit a 65 kódol. Ha ehhez hozzáadunk nullát (0), akkor az eredmény továbbra is 65 marad, de ha karakteresen adunk hozzá nullát ('0'), amelynek a számszer értéke 48, akkor az eredmény 113 lesz (mivel 65+48 az 113), amely szám a 'q' karaktert kódolja.

A karakter literálok két szimpla idézjel közötti karakterek. Általában egy karakter, de ha az els karakter visszaper jel (\), akkor több karakter is lehet a két idézjel között. Érdemes megtanulni a speciális karakter literálok használatát, mivel használatuk sok esetben szükséges a szövegfeldolgozáshoz.

A float és a double

A valós számok használata akkor kerül eltérbe, amikor olyan számolásokat végzünk, amelyek nem végezhetk el könnyedén egész számokon. A valós számok használata lassíthatja a program mködését, mivel ezek kezelése néhány processzoron nincsen kellen gyorsítva, ezért csak akkor használjunk valós számokat, amikor feltétlenül szükséges. Nézzük a *float* típus használatát:

Java

```
float szám = 0;
System.out.println(szám);
szám = 1/3;
System.out.println(szám);
szám = 1f/3f;
System.out.println(szám);
szám = -1*0f;
System.out.println(szám);
System.out.println(szám == 0f);
szám = 10000000000000000000000000000000000000f;
System.out.println(szám);
szám = 1.0E38f;
System.out.println(szám);
```

Eredményül nem pont azt kapjuk, amit várnánk:

Eredmény

```
0.0
0.0
0.33333334
-0.0
true
1.0E38
1.0E38
```

Az els 0.0 érthet, hiszen ezt adtuk értékül a *szám* nev változónak. A második 0.0 azonban elgondolkodtató: az egyharmad értéke nem szabadna nulla legyen. Ennek oka az, hogy a Java nyelv két *egész számot* lát: elosztja az egyet hárommal: hányszor van meg egyben a három? Ugye egyszer sem, ezért kaptunk eredményül nullát. A problémát úgy tudjuk kikerülni, hogy a szám mögé írt **f** betvel mondjuk meg, hogy ez a szám nem egész szám, hanem *float*. A 1f/3f mvelet eredménye már 0.33333334, mivel a float csak nyolc számjegy pontosságú, azt követen nincs több értékes számjegy. A -0.0 eredmény a számábrázolás érdekessége, ugyanis van plusz nulla és mínusz nulla is, és ahogy a következ sorban látjuk: a kett egyenl egymással. A nagy számokat megadhatjuk a számjegyekkel is, de rövidebb a normálforma: $1.0 \cdot 10^{38}$, amely a program forrásában 1.0E38 formán kódolódik.

A *double* használata annyiban tér el a *float* használatától, hogy az **f** karakter helyett **d** karaktert kell használnunk a literál megadásakor (amelyet akár el is hagyhatunk, hiszen a *double*típus az alapértelmezett lebegpontos típus).

3.2.2. Primitív burkoló osztályok (primitive wrappers)

Mindegyik primitív típusnak van egy burkoló osztálya, amely már használható objektum orientált módon, s képes önmagán néhány egyszer (nem matematikai) mveletet elvégezni.

A Boolean osztály

A *boolean* típushoz tartozó burkoló osztály, amelynek leginkább azt a tudását használjuk fel, hogy szövegbl képes önmagát legyártani:

Java

```
Boolean válasz= new Boolean("tRuE");
System.out.println(válasz);
```

A paraméterben átadott szöveg hatására a *válasz* nev változó értéke *true* lesz. Akkor és csak akkor lesz az eredmény *true*, ha az átadott szöveg karakterhelyesen *"true"*, azonban a kis és nagybet nincs megkülönböztetve (*case insensitive*). Ugyanígy használható a *valueOf* metódus is, amely egy picit gyorsabb is:

Java

```
Boolean válasz= Boolean.valueOf("tRuE");
System.out.println(válasz);
```

Az eredmény jelen esetben is *true*lesz. Az 1.5 Java verziótól használhatjuk az automatikus konverziót is:

Java

```
Boolean válasz = true;
boolean érték = válasz;
```

Az 1.5 verzió eltt ezt csak hosszabban tudtuk leírni:

Java

```
Boolean válasz = Boolean.valueOf(true);
boolean érték = válasz.booleanValue();
```

A Byte, a Short, az Integer, a Long, a Float és a Double osztály

Ezen osztályok mind a *Number* osztályból származnak, amely a számokat kezel osztályok se. A *Number*stl örökölt kényszer szerint van hat metódusuk, amely a burkoló osztály által hordozott szám primitív típusát adja vissza:

Java

```
Double szám = new Double(3.141592653589793);
System.out.println(szám.doubleValue());
System.out.println(szám.floatValue());
System.out.println(szám.longValue());
System.out.println(szám.intValue());
System.out.println(szám.shortValue());
System.out.println(szám.byteValue());

szám = Double.parseDouble("2.718281828459045");
System.out.println(szám.doubleValue());
System.out.println(szám.floatValue());
System.out.println(szám.longValue());
System.out.println(szám.intValue());
System.out.println(szám.shortValue());
System.out.println(szám.byteValue());
```

A fenti programrészlet eredménye:

Eredmény

```
3.141592653589793
3.1415927
3
3
3
3
2.718281828459045
2.7182817
2
2
2
2
```

Mint látható, a valós számot hordozó *Double* esetén az egész számra való alakítás is megtörténik, ha egész típusú primitív típust kérek el, azonban az átalakítás során csonkolás történik (eltnik a tizedespont mögötti számsor), nem pedig kerekítés!

Ezek az osztályok már több mveletet is lehetvé tesznek, nekünk azonban egyelre elég a *valueOf*ismerete, amely szöveges formából alakít számmá, akárcsak a Boolean esetén:

Java

```
szám = Double.valueOf("-2");
System.out.println(szám);
szám = Double.valueOf("-2.2E3");
System.out.println(szám);
szám = Double.valueOf("-2.2E3000");
System.out.println(szám);
System.out.println(szám.isInfinite());
```

Ennek eredménye:

Eredmény

```
-2.0
-2200.0
-Infinity
true
```

Fontos, hogy a valós típusoknak lehet negatív és pozitív végtelen értéke - ha túl nagy számot szeretnénk szöveges formából átalakítani vagy a mvelet eredménye túl nagy, ezt lekérdezhetjük az *isInfinite* metódussal.

A Character osztály

A karakter ismét kilóg a sorból, hiszen a célja alapveten nem a számolás, hanem karakterek ábrázolása.

Java

```
Character karakter = new Character('a');
System.out.println(karakter);
System.out.println(Character.isDigit('0'));
System.out.println(Character.isDigit('a'));
System.out.println(Character.isWhitespace('\t'));
System.out.println(Character.isWhitespace('a'));
System.out.println(Character.isISOControl('\t'));
System.out.println(Character.isISOControl('a'));
System.out.println(Character.toLowerCase('A'));
System.out.println(Character.toUpperCase('a'));
```

A futás eredménye:

Eredmény

```
a
true
false
true
false
true
false
a
A
```

A Character osztálynak van egy rakás statikus metódusa, amely a megadott karakterrl döntést hoz. A példában rákérdeztünk párosával arra, hogy az átadott karakter szám-e, helykitölt karakter-e, valamit ISO vezérlkarakter-e. Az utolsó két metódus pedig konvertálta az átadott karaktert kis-, illetve nagybetsre. Ezen túl sok egyéb metódust megtalálunk a Character osztályban (írásirány, tükörírás, stb.), ezeket nem részletezném.

Közös konstansok

A Boolean osztályt leszámítva az összes burkoló osztálynak van négy hasznos konstansa:

- **TYPE**, amely a primitív típust adja vissza
- **SIZE**, amely a primitív típus által elfoglalt bitek számát adja vissza
- **MAX_VALUE**, amely a legnagyobb ábrázolható értéket hordozza
- **MIN_VALUE**, amely a legkisebb ábrázolható értéket hordozza, lebegpontos típus esetén a legkisebb ábrázolható pozitív számot

Lássunk rá példát:

Java

```
System.out.println(Byte.TYPE);
System.out.println(Byte.MAX_VALUE);
System.out.println(Byte.MIN_VALUE);
System.out.println(Byte.SIZE);

System.out.println(Short.TYPE);
System.out.println(Short.MAX_VALUE);
System.out.println(Short.MIN_VALUE);
System.out.println(Short.SIZE);

System.out.println(Integer.TYPE);
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);
System.out.println(Integer.SIZE);

System.out.println(Long.TYPE);
System.out.println(Long.MAX_VALUE);
System.out.println(Long.MIN_VALUE);
System.out.println(Long.SIZE);

System.out.println(Float.TYPE);
System.out.println(Float.MAX_VALUE);
System.out.println(Float.MIN_VALUE);
System.out.println(Float.SIZE);

System.out.println(Double.TYPE);
System.out.println(Double.MAX_VALUE);
System.out.println(Double.MIN_VALUE);
System.out.println(Double.SIZE);

System.out.println(Character.TYPE);
System.out.println(Character.MAX_VALUE);
System.out.println(Character.MIN_VALUE);
System.out.println(Character.SIZE);
```

Az eredményt mindenki nézze meg a saját gépén... :)

3.3. Operátorok

> *Mit sem ér egy üveg sör, ha nem tudunk vele semmit kezdeni.*

A deklarált változóink nem érnek sokat, ha nem tudunk rajtuk mveleteket végezni. Mveletet csak és kizárólag primitív típusokon, illetve típusok között tudunk végezni, és a mveletet az operátor vagy az operátorok határozzák meg. Nem kell megijedni, egyszeren arról van szó, hogy az összeadás mveletét a + jel, mint operátor határozza meg.

3.3.1. Értékadás

Eddigi programjaink során már találkoztunk az értékadás mveletével:

Java

```
int szám;
szám = 2;
```

Ezt úgy kell olvasnunk, hogy **legyen** egy *int* **típusú** és *szám* **nev** változónk, majd a *szám* nev változó értéke **legyen egyenl** *kettvel*.

3.3.2. Egyszer matematikai mveletek

A primitív típusú változóink szinte kivétel számokat tárolnak, ezért mindegyiken képesek vagyunk matematikai mveleteket végezni. Lássuk, miképp tudjuk a matematikai alapmveleteket Java nyelven elvégeztetni.

Eljelváltás

A legegyszerbb operátor az eljelváltás:

Java

```
int szám = 2;
szám = -szám;
System.out.println(szám);
szám = -szám;
System.out.println(szám);
```

Az eljelváltás a tipikus példája a prefix típusú operátornak, hiszen az operátort követi az operandus.

Összeadás

A négy alapvet matematikai mvelet közül a leginkább használt mvelet az összeadás, Java nyelvben is pontosan úgy mködik, mintha papírra vetnénk:

Java

```
int szám;
szám = 3 + 4;
szám = szám + 5;
szám = szám + 1;
System.out.println(szám);
```

A *szám = 3 + 4* utasítás úgy olvasandó, hogy a *szám* változó értéke legyen egyenl a három meg a négy értékével, vagyis a *szám* értéke hét lesz. A *szám = szám + 5* utasítás már feltételez **elz értéket** a *szám* változóban (ami jelen esetben 7), s a jelentése annyi, hogy a *szám* változó értéke legyen egyenl a *szám* változó elz értéke meg öt értékével, vagyis a *szám* változóban a 12 lesz eredményképpen. A következ utasítás hasonlóképpen értelmezhet, azt jelenti, hogy a *szám*változó értékét megnöveljük egyel. A kiírt eredmény 13 lesz.

Kivonás

Azonos módon értelmezhet, mint az összeadás:

Java

```
int szám;
szám = 3 - 4;
szám = szám - 5;
szám = szám - 1;
System.out.println(szám);
```

Az eredmény pedig -7 lesz.

Szorzás

A szorzás is úgy mködik, mint az elz két mvelet, csak a mveleti jel változik:

Java

```
int szám;
szám = 3 * 4;
szám = szám * 5;
szám = szám * 1;
System.out.println(szám);
```

Az eredmény 60 lesz, ahogy az sejthet.

Osztás

Az osztás kilóg az elz három mvelet közül, egy kicsit speciális a helyzete. Az egyik buktató - amirl már volt szó, hogy ha egész számok az operandusok, akkor egész osztás fog történni:

Java

```
int szám;
szám = 30 / 4;
szám = szám / 5;
szám = szám / 1;
System.out.println(szám);
```

Normál esetben az eredmény másfél lenne, de az egész osztások miatt az 30/4 eredménye 7 lesz, ezt követen a 7/5 eredménye pedig 1, aztán az 1/1 természetesen 1. Ha a valós eredmény érdekel minket, akkor valós számokkal kell számolnunk:

Java

```
double szám;
szám = 30 / 4;
szám = szám / 5;
szám = szám / 1;
System.out.println(szám);
```

Az eredményül kapott 1.4 egy kissé gyanús lehet matematikában jártas egyéneknek. A probléma gyökere ott van, hogy a 30/4 még mindig egész osztás marad, attól függetlenül, hogy a *szám* változó típusa valós. Ennek oka, hogy a Java nyelvben – és más nyelvekben is – az egyenlség jel jobb oldalán kezddik a kifejezés végrehajtása, mégpedig balról jobbra. Mivel a 30 egy egész szám és a 4 is egy egész szám, ezért a két számon értelmezett osztás mvelet egész osztás lesz: a *szám* változóba eredményül 7 kerül. A következ sorban az utasítás szerint a *szám* változó értékét el kell osztani öttel. Itt a *szám* változó típusa okán már valós osztás lesz, és a 7/5 eredménye az 1.4. A hibát úgy tudjuk kikerülni, hogy jelezzük a 30 és a 4 valós voltát (illetve a többi szám valós voltát is):

Java

```
double szám;
szám = 30.0 / 4.0;
szám = szám / 5.0;
szám = szám / 1.0;
System.out.println(szám);
```

Eredményül most már másfelet fogunk kapni. A nullával való osztásban is vannak különbségek. Ha valós számot osztunk nullával, akkor eredményül **végte lent**(Infinity) kapunk:

Java

```
double szám;
szám = 30.0 / 4.0;
szám = szám / 5.0;
szám = szám / 0;
System.out.println(szám);
```

Ellenben egész szám esetén a program futása megszakad:

Java

```
int szám;
szám = 30 / 4;
szám = szám / 5;
szám = szám / 0;
System.out.println(szám);
```

Egy kivétel (Exception) keletkezik, hogy nullával próbáltunk meg osztani:

Eredmény

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at kocsma.Main.main(Main.java:11)
```

A kivételekrl majd később, annyit jegyezzünk meg, hogy ha egész számokkal osztunk, akkor annak súlyos következményei lehetnek.

Maradékképzés

Az egész osztás párja a maradékképzés, amikor nem arra vagyunk kíváncsiak az osztás során, hogy mennyi az eredmény, hanem arra, hogy mennyi a maradék:

Java

```
int szám;
szám = 30 % 4;
System.out.println(szám);
```

Az eredményül kapott 2 nem okozhat meglepetést, ha vissza tudunk emlékezni az általános iskola második osztályában tanultakra: harmincban a négy megvan hétszer, maradék kett. Mivel a maradékképzés is osztás, itt is kaphatunk kivételt, ha nullával szeretnénk osztani: például *30 % 0*. Néhány nyelvben a maradékképzés csak egész számokra használható, ám Java nyelvben a mvelet elvégezhet valós számokon is:

Java

```
double szám;
szám = 30.0 % 4.1;
System.out.println(szám);
szám = 30.0 / 4.1;
System.out.println(szám);
szám = 7.0*4.1;
System.out.println(szám);
szám = 30.0 - szám;
System.out.println(szám);
```

Eredményül az alábbi számokat kapjuk:

Eredmény

```
1.3000000000000025
7.317073170731708
28.699999999999996
1.3000000000000043
```

A megoldás egyszer: 30.0 / 4.1 az egy valós számot ad eredményül. Ezt csonkolva hetet kapunk, amelyet ha visszaszorzunk a 4.1 számmal, elvileg 28.7 lesz az eredmény (a példában jól látszik a valós számábrázolás pontatlansága!). Ha a 30.0 számból kivonjuk a 28.7-et, akkor kapunk 1.3-at, mint maradék. El nem tudom képzelni, hogy ezt hol lehet kihasználni... :)

Növelés és csökkentés

Gyakori feladat, hogy egy változó értékét növeljük vagy csökkentsük egyel. Ez normál esetben így nézne ki:

Java

```
int szám = 2;
szám = szám + 1;
System.out.println(szám);
szám = szám - 1;
System.out.println(szám);
```

Mivel ez a forma hosszú és összetett, a C nyelvbl kölcsönzött ++ és -- operátort tudjuk használni, azonban ezek lehetnek prefix és postfix operátorok is:

Java

```
int szám = 2;
System.out.println(szám++);
System.out.println(++szám);
System.out.println(szám--);
System.out.println(--szám);
```

Az eredmény:

Eredmény

```
2
4
4
2
```

Ami nem meglep, hiszen az els esetben a kiírás **után** növekedett a változó értéke, a második esetben a kiírás **eltt**, aztán a harmadik esetben a kiírás **után** csökkentettük a változó értékét, majd pedig a kiírás **eltt**.

3.3.3. Relációs mveletek

Két érték összehasonlítása relációs jelekkel történik, s eredményképpen *boolean* típust kapunk, amely lehet *igaz* vagy *hamis*. Szaladjunk gyorsan át ezeken a mveleteken:

Java

```
int szám = 10;
int másikszám = 20;
System.out.println(szám < másikszám);
System.out.println(szám <= másikszám);
System.out.println(szám > másikszám);
System.out.println(szám >= másikszám);
System.out.println(szám == másikszám);
System.out.println(szám != másikszám);
```

A relációs jelek jelentése sorban:

- a *szám* kisebb, mint a *másikszám*? Igaz, kisebb
- a *szám* kisebb vagy egyenl, mint a *másikszám*? Igaz, kisebb vagy egyenl
- a *szám* nagyobb, mint a *másikszám*? Hamis, nem nagyobb
- a *szám* nagyobb vagy egyenl, mint a *másikszám*? Hamis, nem nagyobb vagy egyenl
- a *szám* egyenl a *másikszám*mal? Hamis, nem egyenl
- a *szám* nem egyenl a *másikszám*mal? Igaz, nem egyenl

Fontos, hogy csak ebben a formában tudjuk használni a relációs jeleket, a '<=' helyett a '=<' nem használható, illetve a '!=' helyett se tudjuk használni a '<>' jelet.

3.3.4. Logikai mveletek

A relációs mveletekkel nem tudjuk kifejezni azt az egyszer matematikai képletet, hogy *4 < szám < 10*, amely akkor igaz, ha a *szám*értéke nagyobb, mint négy, és kisebb, mint tíz:

Java

```
int szám = 10;
System.out.println(4 < szám < 10);
```

Ha mégis megpróbáljuk, akkor fordítási hibát kapunk eredményül (a fordítási hibákról késbb):

Eredmény

```
/home/work/JavaSuli/Kocsma/src/kocsma/Main.java:9: operator < cannot be applied to boolean,int
    System.out.println(4 < szám < 10);
```

ÉS mvelet

A megoldáshoz a matematikai képletet szét kell választanunk két részre: *4 < szám ÉS szám < 10*, vagyis a szám értéke legyen nagyobb, mint négy és legyen kisebb, mint tíz:

Java

```
int szám = 10;
System.out.println(4 < szám && szám < 10);
```

Mint látható, az *&&* jel helyettesíti az ÉS szót, a futás eredménye pedig *false*, mivel a 10 ugyan nagyobb, mint 4, de nem kisebb, mint 10.

VAGY mvelet

Ha két relációs mvelet közül elég, ha az egyik teljesül, akkor össze tudjuk kapcsolni ket egy *VAGY*mvelettel:

Java

```
int szám = 4;
System.out.println(szám % 2 == 0);
System.out.println(szám % 3 == 0);
System.out.println(szám % 2 == 0 || szám % 3 == 0);
```

Az els esetben a kifejezés akkor lesz igaz, ha a *szám* változó értéke maradék nélkül osztható kettvel. A második esetben hárommal kell oszthatónak lennie, a harmadik esetben a kifejezés akkor lesz igaz, ha a *szám* változó értéke maradék nélkül osztható kettvel *vagy* hárommal. A *VAGY* mveletet a ||jelle l tudjuk jelölni.

NEM mvelet

Sokszor elfordul, hogy a kiszámolt *boolean* eredmény ellentéte kell valamilyen okból kifolyólag, ekkor a *NEM*(más néven tagadás) mveletet kell használnunk:

Java

```
boolean válasz = true;
System.out.println(!válasz);
```

A tagadást a logikai változó elé tett felkiáltó jellel tudjuk megejteni, a példában a tagadásból következik, hogy a *false* kerül kiírásra.

3.3.5. Bitmveletek

Mivel számítógépeink egyelre bitek alapján mködnek, ezért célszer beépíteni egy programnyelvbe a bitmveleteket. A bitmveletek alapja a kettes számrendszer, és minden primitív típus alapveten kettes számrendszerben tárolt szám, ezért - a valós számokat leszámítva - értelmezhet rajtuk az összes bitmvelet. Induljunk ki egy szép kerek számból, azaz nézzük meg, hogy a decimális 10 hogy néz ki kettes számrendszerben 8 biten:

Bitek

```
00001010
```

Keressünk egy másik számot is, például nézzük meg a 57-es számot kettes számrendszerben, szintén 8 biten:

Bitek

```
00111001
```

Ezzel a két számmal fogunk bitmveleteket végezni.

Bitléptetés

A bitléptetés egy nagyon alacsony szint mvelet, a legtöbb CPU támogatja, ritkán szükség is van rá, ezért nem lehetett kihagyni a Java nyelvbl. A bitek léptetése során a bináris számban lév bitek sorrendje azonos marad, azonban a pozíciójuk megváltozik, jobbra vagy balra tolódnak el. Fontos tudni, hogy a bitléptetés mindig 32 bites egész számon történ mvelet, ha más adattípuson végeznénk el, akkor is 32 bites eredményt kapunk!

Bitléptetés balra

A balra léptetés során a bitek balra mozognak el, és jobb oldalon 0 értékek jönnek be, a bal oldalon kies bitek pedig elvesznek:

Java

```
int szám = 10;
System.out.println( Integer.toBinaryString( szám << 0 ));
System.out.println( Integer.toBinaryString( szám << 1 ));
System.out.println( Integer.toBinaryString( szám << 2 ));
System.out.println( Integer.toBinaryString( szám << 3 ));
System.out.println( Integer.toBinaryString( szám << 28 ));
System.out.println( Integer.toBinaryString( szám << 29 ));
System.out.println( Integer.toBinaryString( szám << 30 ));
System.out.println( Integer.toBinaryString( szám << 31 ));
```

Az eredmény magáért beszél, a bitek elkezdenek balra mozogni, jobb oldalon pedig 0 értékek jönnek be:

Eredmény

```
1010
10100
101000
1010000
10100000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Bitléptetés jobbra

A jobbra léptetés azonos módon mködik, mint a balra léptetés:

Java

```
int szám = 10;
System.out.println( Integer.toBinaryString( szám >>> 0 ));
System.out.println( Integer.toBinaryString( szám >>> 1 ));
System.out.println( Integer.toBinaryString( szám >>> 2 ));
System.out.println( Integer.toBinaryString( szám >>> 3 ));
System.out.println( Integer.toBinaryString( szám >>> 28 ));
System.out.println( Integer.toBinaryString( szám >>> 29 ));
System.out.println( Integer.toBinaryString( szám >>> 30 ));
System.out.println( Integer.toBinaryString( szám >>> 31 ));
```

Az eredmény itt is magáért beszél:

Eredmény

```
1010
101
10
1
0
0
0
0
```

Eljeles bitléptetés jobbra

A jobbra léptetés esetén létezik eljeles léptetés, amikor a bináris szám bal szélén nem 0 érték jön be, hanem az eljelbit ismétldik:

Java

```
int szám = -10;
System.out.println( Integer.toBinaryString( szám >> 0 ));
System.out.println( Integer.toBinaryString( szám >> 1 ));
System.out.println( Integer.toBinaryString( szám >> 2 ));
System.out.println( Integer.toBinaryString( szám >> 3 ));
System.out.println( Integer.toBinaryString( szám >> 28 ));
System.out.println( Integer.toBinaryString( szám >> 29 ));
System.out.println( Integer.toBinaryString( szám >> 30 ));
System.out.println( Integer.toBinaryString( szám >> 31 ));
```

Mint látható, bal oldalon alapból egy egyes érték van, és ezzel töltdik fel a bithalmaz:

Eredmény

```
11111111111111111111111111110110
11111111111111111111111111111011
11111111111111111111111111111101
11111111111111111111111111111110
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
```

Bitenkénti tagadás - negáció

Ha a kapott bitek mindegyikét negálni szeretnénk, akkor a ~ operátort kell használnunk:

Java

```
byte szám = 10;
System.out.println(~szám);
```

A kiírt eredmény -11, amely binárisan *11110101*, ugyanis a negatív számokat kettes komplemens alapon kezelik az elterjedt számítógépek. A kettes komplemens egy bitenkénti tagadás, majd az eredményhez hozzáadunk egyet - így lesz a 10 kettes komplemense -10.

Bitenkénti ÉS mvelet

A bitenkénti és mvelethez már kett operandus kell:

Java

```
byte szám = 10;
byte másikSzám = 57;
System.out.println(szám & másikSzám);
```

Az eredmény *8* lesz, amelynek az oka, hogy a két számnak csak a negyedik pozícióban van azonosan 1 értéke:

Eredmény

```
00001010
00111001
00001000
```

Bitenkénti VAGY mvelet

A bitenkénti vagy mvelethez is kett operandus kell:

Java

```
byte szám = 10;
byte másikSzám = 57;
System.out.println(szám | másikSzám);
```

Az eredmény *59*, ami binárisan *00111011*, mivel ott tartalmaz 1 értéket, ahol a megadott két operandusnál legalább egy darab 1 érték volt:

Eredmény

```
00001010
00111001
00111011
```

Bitenkénti KIZÁRÓ VAGY mvelet

A kizáró vagy hasonlít a vagy mvelethez, viszont csak akkor lesz az eredmény is 1 érték, ha a megadott két operandusban azonos pozíción csak egy 1 érték van:

Java

```
byte szám = 10;
byte másikSzám = 57;
System.out.println(szám ^ másikSzám);
```

Amelynek eredménye *51*, hiszen:

Eredmény

```
00001010
00111001
00110011
```

Bitmveletek a gyakorlatban

Rikán találkozunk a programozás során bitmveletekkel, szinte kizárólag csak az ÉS mvelet fordul el, amikor olyan adatokon kell dolgoznunk, amelyek egymás mellé zsúfolt bitekbl állnak.

3.3.6. Egyéb mveletek

A fentieken túl maradt néhány mvelet, amelyeket nem lehet könnyedén csoportba sorolni, ezért kerültek az egyéb mveletek közé.

Mvelettel kombinált értékadás

Gyakori eset, hogy egy változó értéke az elz értékéhez képest változik meg:

Java

```
int szám = 10;
szám = szám + 2;
System.out.println(szám);
```

A *változó* = *változó* ***mveleti jel*** *kifejezés további része*jelleg kifejezéseket rövidíteni tudjuk:

Java

```
int szám = 10;
szám += 2;
System.out.println(szám);
```

A rövidítés során a mveleti jel az egyenlség jel elé kerül és az egyenlség jel után eltnik a kifejezésbl a változó neve: *változó* ***mveleti jel***= *kifejezés további része*. Ezt bármilyen mveletnél meg tudjuk tenni, amely két operandussal mködik.

Feltételes értékadás

Ha egy eldöntend kérdés alapján szeretnénk értéket adni egy változónak, akkor ezt a feltétles értékadással tudjuk megtenni. Nézzünk egy példát az abszolút érték képzésére:

Java

```
int szám = -10;
szám = szám < 0 ? -szám : szám;
System.out.println(szám);
```

A második sorban találjuk a lényeget, amelyet úgy tudunk kiolvasni, hogy a *szám* nev változó értéke **legyen egyenl** a *-szám* értékével, ha a *szám* < *0* **felté tel igaz**, **egyébként** legyen egyenl a *szám*értékével.

Szöveg összefzése

A + jel használható szövegek összefzésére is, ekkor két kisebb szövegbl egy nagyobb szöveg lesz:

Java

```
String egy= "egy";
String kett = "kett";
String egykett = egy + kett;
System.out.println(egykett.length());
```

Az eredmény természetesen *8*.

3.4. Kifejezések

Az operátorok és operandusok halmazát kifejezésnek nevezzük, egy kifejezés mindig egy értéket ad vissza, amelyet általában értékadással átadunk egy változónak, vagy paraméterként adjuk át egy metódusnak. A cikksorozat els részében is használtunk már kifejezéseket, csak nem nevezetük így a leírt sorokat, amelyek *egyszer kifejezések* voltak.

Az *összetett kifejezések*több operátorból és operandusból állnak, és a megszokott matematikai kiértékelési sorrend szerint kell írnunk a kívánalmainkat:

Java

```
int szám = 1 + 2 + 3;
System.out.println(szám);
```

A kifejezés értéke *6* lesz, ahogy azt várjuk, a kifejezése kiértékelése során a számítógép pont úgy viselkedik, ahogy azt annak idején tanultuk: veszi az els két számot, összeadja ket, majd az eredményt és a harmadik számot újfent összeadja. Ebben az esetben nem számít a kiértékelés sorrendje, hiszen teljesen mindegy, hogy a számokat milyen sorrendben adjuk össze, ettl függetlenül a számítógép mindig **balról-jobbra haladva**végzi el az egyes mveleteket. A kiértékelés sorrendje változhat a mveleti jelek függvényében, hiszen a szorzás elnyt élvez az összeadással szemben:

Java

```
int szám = 1 + 2 * 3;
System.out.println(szám);
```

A számítógép elször a szorzás mveletét végzi el, majd a kapott eredményhez hozzáad egyet, így a végs eredmény *7*lesz. Zárójelezéssel ez a kiértékelési sorrend felülbírálható:

Java

```
int szám = (1 + 2) * 3;
System.out.println(szám);
```

Ebben az esetben a számítógép elször a zárójelben lév mveletet végzi el, majd a kapott eredményt megszorozza hárommal, így kapjuk meg az eredményt, ami *9*.

3.4.1. Precedencia - kiértékelési sorrend

A kiértékelési sorrend függ az operátoroktól, így fontos tudni a pontos kiértékelési sorrendet, amelyet vagy megtanulunk, vagy zárójelezünk bizonytalan helyzetben. Lássuk a sorrendet:

1. **unáris** operátorok (a kiértékelés jobbról balra történik!):
    a. **--szám**: a szám értékének elzetes csökkentése
    b. **++szám**: a szám értékének elzetes növelése
    c. **szám--**: a szám értékének utólagos csökkentése
    d. **szám**++: a szám értékének utólagos növelése
    e. **- szám**: a szám eljelének megváltoztatása
    f. **~ szám**: a szám bináris tagadása (negáció)
    g. **! igazHamis**: egy boolean érték tagadása
    h. **(type)**: típuskényszerítés
    i. **new**: új példány létrehozása
2. **multiplikatív** operátorok:
    a. **szám * másikSzámszám / másikSzám**: a szám elosztása egy másikSzámma: a két szám összeszorzása
    b. **szám / másikSzám**: a szám elosztása egy másikSzámmal
    c. **szám % másikSzám**: maradék képzése (szám osztva másikSzámmal)
3. **additív** operátorok:
    a. **szám + másikSzám**: a két szám összeadása
    b. **szám - másikSzám**: számból kivonjuk a másikSzámot
4. **bitléptetés** operátorok:
    a. **szám << másikSzám**: a szám változóban lév bitek léptetése balra másikSzám pozícióval
    b. **szám >> másikSzám**: a szám változóban lév bitek léptetése jobbra másikSzám pozícióval - az eljel figyelembevételével
    c. **szám >>> másikSzám**: a szám változóban lév bitek léptetése jobbra másikSzám pozícióval
5. **relációs** operátorok:
    a. **szám < másikSzám**: a szám kisebb-e, mint a másikSzám
    b. **szám <= másikSzám**: a szám kisebb vagy egyenl-e, mint a másikSzám
    c. **szám > másikSzám**: a szám nagyobb-e, mint a másikSzám
    d. **szám >= másikSzám**: a szám nagyobb vagy egyenl-e, mint a másikSzám
    e. **instanceof**: referencia teszt operátor
6. **egyenlség** operátorok:
    a. **szám == másikSzám**: a két szám egyenl-e egymással
    b. **szám != másikSzám**: a két szám nem egyenl-e egymással
7. **bitenkénti ÉS** operátor:
    a. **szám & másikSzám**: a szám és a másikSzám között bitenkénti ÉS mvelet
    b. **igazHamis & másikIgazHamis**: az igazHamis és a másikIgazHamis közötti ÉS mvelet
8. **bitenkénti KIZÁRÓ VAGY** operátor:
    a. **szám ^ másikSzám**: a szám és a másikSzám között bitenkénti KIZÁRÓ VAGY mvelet
    b. **igazHamis ^ másikIgazHamis**: az igazHamis és a másikIgazHamis közötti KIZÁRÓ VAGY mvelet
9. **bitenkénti VAGY** operátor:

    a. **szám | másikSzám**: a szám és a másikSzám között bitenkénti VAGY mvelet
    b. **igazHamis | másikIgazHamis**: az igazHamis és a másikIgazHamis közötti VAGY mvelet
10. **boolean ÉS** operátor:
    a. **igazHamis && másikIgazHamis**: az igazHamis és a másikIgazHamis közötti ÉS mvelet
11. **boolean VAGY** operátor:
    a. **igazHamis || másikIgazHamis**: az igazHamis és a másikIgazHamis közötti VAGY mvelet
12. **feltételes értékadás** operátor:
    a. **szám == másikSzám ? szám : másikSzám**: feltételes értékadás (lásd fent bvebben)
13. **értékadás** operátor:
    a. **szám = másikSzám**: a szám változó értéke felveszi a másikSzám értékét
14. **értékadás mvelettel**operátorok:
    a. **szám += másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám összege lesz
    b. **szám -= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám különbsége lesz
    c. **szám *= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám szorzata lesz
    d. **szám /= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám hányadosa lesz
    e. **szám %= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám maradéke lesz
    f. **szám &= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám változó közötti bitenkénti ÉS eredménye lesz
    g. **szám ^= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám változó közötti bitenkénti KIZÁRÓ VAGY eredménye lesz
    h. **szám |= másikSzám**: a szám változó értéke a szám elz értéke és a másikSzám változó közötti bitenkénti VAGY eredménye lesz
    i. **szám <<= másikSzám**: a szám változó értéke a szám elz értéke balra léptetve másikSzám pozícióval lesz
    j. **szám >>= másikSzám**: a szám változó értéke a szám elz értéke eljellel jobbra léptetve másikSzám pozícióval lesz
    k. **szám >>>= másikSzám**: a szám változó értéke a szám elz értéke jobbra léptetve másikSzám pozícióval lesz

A prefix (**++szám** vagy **++szám**) és a postfix (**szám++** vagy **szám--**) operátorok használata mindig kockázatos, mivel a kifejezés kiértékelés során változik a változó értéke, vessünk egy pillantást az alábbi programrészletre és próbáljuk meghatározni az eredményt:

Java

```
int szám = 10;
System.out.println(szám++ + ++szám * ++szám + szám++);
```

Nos, az eredmény *179*, mégpedig azért, mert az unáris operátorok az elsk a precedencia táblázatban, és kiértékelésük az elsk között történik, ebbl adódóan a kifejezés a következ lesz az unáris operátorok feloldása után:

Java

```
System.out.println(10 + 12 * 13 + 13);
```

Ha végigkövetjük az értékadásokat, akkor láthatjuk, hogy az eredmény valóban 179.

3.5. Blokkok

A nyelvek fejldése során viszonylag korán megjelentek a blokkok, amelyek utasításokat fognak közre. Az eddigiek során láttunk már több blokkot is, amelyek összefogták az eddig beírt utasításainkat:

Java

```
{
}
```

Ugye ismers? :)

A blokkok hasznos társaink lesznek a programozás során, hiszen utasítások kötegelt végrehajtása gyakori feladat. További hasznuk, hogy a deklarált változóknak hatókört adnak: egy változót mindig csak az t tartalmazó blokkon belül tudjuk elérni:

Java

```
int i = 10;
{
  int j = 10;
}
System.out.println(j);
```

A program nem fordítható le, mivel a *j*változó csak a blokkon belül érhet el. A blokkok lehetvé teszik, hogy egy változónevet többször felhasználhassunk, ha azok olyan blokkban vannak deklarálva, ahol egymást nem látják:

Java

```
int j = 10;
{
  int i = 10;
  System.out.println(i);
  System.out.println(j);
}
{
  int i = 20;
  System.out.println(i);
  System.out.println(j);
}
System.out.println(j);
```

A két *i* nev változó egymásról nem tud, tehát gond nélkül tudjuk használni a program különböz részeiben.

Fontos tudni, hogy változókat bármely blokkon belül tudunk deklarálni, s célszer mindig a blokk elején megejteni azokat a deklarációkat, amelyeknek kezdértéket is adunk, ezzel a programunk átláthatóbb lesz. A programunk írása során tucatnyi változót fogunk létrehozni, amelyeket folyamatosan hozunk létre, ezek is az adott blokkban lesznek csak elérhetk. Ha megnézünk egy Java osztályt, akkor láthatjuk, hogy az is egy nagy blokkban helyezkedik el, ezért rögtön a kezd kapcsos zárójel után létrehozhatunk változókat, amelyeknek a neve **osztályváltozó** vagy **példányváltozó**, attól függen, hogy a módosítója szerint statikus vagy sem. Ezek a változók elfedhetk lokális (metódusokon belüli) változókkal, amelyekre oda kell figyelnünk, mivel aljas hibákat tudnak okozni.