övezet volt. A következő évezredek kultúrembere számára a várható klímaváltozások már nem fognak olyan nagy megpróbáltatást jelenteni mint a kőkorszakbeli elődeinknek. A kor technológiai színvonala majd lehetővé teszi a könnyebb alkalmazkodást.

Puskás Ferenc

A Java nyelv

II. rész - alapok, osztályok

A Java a jövő programozási nyelve, legalábbis erre volt felkészítve. Már a karakterkészlete is más, mint a többi ma létező nyelvé. A Java az *Unicode* karakterkészletet használja, amelyben a karakterek 2 byte-on vannak ábrázolva, így tartalmazza az összes ékezetes karaktert, sőt több nyelv (japán, mongol stb.) ábécéje is jól megfér benne. A Java forráskódokban tetszőleges Unicode karakterek szerepelhetnek. A fordítónak ezt a \u előtaggal és egy hexadecimális számmal adhatjuk meg. Pl. á - \u00e1, é - \u00e9, í - \u00ed stb.

A Java azonosítók betűvel kezdődnek, betűvel vagy számmal folytatódnak. Az azonosítók hossza tetszőleges lehet és a betűket bármelyik Unicode-os ábécéből vehetjük. A betűk közé tartozik az _ és a **$** jel is. A nyelv több mint 50 kulcsszava nem lehet azonosító (**abstract**, **boolean**, **char**, **do**, **if**, **while** stb.). A nyelv három speciális literált is tartalmaz:

null: a null objektum referencia. Bárhol szerepelhet, mert bármilyen objektum referencia típusnak megfelel.

true: a logikai igaz,

false: a logikai hamis értékek jelölésére szolgál.

Habár a Java teljesen objektumorientált nyelv, léteznek benne primitív típusok is, amelyeket objektumok nélkül is használhatunk, az eddig megszokott programozási nyelvekhez hasonlóan. Természetesen ezeknek a típusoknak is megvannak az objektumorientált változataik, amelyek konkrét objektumokhoz kapcsolódnak, ezért létrehozni és inicializálni kell őket a **new** operátor segítségével.

Primitív típusok:

boolean: logikai típus (**true** vagy **false** lehet).

char: 2 byte-os Unicode-os karakter.

byte: 1 byte-os szám

short: 2 byte-os előjeles egész.

int: 4 byte-os előjeles egész.

long: 8 byte-os előjeles egész.

float: 4 byte-os lebegőpontos szám.

double: 8 byte-os lebegőpontos szám.

Ha objektumorientált változatukat (*Boolean, Character, Integer, Long, Float, Double*) használjuk, akkor a *MIN_VALUE* és a *MAX_VALUE* mezők deklarálják az adott típus értéktartományának korlátjait. A *Float* és *Double* osztályok, az IEEE

szabványnak megfelelően deklarálják a *POSITIVE_INFINITY, NEGATIVE_INFINITY* és a *NaN* (Not a Number - nem szám) konstansokat is.

Egy kiemelt szerepet tölt be a String osztály, amely egy karaktersorozatot testesít meg, egy karakterekből álló tömb és egy szám (a karakterek száma) segítségével.

A Java nyelv változóit a C nyelv szabályai szerint deklaráljuk: **int** *x, y; String s;*. Beszélhetünk globális változókról (a program teljes területéről elérhetők) és lokális változókról, amelyek csak egy eljárás vagy blokk belsejében definiáltak.

A tömböket a [] jelöléssel lehet megadni és indexelésül 0-tól kezdődik.

```
int[] i;
```

A tömbök tulajdonképpen speciális objektumok. A fenti példában deklarált változó tetszőleges hosszúságú, egész számokból álló tömbre hivatkoztat. A tömböket a new operátor segítségével lehet létrehozni:

```
i = new int[100];
```

A *length* mező segítségével lekérdezhetjük a tömb méretét (*i.length*). Egy tömb állhat újabb tömbökből is, így jönnek létre a többdimenziós tömbök. Ezen elemek hossza különböző is lehet. Pl. hozzunk létre egy háromszög mátrixot:

```
int [][] m = new int[5][];

for (int i = 0; i < m.length; i++) {
      m[i] = new int[i+1];
}
```

A Java nyelvben három típusú megjegyzést használhatunk: egysoros megjegyzést a // jel vezet be, többsoros, hosszabb megjegyzést a /* - */ jelek közé kell tenni,valamint létezik egy speciális megjegyzés is, amelyet a /** - */ jelek határolnak. Ezek a megjegyzéseket az automatikus dokumentációgeneráló használja fel.

A Java nyelv operátorait és ezek prioritási sorrendjét a következő táblázat foglalja össze:

Típus	**Operátorok**
postfix operátorok	[], (paraméter), kifejezés$_{++}$, kifejezés$_{--}$
prefix operátorok	$_{++}$kifejezés, $_{--}$ kifejezés, + kifejezés, - kifejezés, !
példányosítás típuskényszerítés	new (típus) kifejezés
multiplikatív	*, /, %
additív	+, /
eltolások	>>, <<, >>>
összehasonlítás	<, >, <=, >=, instanceof
egyenlőség	= =, !=
bitenkénti ÉS	&
bitenkénti kizáró VAGY	^
bitenkénti VAGY	|
logikai ÉS	&&
logikai VAGY	||
feltételes kifejezés	?:
értékadások	=, +=, -=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, |=

A Java típusossága is fejlett, a kifejezéseket mindig ellenőrzi és a legkisebb inkompatibilitást is kijelzi. Háromféle típuskonverzióról beszélhetünk:

Automatikus konverzió: néhány konverzió (pl. byte - int, byte - short stb.) magától is megvalósul. Ilyen típusú konverzió valósul meg az objektumoknál is: a leszármazottak mindig kompatibilisek az ősökkel.

Explicit konverzió: a *(típus)kifejezés* kényszerítő operátor segítségével történnek. Vigyázni kell vele, mert gyakran adatvesztéshez vezethetnek (pl. ha **int**-et **byte**-tá konvertálunk).

Szövegkonverzió: minden objektum alapszinten tartalmaz egy *toString* metódust, amely az illető objektumot sztringgé konvertálja. Így ha akifejezésekben *String* típusra lenne szükség, de nem ilyent használunk, a fordító automatikusan megpróbálja meghívni ezt a metódust és sztringgé alakítani az értéket.

Vezérlés

A Java két fontos jellemzője: *strukturált* és *objektumorientált*. Strukturált programozás szempontjából, a nyelv a C-hez hasonlít leginkább. Enyhe bővítésekkel tartalmazza a C nyelv összes vezérelemét.

Blokkokat a {} zárójelpár segítségével hozhatunk létre. A program szövegében az utasítások helyére bárhol kerülhet blokk, ami a maga során nem más, mint utasítások valamilyen sorrendbe vett csoportosítása. Az utasításokat pontosvesszővel zárjuk le. Bármely utasítás elé írható címke (*címke: utasítás*), amely lehetővé teszi az utasítás egyértelmű azonosítását a feltétlen vezérlésátadások esetében.

Az elágazásoknak két formája ismeretes: az *egyszerű* és az *összetett* elágazás. Egyszerű elágazás az

```
if (logikai kifejezés)
     utasítás1
[else
     utasítás2]
```

konstrukcióval valósítható meg. Az összetett elágazás formája a következő:

```
switch (egész kifejezés) {
     case címke1:
          utasítások;
          break;
     case címke2:
     case címke3:
          utasítások;
     ...
     default:
          utasítások;
}
```

A **switch** kulcsszóval bevezetett kifejezés kiértékelése után a **case** ágakban levő címkék lesznek megvizsgálva, ha az érték megegyezik a kifejezés értékével, akkor a vezérlés átadódik a címkét követő utasításnak és a **switch** végéig vagy az első **break** utasításig végrehajtja az összes utasítást. Ha nincs megfelelő címke, akkor a **default** részt hajtja végre, ha ez létezik.

A Java a ciklusok három típusát használja: *elől tesztelő, hátul tesztelő* valamint a **for** ciklust. Az előtesztelős ciklus a ciklusmag lefuttatása előtt leteszteli a cikluskifejezést, ha ez igaz, lefuttatja a ciklusmagot, ha nem, a ciklust követő utasítással folytatja a vezérlést.

```
while (logikai kifejezés)
     utasítás
```

A hátultesztelő ciklus először végrehajtja a ciklusmagot, majd ellenőrzi a cikluskifejezést. Ha ennek kiértékelése az igaz logikai értékhez vezet, újraveszi a ciklusmagot. Megfigyelhető, hogy hamis értékű kifejezés esetén is a ciklusmag egyszer mindenképp végrehajtódik.

```
do
      utasítás
while (logikai kifejezés)
```

A **for** ciklus segítségével nagyon egyszerűen írhatók iteratív, számláló, léptető ciklusok. Érdekessége, hogy a ciklus iteráló változóját lokálisan is lehet deklarálni az utasításban. Formája a következő:

```
for (kezdőrész; logikai kifejezés; továbblépés)
      utasítás
```

A *kezdőrész* deklarálhatja és inicializálja a ciklusváltozókat. A *továbblépési* mód szerint a ciklus addig iterál, ameddig a *logikai kifejezés* értéke igaz.

Egy ciklus magjának a hátralévő részét át lehet ugrani a **continue** utasítás segítségével. A **break** utasítás egy blokkból való feltétel nélküli kilépésre szolgál. Egy metódusból a **return** utasítás segítségével lehet visszatérni.

Osztályok

A nyelv legkisebb önálló egységei az osztályok. Az *egybezártság* tulajdonságát felhasználva az osztály logikailag azonos típusú, összetartozó entitások modellje. Ez a modell egyetlen egészet alkot és a külvilág számára egységesnek mutatkozik. A leírás *adatmező* deklarációkat és *metódus* leírásokat tartalmaz. Működése során a program *példányosítja* az osztályokat, s így *objektumokat* hoz létre. A Java az objektumokat dinamikusan kezeli. Minden objektum egy referencia tulajdonképpen egy memóriazónára, amely az adatokat és a metódusok címeit tartalmazza. A referenciákat létrehozni a **new** operátor segítségével lehet, felszabadítani pedig úgy, hogy egyszerűen **null**-ra állítjuk. A Java értelmező tartalmaz egy belső (*Garbage Collection*-nak nevezett) memóriaellenőrző eljárást, amely az értelmezővel párhuzamosan fut és időről időre felszabadítja azokat a memóriahelyeket, amelyeket semmi sem referál.

Egy új osztályt a **class** kulcsszóval lehet deklarálni, majd tetszőleges sorrendben felsorolhatjuk az adatmezőket és a metódusokat. Az osztályokat *csomagokba* lehet szervezni.

A láthatóság minden egyes elemre külön definiálható. Ha azt akarjuk, hogy az illető elem látható legyen a külvilág számára, akkor ezt a **public** direktívával definiáljuk. A leszármazottak számára láthatóvá tehetjük a **protected** direktívával illetve teljesen elrejthetjük a **private** segítségével. Ha semmilyen direktívával sem illetjük az elemet, akkor az csak az illető csomagon belül lesz látható.

```
class Hónap {
      public String név;
      public int napokSzáma;
      public static int év = 1998;
      public boolean Szökő(){
            return (napokSzáma == 29);
      }
}
```

Ha már deklaráltunk egy osztályt, akkor létrehozhatjuk az objektumokat, példányosíthatjuk az osztályt:

```
Hónap január = new Hónap();
```

Így létrehoztunk egy konkrét hónapot (januárt). A **new** operátor lefoglalta a objektum számára szükséges memóriahelyet, feltölthetjük az adatokat:

```
január.név = "Január";
január.napokSzáma = 31;
```

A **static** módosítóval deklarált *év* mező nem egy-egy objektumhoz tartozik, hanem magához az osztályhoz, tehát a **new** nem foglal számára helyet, lehet rá hivatkozni az *osztálynév.mező* referenciával is (Hónap.év). Az ilyen típusú mezőket az osztály létrejöttekor lehet inicializálni és a memóriában az osztály kódjával egyidőben lesz hely foglalva számukra.

Az adatmezőkhöz hasonlóan a metódusokat is az osztály deklarációjában kell megadni. Egy metódust a *láthatósági_terület módosító visszatérési érték metódusnév paraméterlista metódustörzs* konstrukcióval lehet megadni. Itt is használhatjuk a **static** módosítót. Hatására a metódus osztály metódussá válik. A visszatérési érték bármilyen típus lehet, vagy **void**, ha a metódus nem térit vissza semmilyen értéket. A paraméterlista lehet üres is, ebben az esetben is ki kell tenni azonban a () zárójeleket. A metódustörzs nem választható külön a metódus definíciójától.

A **static** módosítón kívül használhatók meg az **abstract** (absztrakt metódus - a törzset valamelyik leszármazott definiálja), **final** (végleges - nem lehet megváltoztatni, felülírni), **synchronized** (párhuzamos szálak számára) és **native** (nem Java-ban implementált metódus) módosítók is.

Ha egy metódustörzsben hivatkozni akarunk az aktuális példányra, akkor ezt a **this** paraméterrel tehetjük meg. A **this** tulajdonképpen az objektumnak egy pszeudó-adatmezője, amelyik mindig az aktuális objektum címét tartalmazza.

Egy osztályban több metódust is el lehet nevezni ugyanazzal a névvel, amennyiben a paraméterlistája különböző. Java-ban a metódusnév mellett a paraméterlista is fontos szerepet játszik egy metódus azonosításakor. A metódusnév többszörös használatát *túlterhelésnek* nevezzük.

Konstruktorok, destruktorok

A fenti példán is megfigyelhettük, hogy amikor példányosítottunk egy osztályt, az *Osztály Változó* = **new** *Osztály();* konstrukciót használtuk. Joggal vetődik fel a kérdés, hogy a **new** operátor után miért írtuk még egyszer az osztály nevét () zárójellel - mintha valamilyen metódus lenne. A válasz: tényleg metódusról van szó, mégpedig egy sajátos metódusról, a *konstruktor*ról. A konstruktor olyan programkód, amely automatikusan végrehajtódik egy objektum létrehozásakor. A konstruktorokat bizonyos inicializálásokra használhatjuk fel. Nevüknek meg kell egyeznie az osztály nevével, de lehet paraméterlistájuk. A túlterhelés miatt egy osztálynak több konstruktora is lehet, a paraméterlistától függ, hogy melyik hívódik meg. A konstruktor definíciója majdnem olyan mint egy metódus definíció, azzal a különbséggel, hogy a konstruktornak nincs visszatérési értéke, tehát **void** metódusként viselkedik. Pl. lássuk el a hónap osztályunkat egy konstruktorral:

```
class Hónap {
        ...
        public Hónap(String név, int napokSzáma) {
                this.név = név;
                this.napokSzáma = napokSzáma;
        }
        ...
}
```

Egy osztálynak mindig van konstruktora. Ha a programozó nem ír konstruktort, akkor a fordítóprogram biztosít egy úgynevezett *implicit konstruktort*, amelynek törzse üres, nincsenek paraméterei és publikus. Az objektumokat most már úgy hozhatjuk létre, hogy a mi konstruktorunkat hívjuk:

```
Hónap január = new Hónap("Január", 31);
```

Az osztályváltozók inicializálása nem történhet a fent említett módszerrel, hiszen **static** konstruktorok nincsenek. A megoldás az *inicializáló blokk*. Ez egy olyan utasításblokk, amely a változódeklarációk és metódusdeklarációk között helyezkedik el és mindig lefut az osztály inicializálásakor:

```
class Hónap {
        ...
        public static int év;
        static {
                év = 1998;
        }
        ...
}
```

A **static** kulcsszó *osztályinicializátort* vezetett be. Beszélhetünk *példányinicializátorról* is, amennyiben nem használunk **static** módosítót. A példányinicializáló mindig végrehajtódik az objektumok létrehozásakor és azt a kódot tartalmazhatja, amelyik minden konstruktor hívásakor végre kell hajtódjon. Egy osztálynak akárhány inicializáló blokkja lehet, és ezek az előfordulás sorrendjében hajtódnak végre.

Az objektumok megszüntetéséről a *szemétgyűjtő* algoritmus (Garbage Collector) gondoskodik. Felvetődhetnek azonban olyan feladatok, amelyek megoldásához elengedhetetlen, hogy értesüljünk az objektum megszüntetéséről. Erre ad választ az a mechanizmus, amely biztosítja, hogy a megszüntetés előtt meghívódjon az osztály **finalize** nevű (destruktor jellegű) metódusa. Fontos, hogy ez a metódus paraméter nélküli, **void** és **protected** legyen. Ennek a metódusnak az osztályszintű megfelelője a **classFinalize** osztálymetódus (**static**, **void** és paraméter nélküli), amely mindig meghívódik az osztály felszabadításakor. Egy osztály akkor szabadul fel, ha már nem rendelkezik példányokkal és már nem hivatkoznak rá.

A program

Mint már említettük a Java teljesen objektumorientált nyelv. A Java forrásszöveg *.java* kiterjesztésű állományba kerül, ezt fordítja le köztes (*byte*) kóddá a *javac* fordító. A köztes kód *.class* kiterjesztésű állományokban található, és mindegyik állomány egy osztályt tartalmaz. A *.class* kiterjesztésű állományt pedig a *java* értelmező (Java Virtual Machine) futtatja. Joggal tevődik fel az a kérdés, hogy honnan tudja az értelmező melyik az első objektum, melyik metódust kell először meghívni. Más szóval milyen objektumot hozzon először létre, mert objektumot csak egy metódusbeli kódrész hozhat létre, de metódus nem létezhet az objektum létrejötte előtt. A megoldás a következő: a *java* értelmező a neki megadott osztályt futtatja, éspedig úgy, hogy megkeresi az osztály speciális, **main** nevű metódusát. A **main** metódus **static**, vagyis osztálymetódus, objektumok nélkül is hívható, **void** és **public** elérhetőségű.

```
class Helló {
        public static void main(String[] args) {
                System.out.println("Helló!");
        }
}
```

A **main** metódusnak mindig van egy paramétere az args, amely a parancssorban megadott argumentumokat tartalmazza szöveges formában. Az argumentumok számát az *args* tömb *length* metódusa segítségével lehet lekérdezni. Megfigyelhető az is, hogy szöveget megjeleníteni a *System* osztály *out* objektumának *println* metódusával lehet. Mindezekről azonban következő lapszámainkban...

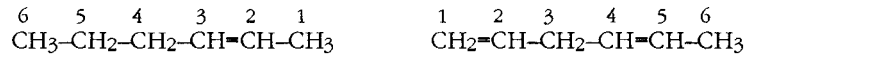
Kovács Lehel

Szerves vegyületek nevezéktana

III. Nyíltláncú telítetlen szénhidrogének és gyökeik megnevezése

A kettes kötést tartalmazó ciklikus, nem elágazó telítetlen szénhidrogének nevét a megfelelő alkánok nevéből képezzük az -án végződést -én-re cserélve. Ha több kettes kötést tartalmaz a molekula, akkor a végződés -adién, -atrién stb.

E vegyületek nevében a kettes kötés(eke)t hordozó szénatom(ok) sorszáma a lehető legkisebb kell legyen. Pl.



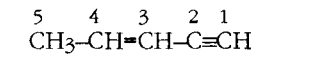
2–hexén 1,4–hexadién

A hármas kötést tartalmazó, aciklikus, nem elágazó telítetlen szénhidrogének nevét a megfelelő alkánok nevéből képezzük az -án végződést -in-re cserélve. Ha több hármas kötés van jelen, a végződés -adiin, atriin stb.

Ezen vegyületek nevében a hármas kötés(eke)t hordozó szénatom(ok) sorszáma a lehető legkisebb kell legyen.

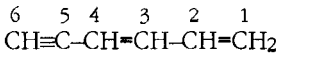
Triviális neve van az etinnek: CH ≡ CH acetilén.

A kettes és hármas kötést egyaránt tartalmazó, aciklikus, nem elágazó telítetlen szénhidrogének estén a szénlánc végéhez legközelebb eső, többszörös kötést hordozó szénatom sorszáma a legkisebb. A vegyületet mindig, mint alkén, alkadién stb. nevezzük el, a hármas kötéseket pedig a megfelelő sorszámokkal ellátva az alapelnevezés után soroljuk föl. Pl.



3–pentén–1–in

Ha kétféleképpen is lehet számozni, a kettes kötést hordozó szénatomok kapják az alacsonyabb sorszámot. Pl.



1,3–hexadién–5–in

Az elágazást is tartalmazó telítetlen vegyületeknél a fő láncot úgy választjuk meg, hogy az

– a legtöbb többszörös kötést tartalmazza;

– a leghosszabb legyen;