Java Programozás (folytatás)

6. Vezérlési szerkezetek

6.1. A *while* és a *do-while* ciklusok

A *while* ciklus utasításblokk végrehajtására használható, amíg a feltétel igaz. A *while* ciklus szintaxisa:

```
while (feltétel) {
    utasítások
}
```

A *while* ciklus először kiértékeli a feltételt, amely művelet egy *boolean* értéket ad vissza. Ha a kifejezés értéke igaz, a *while* ciklus végrehajtja *while* blokkjában szereplő utasításokat. A *while* ciklus addig értékeli ki a kifejezést és hajtja végre az utasításblokkot, amíg a kifejezés hamis értékű nem lesz.

A Java nyelv egy a *while* ciklushoz hasonló utasítást is biztosít — *a do-while* ciklust. A *do-while* szintaxisa:

```
do {
    utasítás(ok)
} while (feltétel);
```

Ahelyett, hogy a feltételt a ciklus végrehajtása előtt értékelné ki, a *do-while* ezt a ciklusmag lefutása után teszi meg. Így a *do-while* magjában szereplő utasítások minimum egyszer végrehajtódnak.

6.2. A *for* ciklus

A *for* utasítás jó módszer egy értéktartomány bejárására. A *for* utasításnak van egy hagyományos formája, és a Java 5.0-tól kezdődően egy továbbfejlesztett formája is, amit tömbökön és gyűjteményeken való egyszerű bejárásnál használhatunk. A *for* utasítás általános formája a következőképpen néz ki:

```
for (inicializálás; feltétel; növekmény) {
    utastás(ok)
}
```

Az inicializálás egy olyan kifejezés, amely kezdőértéket ad a ciklusnak – ez egyszer, a ciklus elején fut le. A feltétel kifejezés azt határozza meg, hogy meddig kell a ciklust ismételni. Amikor a kifejezés hamisként értékelődik ki, a ciklus nem folytatódik. Végezetül a növekmény egy olyan kifejezés, amely minden ismétlődés után végrehajtódik a ciklusban. Mindezen összetevők opcionálisak. Tulajdonképpen ahhoz, hogy egy végtelen ciklust írjunk, elhagyjuk mindhárom kifejezést:

```
for ( ; ; ) {
    ...
}
```

A *for* ciklusokat gyakran arra használjuk, hogy egy tömb elemein vagy egy karakterláncon végezzünk iterációt. Az alábbi példa, *ForDemo*, egy *for* utasítást használ arra, hogy végighaladjon egy tömb elemein és kiírja őket.

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12,
                              1076, 2000, 8, 622, 127 };

        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

A program futási eredménye:

```
32 87 3 589 12 1076 2000 8 622 127
```

6.3. Az *if-else* szerkezet

Az *if* utasítás lehetővé teszi a programunk számára, hogy valamilyen kritérium szerint kiválasztva futtasson más utasításokat. Például tegyük fel azt, hogy a programunk hibakereső (*debugging*) információkat ír ki egy *DEBUG* nevű, *boolean* típusú változó értéke alapján. Ha a *DEBUG* igaz, a program kiírja az információt, az *x* változó értékét. Különben a program futása normálisan folytatódik. Egy ilyen feladatot implementáló programrész a következőképpen nézhet ki:

```
if (DEBUG) {
    System.out.println("DEBUG: x = " + x);
}
```

Ez az *if* utasítás legegyszerűbb formája. Az *if* által vezérelt blokk végrehajtódik, ha a feltétel igaz. Általában az *if* egyszerű alakja így néz ki:

```
if (feltétel) {
    kifejezések
}
```

Mi van akkor, ha az utasítások más változatát akarjuk futtatni, ha a feltétel kifejezés hamis? Erre az *else* utasítást használhatjuk. Vegyünk egy másik példát. Tegyük fel azt, hogy a programunknak különböző műveleteket kell végrehajtania attól függően, hogy a felhasználó az *OK* gombot vagy más gombot nyom meg a figyelmeztető ablakban. A programunk képes lehet erre, ha egy *if* utasítást egy *else* utasítással együtt használunk.

```
if (response == OK) {
    //code to perform OK action
} else {
    //code to perform Cancel action
}
```

Az *else* blokk akkor kerül végrehajtásra, ha az *if* feltétele hamis. Az *else* utasítás egy másik formája az *else if* egy másik feltételen alapulva futtat egy utasítást. Egy *if* utasításnak lehet akárhány *else if* ága, de *else* csak egy. Az alábbi *IfElseDemo* program egy teszt pontszámot alapul véve egy osztályzatot határoz meg: 5-ös 90%-ért vagy afölött, 4-es 80%-ért vagy afölött és így tovább:

```
public class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;
        int grade;

        if (testscore >= 90) {
            grade = 5;
        } else if (testscore >= 80) {
            grade = 4;
        } else if (testscore >= 70) {
            grade = 3;
        } else if (testscore >= 60) {
            grade = 2;
        } else {
            grade = 1;
        }

        System.out.println("Grade = " + grade);
    }
}
```

Ennek a programnak a kimenete:

```
Grade = 3
```

Megfigyelhetjük, hogy a *testscore* értéke több kifejezés feltételének is eleget tehet az alábbi *if* utasítások közül: 76 >= 70 és 76 >= 60. Azonban, ahogy a végrehajtó rendszer feldolgoz egy olyan összetett *if* utasítást, mint ez, amint egy feltétel kielégül, lefutnak a megfelelő utasítások (*grade = 3*), és a vezérlés kikerül az *if* utasításból anélkül, hogy a további feltételeket kiértékelné.

6.4. A *switch-case* szerkezet

Akkor használhatjuk a *switch* utasítást, ha egy egész szám értéke alapján akarunk végrehajtani utasításokat. A következő *SwitchDemo* példaprogram egy *month* nevű egész típusú változót deklarál, melynek értéke vélhetőleg a hónapot reprezentálja egy dátumban. A program a *switch* utasítás használatával a hónap nevét jeleníti meg a *month* értéke alapján.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;

        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Not a month!");
                     break;
        }
    }
}
```

Végül a *switch* utasításban használhatjuk a *default* utasítást, hogy mindazokat az értékeket is kezelhessük, amelyek nem voltak egy *case* utasításban sem kezelve.

A *switch* utasítás kiértékeli kifejezést, ez esetben a *month* értékét, és lefuttatja a megfelelő *case* utasítást. Ezáltal a program futási eredménye az *August* lesz. Természetesen ezt *az if* utasítás felhasználásával is megoldhatjuk:

```
int month = 8;

if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}

...
```

Annak eldöntése, hogy az *if* vagy a *switch* utasítást használjuk, programozói stílus kérdése. Megbízhatósági és más tényezők figyelembevételével eldönthetjük, melyiket használjuk. Míg egy *if* utasítást használhatunk arra, hogy egy értékkészlet vagy egy feltétel alapján hozzunk döntéseket, addig a *switch* utasítás egy egész szám értéke alapján hoz döntést. Másrészt minden *case* értéknek egyedinek kell lennie, és a vizsgált értékek csak konstansok lehetnek.

Egy másik érdekesség a *switch* utasításban a minden *case* utáni *break* utasítás. Minden egyes *break* utasítás megszakítja az épp bezáródó *switch* utasítást, és a vezérlés szála a *switch* blokk utáni első utasításhoz kerül. A *break* utasítások szükségesek, mivel nélkülük a *case* utasítások értelmüket vesztenék. Vagyis egy explicit *break* nélkül a vezérlés folytatólagosan a rákövetkező *case* utasításra kerül (átcsorog). Az alábbi *SwitchDemo2* példa azt illusztrálja, hogyan lehet hasznos, ha a *case* utasítások egymás után lefutnak.

```
public class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                numDays = 0;
                   break;
        }

        System.out.println("Number of Days = " + numDays);
    }
}
```

A program kimenete:

```
Number of Days = 29
```

6.4.1 A *switch* utasítás és a felsorolt típus

A felsorolt adattípus az 5.0-ban bevezetett újdonság, amiről később olvashat majd. Ez a rész csak azt mutatja be, hogyan használhatjuk őket egy *switch* utasításban. Szerencsére ez pont olyan, mint a *switch* használata az egész típusú változók esetén.

Az alábbi *SwitchEnumDemo* kódja majdnem megegyezik azzal a kóddal, amit korábban a *SwitchDemo2*-ben láttunk. Ez az egész típusokat felsorolt típusokkal helyettesíti, de egyébként a *switch* utasítás ugyanaz.

```
public class SwitchEnumDemo {
    public enum Month { JANUARY, FEBRUARY, MARCH, APRIL,
                        MAY, JUNE, JULY, AUGUST, SEPTEMBER,
                        OCTOBER, NOVEMBER, DECEMBER }

    public static void main(String[] args) {
        Month month = Month.FEBRUARY;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case JANUARY:
            case MARCH:
            case MAY:
            case JULY:
            case AUGUST:
            case OCTOBER:
            case DECEMBER:
                numDays = 31;
                break;
            case APRIL:
            case JUNE:
            case SEPTEMBER:
            case NOVEMBER:
                numDays = 30;
                break;
```

```
                case FEBRUARY:
                    if ( ((year % 4 == 0) && !(year % 100 == 0))
                         || (year % 400 == 0) )
                        numDays = 29;
                    else
                        numDays = 28;
                    break;
                default:
                    numDays=0;
                    break;
            }

            System.out.println("Number of Days = " + numDays);
        }
}
```

Ez a példa csak egy kis részét mutatta be annak, amire a Java nyelvi felsorolások képesek. A továbbiakat később olvashatja el.

6.5. Vezérlésátadó utasítások

Kivételkezelő utasítások

A Java programozási nyelv egy kivételkezelésnek nevezett szolgáltatást nyújt, hogy segítse a programoknak a hibák felderítését és kezelését. Amikor egy hiba történik, a program „dob egy kivételt”. Ez azt jelenti, hogy a program normális végrehajtása megszakad, és megkísérel találni egy kivételkezelőt, vagyis egy olyan kódblokkot, ami a különféle típusú hibákat le tudja kezelni. A kivételkezelő blokk megkísérelheti a hiba kijavítását, vagy ha úgy tűnik, hogy a hiba visszaállíthatatlan, akkor szabályosan kilép a programból.

Alapvetően három utasítás játszik szerepet a kivételkezelésekben:

- a *try* utasítás tartalmaz egy utasítás blokkot, amiben a kivétel dobása elképzelhető
- a *catch* utasítás tartalmaz egy olyan utasításblokkot, ami le tudja kezelni az azonos típusú kivételeket. Az utasítások akkor hajtódnak végre, ha *kivételtípus* típusú kivétel váltódik ki a *try* blokkban
- a *finally* egy olyan utasítás blokkot tartalmaz, ami végrehajtódik akkor is, ha a *try* blokkban hiba történt, és akkor is, ha hiba nélkül futott le a kód.

Az utasítások általános alakja:

```
try {
    utasítás(ok)
} catch (kivételtípus kivételobjektum) {
    utasítás(ok)
} finally {
    utasítás(ok)
}
```

A kivételkezelés módszerének részletes ismertetésére később kerül sor.

Függvény visszatérése

A *return* (visszatérés) utasítás

Ez az utasítás az utolsó a feltétlen vezérlésátadó utasítások közül. A *return*-t az aktuális metódusból vagy konstruktorból való kilépésre használjuk. A vezérlés visszaadódik annak az utasításnak, ami az eredeti hívást követi. A *return* utasításnak két formája van: ami visszaad értéket, és ami nem. Hogy visszatérjen egy érték, egyszerűen tegyük az értéket (vagy egy kifejezést, ami kiszámítja azt) a *return* kulcsszó után:

```
return ++count;
```

A visszaadott érték adattípusa meg kell, hogy egyezzen a függvényben deklarált visszatérési érték típusával. Ha a függvényt *void*-nak deklaráltuk, használjuk a *return* azon alakját, ami nem ad vissza értéket:

```
return;
```

7. Objektumok használata

Egy tipikus Java program sok objektumot hoz létre, amik üzenetek küldésével hatnak egymásra. Ezeken keresztül tud egy program különböző feladatokat végrehajtani. Amikor egy objektum befejezi a működését, az erőforrásai felszabadulnak, hogy más objektumok használhassák azokat.

A következő *CreateObjectDemo* program három objektumot hoz létre: egy *Point* és két *Rectangle* objektumot:

```
public class CreateObjectDemo {
    public static void main(String[] args) {
        Point originOne = new Point(23, 94);

        Rectangle rectOne =
            new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        System.out.println("Width of rectOne: " +
            rectOne.width);
        System.out.println("Height of rectOne: " +
            rectOne.height);
        System.out.println("Area of rectOne: "
            + rectOne.area());

        rectTwo.origin = originOne;
        System.out.println("X Position of rectTwo: "
            + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
        + rectTwo.origin.y);
```

```
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: "
            + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: "
            + rectTwo.origin.y);
    }
}
```

Ez a program létrehoz, megváltoztat és információt ír ki különböző objektumokról. Kimenete:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000

X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

7.1. Objektumok létrehozása

Az objektum alapját egy osztály szolgáltatja, osztályból hozunk létre (példányosítunk) objektumot. A következő sorok objektumokat hoznak létre, és változókhoz rendelik őket:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

Az első sor egy *Point* osztályból, a második és harmadik a *Rectangle* osztályból hoz létre objektumot.

Minden sor a következőket tartalmazza:

- **Deklaráció**: Az = előtti részek a deklarációk, amik a változókhoz rendelik hozzá az objektum típusokat.
- **Példányosítás**: A *new* szó egy Java operátor, ami létrehoz egy objektumot.
- **Inicializáció**: A *new* operátort egy konsturktorhívás követi. Pl. a *Point (23,94)* meghívja a *Point* egyetlen konstruktorát. A konstruktor inicializálja az új objektumot.

Objektum példányosítása

A *new* operátor egy példányt hoz létre egy osztályból, és memóriaterületet foglal az új objektumnak.

A *new* operátor után szükség van egy osztályra, ami egyben egy konstruktor hívást is előír. A konstruktor neve adja meg, hogy melyik osztályból kell példányt létrehozni. A konstruktor inicializálja az új objektumot.
A *new* operátor egy hivatkozást ad vissza a létrehozott objektumra. Gyakran ezt a hivatkozást hozzárendeljük egy változóhoz. Ha a hivatkozás nincs hozzárendelve változóhoz, az objektumot nem lehet majd elérni, miután a *new* operátort tartalmazó utasítás végrehajtódott. Az ilyen objektumot **névtelen objektumnak** is szoktuk nevezni.

Megjegyzés: A névtelen objektumok nem olyan ritkák, mint ahogy azt gondolhatnánk. Pl. egy tömbbe vagy tárolóba helyezett objektum is névtelen, hiszen nincs saját, névvel ellátott hivatkozása.

Objektum inicializálása

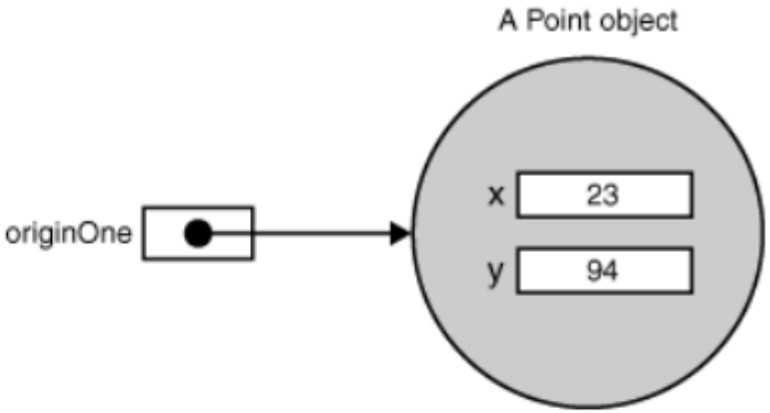
A *Point* osztály kódja:

```
public class Point {
    public int x = 0;
    public int y = 0;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Ez az osztály egy konstruktort tartalmaz. A konstruktornak ugyanaz a neve, mint az osztálynak, és nincs visszatérési értéke. A *Point* osztály konstruktora két egész típusú paramétert kap: (*int x, int y*). A következő utasítás a 23 és 94 értékeket adja át paraméterként:

```
Point originOne = new Point(23, 94);
```

Ennek a hatását mutatja a következő ábra:



A *Rectangle* osztály négy konstruktort tartalmaz:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public Rectangle() {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p) {
        origin = p;
    }

    public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
    }
```

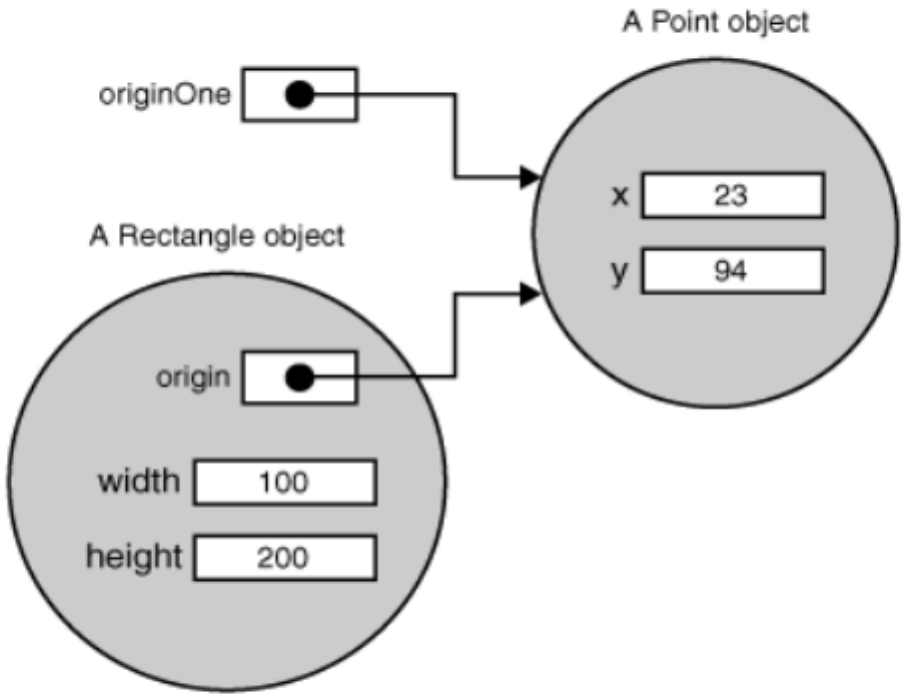
```
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    public int area() {
        return width * height;
    }
}
```

Akármelyik konstruktorral kezdeti értéket adhatunk a téglalapnak, különböző szempontok szerint: a koordinátái (*origin*); szélessége és magassága; mind a három; vagy egyik sem.

Ha egy osztálynak több konstruktora van, mindnek ugyanaz a neve, de különböző számú vagy különböző típusú paraméterekkel rendelkezik. A Java platform a konstruktort a paraméterek száma és típusa alapján különbözteti meg. A következő kódnál a *Rectangle* osztálynak azt a konstruktorát kell meghívnia, ami paraméterként egy *Point* objektumot és két egészet vár:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

Ez a hívás inicializálja a téglalap *origin* változóját az *originOne*-nal (*originOne* értékét veszi fel az *origin* változó), ami egy *Point* objektumra hivatkozik, a *width* értéke 100-zal lesz egyenlő, a *height* értékét 200-zal. Most már két hivatkozás van ugyanarra a *Point* objektumra; egy objektumra több hivatkozás is lehet:



A következő sor két egész típusú paraméterrel rendelkező konstruktort hívja meg, amik a *width* és *height* értékei. Ha megnézzük a konstruktor kódját, látjuk, hogy létrehoz egy új *Point* objektumot, aminek az *x* és *y* értéke is nulla lesz:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

Ez a konstruktor nem vár paramétereket, paraméter nélküli konstruktor:

```
Rectangle rect = new Rectangle();
```

A tagváltozók hozzáférhetősége

Konvenció szerint egy objektum tagváltozóit más objektum vagy osztály közvetlenül nem módosíthatja, mert lehet, hogy értelmetlen érték kerülne bele.

Ehelyett, hogy a változtatást megengedje, egy osztály biztosíthat metódusokat, amiken keresztül más objektumok megnézhetik, illetve módosíthatják a változóit. Ezek a metódusok biztosítják, hogy a megfelelő típus kerüljön a változóba. A másik előnyük, hogy az osztály megváltoztathatja a változó nevét és típusát anélkül, hogy hatással lenne a klienseire.

Azonban néha szükség lehet arra, hogy közvetlen hozzáférést biztosítsunk a változókhoz. Ezt úgy tehetjük meg, hogy a *public* szót írjuk eléjük, a *private*-tel pedig tilthatjuk a külső hozzáférést.

Metódusok hozzáférhetősége

Ugyanúgy működik, mint a változókhoz való hozzáférés. A metódusokhoz való hozzáférést is a *public* kulcsszóval engedélyezhetjük más objektumoknak, a *private*-tel pedig tilthatjuk.

A Stringek hossza

Azon metódusokat, amelyeket arra használunk, hogy információt szerezzünk egy objektumról, olvasó (vagy hozzáférő) metódusoknak nevezzük. Egy ilyen *String*-eknél használható metódus a *length*, amely visszaadja az objektumban tárolt karakterek számát. Miután az alábbi két sor végrehajtódik, a *len* változó értéke 17 lesz:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

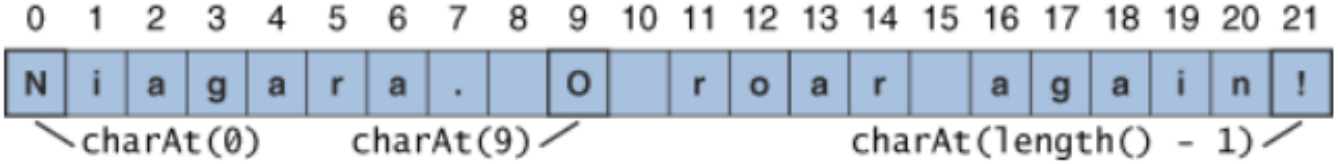
Stringek karaktereinek olvasása

A megfelelő indexű karaktert megkapjuk a *String*-en belül, ha meghívjuk a *charAt* függvényt. Az első karakter indexe 0, az utolsó karakteré pedig a *length()-1*.

Például az alábbi forráskódban a 9. indexű karaktert kapjuk meg a *String*-ben:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Az indexelés 0-val kezdődik, tehát a 9-es indexű karakter az 'O', mint ahogy a következő ábra is mutatja:



Használjuk a *charAt* függvényt, hogy megkapjuk a megfelelő indexű karaktert.

Az ábra is mutatja, hogy hogyan lehet kiszámítani egy *String*-ben az utolsó karakter indexét. Ki kell vonni a *length*() függvény visszatérési értékéből 1-et.

Karakter vagy String keresése Stringben

A *String* osztály két függvényt nyújt, amelyek pozícióval térnek vissza: *indexOf* és a *lastIndexOf*. A következő táblázat e két függvény alakjait mutatja be:

int indexOf(int) *int lastIndexOf(int)*	Visszaadja az első (utolsó) előforduló karakter indexét.
int indexOf(int, int) *int lastIndexOf(int, int)*	Visszaadja az első (utolsó) előfordulókarakter indexét, az indextől előre (visszafele) keresve.
int indexOf(String) *int lastIndexOf(String)*	Visszaadja az első (utolsó) előforduló *String* indexét.
int indexOf(String, int) *int lastIndexOf(String, int)*	Visszaadja a String első (utolsó) előfordulásának indexét, a megadott indextől előre (visszafele) keresve.

Sztringek és rész-sztringek összehasonlítása

A *String* osztálynak van néhány függvénye a sztringek és a rész-sztringek összehasonlítására. Az alábbi táblázat ezeket a függvényeket mutatja be:

boolean endsWith(String) *boolean startsWith(String)* *boolean startsWith(String, int)*	Visszatérési értéke igaz, ha a *String* a paraméterben megadott szóval kezdődik, vagy végződik. Az *int* paraméterben az eltolási értéket adhatjuk meg, hogy az eredeti *String*-ben hanyadik indextől kezdődjön a keresés.
int compareTo(String) *int compareTo(Object)* *int compareToIgnoreCase(String)*	Két *String*-et hasonlít össze ABC szerint, és egy egész számmal tér vissza, jelezve, hogy ez a *String* nagyobb (eredmény>0), egyenlő (eredmény=0), illetve kisebb (eredmény<0), mint a paraméter. A *CompareToIgnoreCase* nem tesz különbséget a kis-és nagybetűk között.
boolean equals(Object) *boolean equalsIgnoreCase(String)*	Visszatérési értéke igaz, ha a *String* ugyanazt a karaktersorozatot tartalmazza, mint a paramétere. Az *equalsIgnoreCase* függvény nem tesz különbséget kis- és nagybetűk között; így *'a'* és *'A'* egyenlő.

Sztringek módosítása

A *String* osztály sokféle metódust tartalmaz a *String*-ek módosításához. Természetesen a *String* objektumokat nem tudja módosítani, ezek a metódusok egy másik *String*-et hoznak létre, ez tartalmazza a változtatásokat. Ezt követhetjük az alábbi táblázatban.

String concat(String)	A *String* végéhez láncolja a *String* paramétert. Ha az paraméter hossza 0, akkor az eredeti *String* objektumot adja vissza.
String replace(char, char)	Felcseréli az összes első paraméterként megadott karaktert a második paraméterben megadottra. Ha nincs szükség cserére, akkor az eredeti *String* objektumot adja vissza.
String trim()	Eltávolítja az elválasztó karaktereket a *String* elejéről és a végéről.
String toLowerCase() *String toUpperCase()*	Konvertálja a *String*-et kis, vagy nagybetűsre. Ha nincs szükség konverzióra, az eredeti *String*-et adja vissza.

8.3. Sztringek darabolása

A *java.util.StringTokenizer* osztály hasznos lehet, ha egy *String*-et adott elválasztó karakter(ek) mentén szét kell bontani. A következő egyszerű példa bemutatja a használat módját:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

A kód a következő eredményt írja ki:

```
this
is
a
test
```

A *StringTokenizer* objektum nyilván tartja, hogy a feldolgozás a *String* melyik pontján jár. A konstruktornak megadhatunk a szövegen kívül egy elválasztó-karaktereket tartalmazó *String*-et is, ekkor az alapértelmezett *" \t\n\r\f"* elválasztók helyett ezt fogja az objektum figyelembe venni.

9.5. Aritmetika

A Java programozási nyelv támogatja az alapvető aritmetikai számításokat az aritmetikai operátorokkal együtt: +, -, *, /, és %. A *java.lang* csomagban a Java platform biztosítja a *Math* osztályt. Ez olyan metódusokat és változókat biztosít, amik segítségével már magasabb rendű matematikai számítások is elvégezhetők, mit például egy szög szinuszának kiszámítása, vagy egy szám bizonyos hatványra emelése.

A *Math* osztály metódusai osztálymetódusok, tehát közvetlenül az osztály nevével kell őket meghívni, például így: (Statikus metódusok):

```
Math.round(34.87);
```

A *Math* osztály metódusai közül elsőként különböző alapvető matematikai függvényeket nézzük meg, mint például egy szám abszolút értékének kiszámítása vagy egy szám kerekítése valamelyik irányban. A következő táblázat ezeket a metódusokat tartalmazza:

double abs(double) *float abs(float)* *int abs(int)* *long abs(long)*	A paraméterként kapott paraméter abszolút értékével tér vissza.
double ceil(double)	A legkisebb *double* értékkel tér vissza, ami nagyobb vagy egyenlő a paraméterrel, és egyenlő egy egész számmal. (felfelé kerekít)
double floor(double)	A legnagyobb *double* értékkel tér vissza, ami kisebb vagy egyenlő a paraméterrel, és azonos egy egész számmal. (lefelé kerekít)

double rint(double)	A paraméterhez legközelebb álló *double* értékkel tér vissza, és azonos egy egész számmal. (a legközelebbi egészhez kerekít)
long round(double) *int round(float)*	A legközelebbi *long* vagy *int* értéket adja vissza, ahogy azt a metódus visszatérési értéke jelzi.

További két alapvető metódus található a *Math* osztályban. Ezek a *min* és a *max*. Az alábbi táblázat mutatja be a *min* és *max* metódusok különböző formáit, amik két számot hasonlítanak össze, és a megfelelő típusú értékkel térnek vissza.

double min(double, double) *float min(float, float)* *int min(int, int)* *long min(long, long)*	A két paraméterből a kisebbel térnek vissza.
double max(double, double) *float max(float, float)* *int max(int, int)* *long max(long, long)*	A két paraméterből a nagyobbal térnek vissza.

A *MinDemo* program mutatja be a *min* alkalmazását két érték közül a kisebb kiszámítására:

```
public class MinDemo {
    public static void main(String[] args) {

        double enrollmentPrice = 45.875;
        double closingPrice = 54.375;

        System.out.println("A vételár: $"
            + Math.min(enrollmentPrice, closingPrice));
    }
}
```

A program valóban az alacsonyabb árat adta vissza:

```
A vételár: $45.875
```

A *Math* osztály következő metódusai a hatványozással kapcsolatosak. Ezen kívül megkaphatjuk az **e** értékét a *Math.E* használatával.

double exp(double)	A szám exponenciális értékével tér vissza.
double log(double)	A szám természetes alapú logaritmusával tér vissza.
double pow(double, double)	Az első paramétert a második paraméternek megfelelő hatványra emeli.
double sqrt(double)	A paraméter négyzetgyökével tér vissza.

A következő *ExponentialDemo* program kiírja az *e* értékét, majd meghívja egyenként a fenti táblázatban látható metódusokat egy számra:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;
```

```
        System.out.println("Az e értéke: " +
                           Math.E);
        System.out.println("exp(" + x + ") is " +
                           Math.exp(x));
        System.out.println("log(" + x + ") is " +
                           Math.log(x));
        System.out.println("pow(" + x + ", " + y + ") is " +
                           Math.pow(x, y));
        System.out.println("sqrt(" + x + ") is " +
                           Math.sqrt(x));
    }
}
```

Az *ExponentialDemo* kimenete:

```
Az e értéke: 2.71828
exp(11.635) is 112984
log(11.635) is 2.45402
pow(11.635, 2.76) is 874.008
sqrt(11.635) is 3.41101
```

A *Math* osztály egy sor trigonometrikus függvényt is kínál, ezek a következő táblázatban vannak összefoglalva. A metóduson áthaladó szögek radiánban értendők. Radiánból fokká, és onnan visszakonvertálásra két függvény áll rendelkezésünkre: *toDegrees* és *toRadians*. Előbbi fokká, utóbbi radiánná konvertál. A *Math.PI* függvény meghívásával a PI értékét kapjuk meg a lehető legpontosabban.

double sin(double)	Egy *double* szám szinuszával tér vissza.
double cos(double)	Egy *double* szám koszinuszával tér vissza.
double tan(double)	Egy *double* szám tangensével tér vissza.
double asin(double)	Egy *double* szám arc szinuszával tér vissza.
double acos(double)	Egy *double* szám arc koszinuszával tér vissza.
double atan(double)	Egy *double* szám arc tangensével tér vissza.
double atan2(double)	Derékszögű koordinátákat konvertál (b, a) polárissá (r, théta).
double toDegrees(double) *double toRadians(double)*	A paramétert radiánná vagy fokká konvertálják, a függvények adják magukat.

A következő *TrigonometricDemo* program használja a fenti táblázatban bemutatott összes metódust, hogy különböző trigonometrikus értékeket számoljon ki a 45 fokos szögre:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);
        System.out.println("The value of pi is " +
                           Math.PI);
        System.out.println("The sine of " + degrees +
                           " is " + Math.sin(radians));
        System.out.println("The cosine of " + degrees +
                           " is " + Math.cos(radians));
        System.out.println("The tangent of " + degrees +
                           " is " + Math.tan(radians));
    }
}
```

A program kimenetei:

```
The value of pi is 3.141592653589793
The sine of 45.0 is 0.8060754911159176
The cosine of 45.0 is -0.5918127259718502
The tangent of 45.0 is -1.3620448762608377
```

Az utolsó *Math* metódus, amiről szót ejtünk, a *random*. A metódus egy kvázi-véletlen 0.0 és 1.0 közé eső számmal tér vissza. Pontosabban leírva:

0.0 ≤ Math.random() < 1.0

Hogy más intervallumban kapjunk meg számokat, műveleteket hajthatunk végre a függvény által visszaadott értéken. Például, ha egy egész számot szeretnénk kapni 1 és 10 között, akkor a következőt kell begépelnünk:

```
int number = (int)(Math.random() * 10 + 1);
```

Megszorozva ezt az értéket 10-el a lehetséges értékek intervalluma megváltozik:

0.0 ≤ szám < 10.0.

1-et hozzáadva az intervallum ismét megváltozik:

1.0 ≤ szám < 11.0.

Végül, az érték egésszé konvertálásával egy konkrét számot (int) kapunk 1 és 10 között.

A *Math.random* használata tökéletes, ha egy egyszerű számot kell generálni. Ha egy véletlen számsorozatot kell generálni, akkor egy hivatkozást kell létrehozni a *java.util.Random*–ra, és meghívni ennek az objektumnak a különböző metódusait.

125

```
Random rand = new Random();

int  n = rand.nextInt(50) + 1;
//50 is the maximum and the 1 is our minimum
```

share improve this answer

edited Apr 26 '15 at 14:35

gprathour

5,915 ● 3 ● 24 ● 51

answered May 4 '11 at 17:54

n_yanev

2,654 ● 4 ● 26 ● 65

11. Osztályok létrehozása

Ez a fejezet az osztályok fő alkotóelemeit mutatja be.

Az osztály definíciója 2 fő alkotóelemből áll:

- az osztály deklarációból,
- és az osztály törzsből.

```
class MyClass {
    // tagváltozók, konstruktor és metódus deklarációk
}
```

A kód első sorát osztálydeklarációnak nevezzük.

A következő táblázat bemutatja az összes lehetséges elemet, mely előfordulhat egy osztálydeklarációban előfordulásuk szükséges sorrendjében.

public	(opcionális) az osztály nyilvánosan hozzáférhető
abstract	(opcionális) az osztályt nem lehet példányosítani
final	(opcionális) az osztály nem lehet őse más osztálynak
class NameOfClass	az osztály neve
extends Super	(opcionális) az osztály őse
implement Interfaces	(opcionális) az osztály által implementált interfészek
{ *osztálytörzs* *}*	az osztály működését biztosítja

11.2. Tagváltozók deklarálása

A *Bicycle* a következő kód szerint definiálja tagváltozóit:

```
private int cadence;
private int gear;
private int speed;
```

A *private* kulcsszó mint privát tagokat vezeti be a tagokat. Ez azt jelenti, hogy csak a *Bicycle* osztály tagjai férhetnek hozzájuk.

Nem csak típust, nevet és hozzáférési szintet lehet meghatározni, hanem más attribútumokat is, ideértve azt, hogy a változó-e, vagy konstans. A következő táblázat tartalmazza az összes lehetséges tagváltozó deklarációs elemet.

- A static kulcsszó segítségével a változó az osztály konkrét példányosítása (objektum létrehozása) nélkül is elérhető, ez az érték az objektumok között azonos

hozzáférési szint	(opcionális) A következő négy hozzáférési szint szerint vezérelhető, hogy más osztályok hogyan férhessenek hozzá a tagváltozókhoz: *public*, *protected*, csomag szintű, vagy *private*.
static	(opcionális) Osztályváltozót, és nem példányváltozót deklarál.
final	(opcionális) a változó értéke végleges, meg nem változtatható (konstans) Fordítási idejű hibát okoz, ha a program megpróbál megváltoztatni egy *final* változót. Szokás szerint a konstans értékek nevei nagybetűvel íródnak. A következő változó deklaráció a PI változót határozza meg, melynek értéke π (.3.141592653589793), és nem lehet megváltoztatni: *final double PI = 3.141592653589793;*
típusnév	A változó típusa és neve Mint más változóknak, a tagváltozóknak is szükséges, hogy típusa legyen. Használhatók egyszerű típusnevek, mint például az *int*, *float* vagy *boolean*. Továbbá használhatók referencia típusok, mint például a tömb, objektum vagy interfész nevek. Egy tagváltozó neve lehet minden megengedett azonosító, mely szokás szerint kisbetűvel kezdődik. Két vagy több tagváltozónak nem lehet megegyező neve egy osztályon belül.

11.3. Metódusok deklarálása

A következő példa a *setGear* metódus, amely a sebességváltást teszi lehetővé:

```
public void setGear(int newValue) {
    gear = newValue;
}
```

A metódus deklaráció kötelező elemei: a metódus neve, visszatérő típusa, és egy zárójel-pár: (). A következő táblázat megmutat minden lehetséges részt a metódus deklarációjában.

hozzáférési szint	(opcionális) A metódus hozzáférési szintje
static	(opcionális) Osztály metódust deklarál
abstract	(opcionális) Jelzi, hogy a metódusnak nincs törzse
final	(opcionális) Jelzi, hogy a metódus nem írható felül a leszármazottakban
returnType methodName	Az metódus visszatérő típusa és neve
(paramList)	A paraméterlista a metódushoz

A metódus neve

A metódus neve szokás szerint kis betűvel kezdődik, hasonlóan a változók neveihez.

Általában az osztályon belül egyedi neve van a metódusnak.

Ha a metódus neve, paraméterlistája és visszatérési értéke megegyezik az ősében definiált metódussal, akkor felülírja azt.

Javában az is megengedett, hogy ugyanazzal a névvel, de különböző paraméterlistával hozzunk létre metódusokat.

Nézzük a következő példát:

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(float f) {
        ...
    }
}
```

Azonos paraméterlistával, de különböző típusú visszatérési értékkel nem lehet metódusokat létrehozni.

11.4. Konstruktorok

Minden osztályban van legalább egy konstruktor. A konstruktor inicializálja az új objektumot. A neve ugyanaz kell, hogy legyen, mint az osztályé. Például a *Bicycle* nevű egyszerű osztálynak a konstruktora is *Bicycle*:

```
public Bicycle(int startCadence, int startSpeed,
               int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

Ebben a példában a konstruktor a paraméter alapján tudja létrehozni a kívánt méretű tömböt.

11.7. A *this* kulcsszó használata

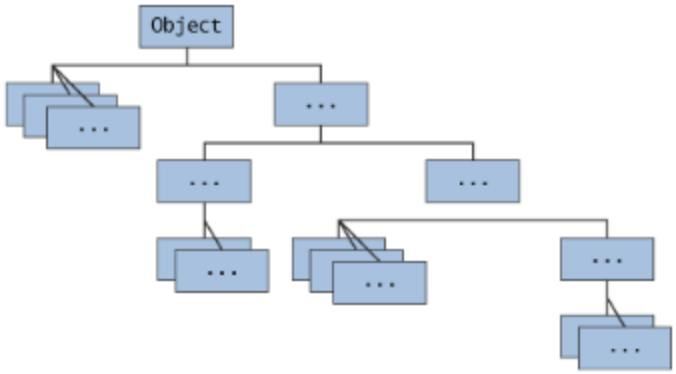
A példa metóduson, vagy a konstruktoron belül a *this* az *aktuális objektumra* való hivatkozást (referenciát) jelenti – azaz arra az objektumra, amelynek a metódusa vagy a konstruktora meghívásra kerül. Az aktuális objektum bármelyik tagja hivatkozható egy példány metódusból, vagy egy konstruktorból a *this* használatával. A leggyakoribb használat oka az, hogy egy változó tag egy paraméter által kerül elfedésre a metódus vagy a konstruktor számára.

Például a következő konstruktor a *HSBColor* osztály számára inicializálja az objektum tagváltozóit a konstruktornak átadott paramétereknek megfelelően. A konstruktor mindegyik paramétere elfed egy-egy objektum tagváltozót, ezért az objektum tagváltozóira a konstruktorban a *this* megadással kell hivatkozzon:

```
public class HSBColor {
    private int hue, saturation, brightness;

    public HSBColor (int hue, int saturation, int brightness){
        this.hue = hue;
        this.saturation = saturation;
        this.brightness = brightness;
    }
}
```

12.Öröklődés

A *java.lang* csomagban definiált *Object* osztály meghatározza és megvalósítja azokat a metódusokat, amelyek minden osztály számára szükségesek. A következő ábrán látható, hogy sok osztály ered az *Object*-ből, majd sok további osztály származik az előbbi osztályokból, és így tovább, létrehozva ezzel az osztályok hierarchiáját.



A hierarchia csúcsán álló *Object* az osztályok legáltalánosabbja. A hierarchia alján található osztályok sokkal specializáltabb viselkedést eredményeznek. Egy leszármazott osztály valamely osztályból származik. A *superclass* kifejezés (továbbiakban szülőosztály vagy ősosztály) egy osztály közvetlen ősére/elődjére, vagy annak bármely felmenő osztályára utal. Minden osztálynak csak és kizárólag egyetlen közvetlen szülőosztálya van.

Ha egy leszármazott osztály egy osztálymetódust ugyanazzal az aláírással definiál, mint a felsőbb osztálybeli metódus, akkor a leszármazott osztály metódusa elrejti (másként fogalmazva elfedi) a szülőosztálybelit. Nagy jelentősége van az elrejtés és a felülírás megkülönböztetésének. Nézzük meg egy példán keresztül, hogy miért! E példa két osztályt tartalmaz. Az első az *Animal*, melyben van egy példánymetódus és egy osztálymetódus:

```
public class Animal {
    public static void hide() {
        System.out.println("The hide method in Animal.");
    }
    public void override() {
        System.out.println("The override method in Animal.");
    }
}
```

A második osztály neve *Cat*, ez az *Animal*-nak egy leszármazott osztálya:

```
public class Cat extends Animal {
    public static void hide() {
        System.out.println("The hide method in Cat.");
    }
    public void override() {
        System.out.println("The override method in Cat.");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = (Animal)myCat;
        myAnimal.hide();
        myAnimal.override();
    }
}
```

A *Cat* osztály felülírja az *override* metódust az *Animal*-ban, és elrejti a *hide* osztálymetódust az *Animal*-ban. Ebben az osztályban a *main* metódus létrehoz egy *Cat* példányt, beteszi az *Animal* típusú hivatkozás alá is, majd előhívja mind az elrejtett, mind a felülírt metódust. A program eredménye a következő:

```
The hide method in Animal.
The override method in Cat.
```

A szülőosztályból hívjuk meg a rejtett metódust, a leszármazott osztályból pedig a felülírtat. Osztálymetódushoz a futtatórendszer azt a metódust hívja meg, mely a hivatkozás szerkesztési idejű típusában van definiálva, amellyel a metódust elnevezték. A példánkban az *myAnimal* szerkesztési idejű típusa az *Animal*. Ekképpen a futtatórendszer az *Animal*-ban definiált rejtett metódust hívja meg. A példánymetódusnál a futtatórendszer a hivatkozás futásidejű típusában meghatározott metódust hívja meg. A példában az *myAnimal* futásidejű típusa a *Cat*. Ekképpen a futtatórendszer a *Cat*-ban definiált felülíró metódust hívja meg.

12.3. A *super* használata

Ha egy metódus felülírja az ősosztálya metódusainak egyikét, akkor a *super* használatával segítségül hívható a felülírt metódus. A *super* arra is használható, hogy egy rejtett tag variánsra utaljunk. Ez a szülőosztály:

```
public class Superclass {
    public boolean aVariable;

    public void aMethod() {
        aVariable = true;
    }
}
```

Most következzen a *Subclass* nevű leszármazott osztály, mely felülírja *aMethod*-ot és *aVariable*-t:

```
public class Subclass extends Superclass {
    public boolean aVariable; //hides aVariable in Superclass
    public void aMethod() { //overrides aMethod in Superclass
        aVariable = false;
        super.aMethod();
        System.out.println(aVariable);
        System.out.println(super.aVariable);
    }
}
```

A leszármazott osztályon belül az *aVariable* név a *SubClass*-ban deklaráltra utalt, amely a szülőosztályban deklaráltat elrejti. Hasonlóképpen, az *aMethod* név a *SubClass*-ban deklaráltra utalt, amely felsőbb osztályban deklaráltat felülírja. Tehát ha egy a szülőosztályból örökölt *aVariable*-ra és *aMethod*-ra szeretnénk utalni, a leszármazott osztálynak egy minősített nevet kell használnia, használva a *super*-t, mint azt láttuk. A *Subclass* *aMethod* metódusa a következőket írja ki:

```
false
true
```

Használhatjuk a *super*-t a konstruktoron belül is az ősosztály konstruktora meghívására. A következő kódpélda bemutatja a *Thread* osztály egy részét – az osztály lényegében többszálú programfutást tesz lehetővé –, amely végrehajt egy animációt. Az *AnimationThread* osztály konstruktora beállít néhány kezdeti értékeket, ilyenek például a keretsebesség és a képek száma, majd a végén letölti a képeket:

```
class AnimationThread extends Thread {
    int framesPerSecond;
    int numImages;
    Image[] images;

    AnimationThread(int fps, int num) {
        super("AnimationThread");
        this.framesPerSecond = fps;
        this.numImages = num;
        this.images = new Image[numImages];

        for (int i = 0; i <= numImages; i++) {
            ...
            // Load all the images.
            ...
        }
    }
    ...
}
```

A félkövérrel szedett sor a közvetlen szülőosztály konstruktorának explicit meghívása, melyet a *Thread* nyújt. Ez a *Thread* konstruktor átvesz egy *String*-et, és így nevezi a *Thread*-et. Ha a leszármazott osztály konstruktorában van explicit *super* konstruktorhívás, akkor annak az elsőnek kell lennie. Ha egy konstruktor nem hív meg explicit módon egy szülőosztálybeli konstruktort, akkor a Java futtatórendszer automatikusan (implicit)

16.Interfészek

Az interfész olyan viselkedéseket definiál, amelyet az osztályhierarchia tetszőleges osztályával megvalósíthatunk. Egy interfész metódusok halmazát definiálja, de nem valósítja meg azokat. Egy konkrét osztály megvalósítja az interfészt, ha az összes metódusát megvalósítja.

Definíció: Az interfész implementáció nélküli metódusok névvel ellátott halmaza.

Mivel az interfész a megvalósítás nélküli, vagyis absztrakt metódusok listája, alig különbözik az absztrakt osztálytól. A különbségek:

- Az interfész egyetlen metódust sem implementálhat, az absztrakt osztály igen.
- Az osztály megvalósíthat több interfészt, de csak egy ősosztálya lehet.
- Az interfész nem része az osztályhierarchiának. Egymástól "független" osztályok is megvalósíthatják ugyanazt az interfészt.

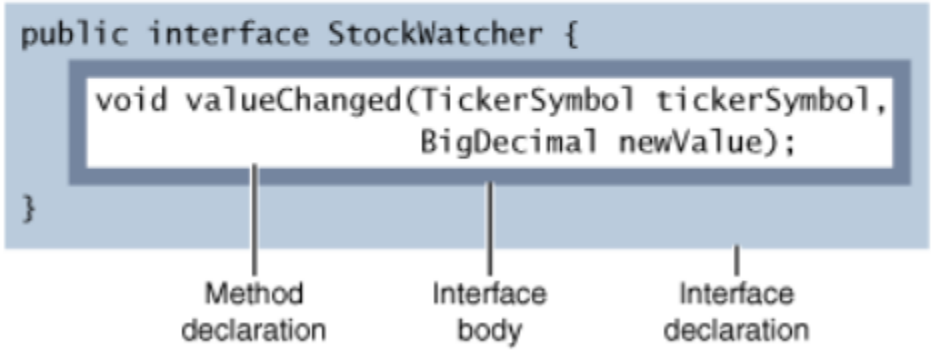
Ebben a fejezetben egy olyan példaprogramot fogjuk használni, ami a Tervezési minták (*Design Patterns*) között szokás emlegetni megfigyelő (*Observer*) néven. Ez az osztály megengedi más osztályoknak, hogy regisztrálják magukat bizonyos adatai változásának figyelésére. A *StockApplet* osztály fog megvalósítani egy olyan metódust, amivel regisztrálni tud a változás figyeléséhez:

```
public class StockMonitor {
      public void watchStock(StockWatcher watcher,
                             TickerSymbol tickerSymbol,
                             BigDecimal delta) {
      ...
      }
}
```

Az első paraméter egy *StockWatcher* objektum. A *StockWatcher* annak az interfésznek a neve, amelyet hamarosan látni fogunk. Az interfész egyetlen *valueChanged* nevű metódust definiál. Egy objektum akkor tudja magát megfigyelőként regisztrálni, ha az osztálya megvalósítja ezt az interfészt. Amikor a *StockMonitor* osztály érzékeli a változást, meghívja a figyelő *valueChanged* metódusát.

16.1. Interfész definiálása

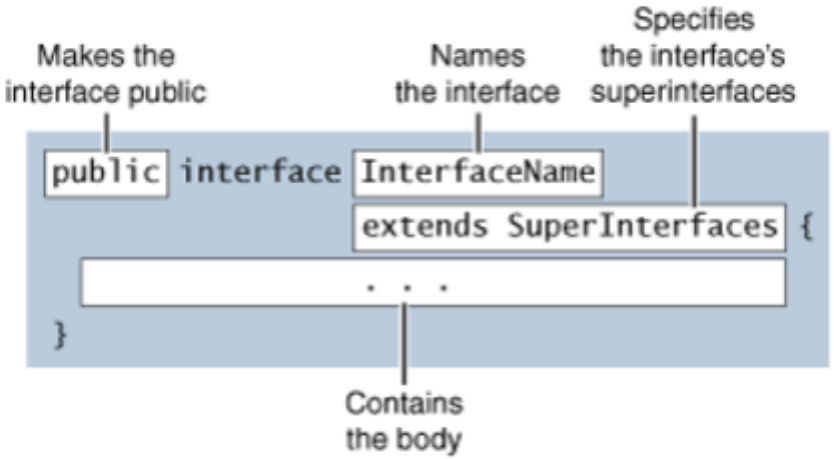
A következő ábra bemutatja az interfész definíció két összetevőjét: az interfész deklarációt és a törzset.



A *StockWatcher* interfész deklarálja, de nem implementálja a *valueChanged* metódust. Az interfészt megvalósító osztályok fogják a metódust implementálni.

Interfész deklaráció

A következő ábra az interfész deklaráció minden részét bemutatja:



Az interfész deklarációban két elem kötelező: az *interface* kulcsszó és az interfész neve. Ez után szerepelhetnek a szülőinterfészek.

Az interfész törzs

Az interfész törzs metódus deklarációkat tartalmaz ';'-el lezárva. Minden deklarált metódus értelemszerűen publikus és absztrakt (bár maguk a kulcsszavak nem írhatók ki).

16.2. Interfészek implementálása

Egy interfész viselkedési formákat definiál. Az interfészt megvalósító osztály deklarációjában szerepel az *implements* záradék. Az osztály akár egy vagy több interfészt is megvalósíthat. (Az interfészek között megengedett a többszörös öröklődés.)

Konvenció: az *implements* záradék az *extends* záradékot követi, ha mindkettő van.

A következő példa applet implementálja a *StockWatcher* interfészt.

```
public class StockApplet extends Applet
                         implements StockWatcher {

    public void valueChanged(TickerSymbol tickerSymbol,
                             BigDecimal newValue) {
        switch (tickerSymbol) {
            case SUNW:
                ...
                break;
            case ORCL:
                ...
                break;
            case CSCO:
                ...
                break;
            default:
                // handle unknown stocks
                ...
                break;
        }
    }
}
```

Amikor egy osztály megvalósít egy interfészt, akkor alapvetően aláír egy szerződést. Az osztálynak implementálni kell az interfészben és szülőinterfészeiben deklarált összes metódust, vagy az osztályt absztraktként kell deklarálni. (Az ilyen absztrakt osztályoknak csak akkor lehet nem absztrakt egy leszármazottja, ha az ezen az öröklési szinten meg nem valósított metódusokat már maradéktalanul megvalósítja.)

A *StockApplet* megvalósítja a *StockWatcher* interfészt, azaz szolgáltatásként nyújtja a *valueChanged* metódust.