0

04

Java SE 8 Documentation (http://docs.oracle.com/javase/8/docs/)

Feladatok

Osztályok definíciója

A Java nyelvben az absztrakció egyik eszköze az osztály. Segítségével több különböző módon strukturálhatjuk a programjainkat: el tudjuk rejteni az implementációs részleteket, egységbe tudjuk zárni az összetartozó fogalmakat, illetve saját típust tudunk hozzáadni a nyelvhez, a rajta értelmezett műveletekkel együtt. Minden esetben nagyon fontos egy ún. interfész kialakítása, amelyen keresztül manipulálni tudjuk az osztállyal leírt objektumok belső állapotát.

Egy osztálydefiníció tartalmazhat adattagokat, metódusokat (tagfüggvényeket) és egyéb osztályokat (ez utóbbiról majd a későbbiekben lesz szó). Mindezek szemléltetéséhez nézzünk meg egy köröket ábrázoló `Circle` osztály és annak a műveleteit! Az egyes osztályok definícióit célszerű külön állományba tenni, amelyek így külön fordítási egységek lesznek.

```
class Circle {
    Point2D center;
    double  radius;
```



```
    /*
     * Feltételezés: származtatott értékek, vagyis ezek az értékek a sugárral
     * együtt változnak.
     */
    double  area;
    double  circumference;

    // Az értékek konkrét származtatása.
    void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    void scale(double k) {
        radius *= k;
        derive();
    }

    String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

Ahogy korábban már megismerhettük, a `static` módosítóval ellátott metódusok osztályszintűek lesznek, vagyis nem szükséges aktív objektumpéldány a használatukhoz. Ezek tulajdonképpen a hagyományos nyelvek függvényfogalmához állnak a legközelebb.

Láthatósági módosítók

Ez a stílusú programozás programozás azonban tekinthető az absztrakció szempontjából megbízhatónak. Ebben a felírásban ugyanis az objektum állapotát kívülről tetszőleges módon meg tudjuk változtatni, amellyel így akár akaratlanul (vagy netalán szándékosan) megsértünk bizonyos feltételezéseket. Mivel az implementációt ezekre a feltételezésekre alapoztuk, így lényegében hibás működést tudunk előidézni ezzel.

```
Circle c = new Circle();

c.radius = 2.0;
// Itt a c.area és c.circumference még mindig 0.0!
```

A másik fontos tudnivaló, hogy az osztályokban a metódusok viselkedése a felhasználó számára mindig lényegében egy fekete doboz. Ha megengedjük számára, hogy elérje az objektum belső állapotát, akkor olyan módon fogja használni az osztályunkat, amelyik mindezen ismereteket kihasználja. Ha viszont

később módosítunk az osztály implementációján, akkor a felhasználó kódja elromlik, amely fordítási hibákhoz vagy akár nagyon rejtett programhibákhoz vezethet.


```
Circle c = new Circle();
c.center.x += dx;
c.center.y += dy;
```

Mindezek elkerülésére hoztak létre további módosítószavakat, amelyekkel az osztály definíciójában szereplő adattagok és metódusok láthatóságát szabályozhatjuk. Ezek a módosítószavak a következők:

- `public` : Az kívülről teljesen szabadon elérhető eleme, mindenféle korlátozás nélkül használhatjuk például az adott metódust vagy adattagot. Ezt korábban már használtuk, mivel csomagokból az ilyen módosítóval ellátott nevek látszanak, valamint a főprogramnak is ilyennek kell lennie, hogy futtatható legyen.
- `protected` : Csak adott csomagon belülről, valamint a származtatott osztálydefiníciókból érhető el csak. Ezzel majd később, a származtatás megismerése után fogunk foglalkozni.
- `private` : Kizárólag csak az adott osztályon belül érhető el.
- Nincs módosítószó: Az ún. "package private" láthatóság, amikor csak az adott csomagon belülről érthető el. Ez így lényegében az alapértelmezett láthatóság, vagyis ezt használtuk eddig, ha nem `public` volt valami.

Ennek értelmében láthatóság szerint a következő relációt írhatjuk fel. A relációval azt tekintjük mindig nagyobbnak, amelyik többet enged láttatni az objektumból.

`public` > (nincs) > `protected` > `private`

Ezek segítségével most már átalakíthatjuk az előbbi definíciónkat úgy, hogy biztosan fenntartsa az implementációra vonatkozó feltételezéseinket.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

Ennek viszont az az ára, hogy az eddig bárki által elérhető, `public` adattagjainkat sem tudjuk már elérni. Ezért kell ilyenkor mindegyik adattaghoz készíteni egy lekérdező ("getter") metódust, amellyel az adattag értékét tudjuk lekérni. Értelemszerűen a lekérdező metódus már bárki által hívható lesz.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    public Point2D getCenter() {
        return center;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

A referenciák esetében azonban vigyáznunk kell, hogy a lekérdező metódus ne közvetlenül magát a referenciát adja ki, hanem mindig másolja le az általa hivatkozott objektumot. Ennek elmulasztása esetén ugyanis kiszivárogtatunk egy referenciát az objektum belső állapotára, amellyel így kívülről ismét megváltoztathatóvá válik!

```
Circle c = new Circle();

c.getCenter().x += dx;
c.getCenter().y += dy;
```

Ezért ezt a hibánkat most javítsuk ki!


```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

Ha egy adattaghoz csak lekérdező metódust írunk, akkor kívülről lényegében írásvédetté válik. Ha kívülről írhatóvá akarjuk tenni, akkor továbbra sem az a helyes megoldás, hogy `public` láthatóságot adunk neki, hanem készítünk hozzá egy beállító ("setter") metódust. A beállító metódus előnye a nyers eléréssel

szemben, hogy segít az implementáció során meglevő feltételezéseinket fenntartani és szükség esetén más olyan adattagok értékét is módosítani, amelyek valamilyen módon függenek az eredetileg módosítandótól. Ha referenciáról van szó, akkor ne felejtsük el megint másolni az objektumot!

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public void setCenter(Point2D center) {
        this.center = new Point2D();

        this.center.x = center.x;
        this.center.y = center.y;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
        derive();
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }
```

```
    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
                center.show(), radius);
    }
}
```


Konstruktorok

Az osztályokhoz tartozó objektumok létrehozásáért speciális metódusok, ún. konstruktorok felelősek. Korábban is láthattuk már, hogy példányokat úgy hozunk létre, hogy a `new` operátor után az osztály nevét adjuk meg egy üres zárójelpárral. Ez tulajdonképpen az alapértelmezett konstruktor meghívása volt. Az alapértelmezett konstruktor mindig létrejön az osztályhoz, ha nem adunk meg a definíciójában konstruktort.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    /* Ezt nem kell külön definiálni, magától is létrejön, ha nem adunk meg
     * konstruktort.
     */
    public Circle() {
        center        = null;
        radius        = 0.0;
        area          = 0.0;
        circumference = 0.0;
    }

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public void setCenter(Point2D center) {
        this.center = new Point2D();

        this.center.x = center.x;
        this.center.y = center.y;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
        derive();
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
```

```
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```


Természetesen mi magunk is készíthetünk konstruktort az osztályhoz. A konstruktor metódusként viselkedik, noha nincs visszatérési értéke, viszont lehet paraméterlistája és vonatkozhatnak rá a láthatósági módosítók. Leginkább úgy képzelhető el, mint egy eljárás, amely az objektum létrehozását követően meghívódik. Ekkor lehetőségünk az objektum kezdőállapotát közvetlenül a konstruktor paramétereitől függően beállítani.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    /* Ilyenkor viszont már nincs paraméter nélküli konstruktorunk.  Ha kell,
     * akkor nekünk kell már megadni (ld. lentebb).
     */
    public Circle(double cx, double cy, double radius) {
        center = new Point2D();
        center.x = cx;
        center.y = cy;
        this.radius = radius;
        derive();
    }

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public void setCenter(Point2D center) {
        this.center = new Point2D();

        this.center.x = center.x;
        this.center.y = center.y;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
        derive();
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
```

```
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```


Érdemes azonban megjegyezni, hogy a konstruktor paraméterei lehetnek olyanok, amelyek az objektumokkal kapcsolatos valamilyen korábbi feltételezéseinkkel ellentétesek. Ha emiatt nem felelnének meg, akkor viszont már nincs lehetőségünk az objektumpéldány "visszacsinálására". Ezért ezt például úgy szokták megoldani, hogy a konstruktort `private` láthatóságúvá tesszük, és egy osztályhoz tartozó, `public` láthatóságú metódussal hozzuk létre az osztály objektumainak példányait. Ha ez nem sikerül, mert például a megadott paraméterek nem felelnek meg az elvárásainknak, akkor `null` referenciát adunk vissza.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    /* A kör sugara csak pozitív lehet.
     */
    public static Circle make(double cx, double cy, double radius) {
        return ((radius > 0.0) ? new Circle(cx, cy, radius) : null);
    }

    private Circle(double cx, double cy, double radius) {
        center = new Point2D();
        center.x = cx;
        center.y = cy;
        this.radius = radius;
        derive();
    }

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public void setCenter(Point2D center) {
        this.center = new Point2D();

        this.center.x = center.x;
        this.center.y = center.y;
    }

    public double getRadius() {
        return radius;
    }

    /* Később sem engedjük a sugarat negatívvá tenni.
     */
    public void setRadius(double radius) {
        if (radius <= 0.0)
           return;

        this.radius = radius;
        derive();
    }

    public double getArea() {
        return area;
```

```
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

Az objektumok felszabadításával nem kell foglalkoznunk, a Java nyelvben ugyanis nincs destruktor. Természetesen előfordulhatnak olyan esetek, amikor az objektum eldobásakor el kell végeznünk valamilyen műveletet (például elengedni egy addig használt erőforrást, állományt), de ezt másképpen oldják meg.

Túlterhelés

A konstruktorokhoz kapcsolódóan érdemes megjegyezni, hogy az osztályunk metódusainak és konstruktorainak lehet többféle paraméterezése is. Ezt túlterhelésnek (overloading) nevezzük. Azért, mert ilyenkor egyetlen névhez társítunk többféle viselkedést, vagyis igazából több metódust nevezünk el ugyanazzal a névvel. Ezeket a változatokat tulajdonképpen a paraméterlista alapján választjuk ki. Visszatérési értékre azonban nem lehet metódusokat túlterhelni.

```
class Circle {
    private Point2D center;
    private double  radius;
    private double  area;
    private double  circumference;

    public static Circle make(double cx, double cy, double radius) {
        return ((radius > 0.0) ? new Circle(cx, cy, radius) : null);
    }

    public static Circle make(double cx, double cy) {
        return new Circle(cx, cy);
    }

    public static Circle make(double radius) {
        return make(0.0, 0.0, radius);
    }

    public static Circle make() {
        return new Circle();
    }

    private Circle(double cx, double cy, double radius) {
        center = new Point2D();
        center.x = cx;
        center.y = cy;
        this.radius = radius;
        derive();
    }

    private Circle(double radius) {
        this(0.0, 0.0, radius);
    }

    private Circle(double cx, double cy) {
        this(cx, cy, 1.0);
    }

    private Circle() {
        this(0.0, 0.0, 1.0);
    }

    public Point2D getCenter() {
        Point2D result = new Point2D();

        result.x = center.x;
        result.y = center.y;
        return result;
    }

    public void setCenter(Point2D center) {
```

```
        this.center = new Point2D();

        this.center.x = center.x;
        this.center.y = center.y;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        if (radius <= 0.0)
           return;

        this.radius = radius;
        derive();
    }

    public double getArea() {
        return area;
    }

    public double getCircumference() {
        return circumference;
    }

    private void derive() {
        area          = radius * radius * Math.PI;
        circumference = 2 * radius * Math.PI;
    }

    public void translate(double dx, double dy) {
        center.translate(dx, dy);
    }

    public void scale(double k) {
        radius *= k;
        derive();
    }

    public String show() {
        return String.format("C { center = %s, radius = %.3f }",
          center.show(), radius);
    }
}
```

Dinamikus méretű tömbök

A programjainkban olykor hasznos lehet, hogy futás közben változó méretű tömbökkel dolgozzunk. Ugyan nem teljesen tömbök, de erre a célra használhatóak a `java.util.ArrayList` (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) típusú objektumok. A

`java.util.ArrayList` (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) paraméterezhető az elemek típusával, így lényegében tud tömbként viselkedni.

A fontosabb műveletek:

- Lista létrehozása: A létrehozás során ügyelnünk kell arra, hogy a listákat a bennük tárolt elemek típusával paraméterezzük. Ezt `<` és `>` jelek közt kell megadnunk (ezért a listákat később majd sablonnak nevezzük). Itt viszont csak osztálynév szerepelhet, így, ha primitív típust akarunk elhelyezni bennük, akkor azok csomagoló osztályai kell használnunk!

```
ArrayList<Integer>   listOfInts    = new ArrayList<Integer>();   // nem pedig
ArrayList<String>    listOfStrings = new ArrayList<String>();
ArrayList<Character> listOfChars   = new ArrayList<Character>(); // és nem Arr
```

- Új elem hozzáadása a lista végéhez: A létrehozás során kapunk egy referenciát, amelyre meghívhatjuk az `add()` műveletet és ezzel felvehetünk egy elemet a lista végére. Természetesen a típusnak egyeznie kell a listában tárolt elemek típusával.

```
listOfInts.add(1);           // az autoboxing miatt ez működik
listOfStrings.add("hello");
listOfStrings.add("world");
```

- Másik lista hozzáfűzése a lista végéhez: Egy komplett listát is hozzá tudunk fűzni a meglevőhöz, az elemek típusának megtartásával.

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
ArrayList<Integer> l2 = new ArrayList<Integer>();
l1.add(1); l1.add(2); l1.add(3);
l2.add(4); l2.add(5); l2.add(6);
l1.addAll(l2);  // = l1 + l2
```

- Elem beszúrása egy adott pozícióra: Az adott indexen létrejön egy szabad hely, ahova az új elem értéke bemásolódik. A beszúrandó érték típusának meg kell egyeznie a vektorban tárolt elemek típusával. A számozás 0-tól kezdődik.

```
ArrayList<String> l = new ArrayList<String>();
l.add("hello"); l.add("there");
l.add(1, "world"); // = "hello", "world", "there"
```

- Összes tárolt elem törlése: Minden korábban eltárolt érték eltűnik a listából, a hossza nulla lesz.

```
ArrayList<Circle> l = new ArrayList<Circle>();
l.add(Circle.make()); l.add(Circle.make(1.0, 2.0, 3.0));
l.clear();
```

- Elem elérése az adott pozíción: Nulla és a vektor hossza mínusz egy között kell lennie a pozíció értékének. Tehát így a nulla mindig a vektor első elemére vonatkozik.

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(3);
int anElem = l.get(1);
```


- Elem módosítása adott pozíción:

```
ArrayList<Boolean> l = new ArrayList<Boolean>();
l.add(true); l.add(false); l.add(true);
l.set(1, true);
```

- Aktuális méret lekérdezése: A listában tárolt elemek számának lekérdezése. Mivel ez használat közben változhat, ezért ez egy metódus.

```
ArrayList<Byte> l = new ArrayList<Byte>();
l.add((byte) 1);
int lLength = l.size();
```

- Adott értékű elem keresése: Megkeresi a listában a paraméterként megadott érték első előfordulását és visszaadja annak az indexét. Ha nincs ilyen, akkor az eredménye `-1` lesz.

```
ArrayList<Character> l = new ArrayList<Character>();
l.add('f'); l.add('o'); l.add('o'); l.add('b'); l.add('a'); l.add('r');
int firstO = l.indexOf('o');
int firstA = l.indexOf('a');
int firstZ = l.indexOf('z');  //  = -1
```

- Elem törlése adott pozícióról:

```
ArrayList<Boolean> l = new ArrayList<Boolean>();
l.add(false);
l.remove(0);  // = üres lista
```

- Tömbbé alakítás: A listát bármikor vissza tudjuk alakítani tömbbé. Ekkor viszont arra kell vigyázni, hogy az erre használatos `toArray()` metódusnak adnunk kell olyan típusú tömböt, amire alakítani akarjuk. A benne tárolt értékek teljesen lényegtelenek. (Ez a Java nyelv korlátozásaiból fakad.)

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(3); l.add(4);
Integer[] intArray = l.toArray(new Integer[0]);  // = { 1, 2, 3, 4 }
```

Feladatok

- Készítsük a `utils.IntList` osztály implementációját! Ezt az osztályt `int` értékek tömbösített, dinamikus tárolására kell tudnunk használni és nagyban hasonlítani fog a `java.util.ArrayList` (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) osztályhoz. Legyenek ilyen konstruktorai:
  - nincs semmilyen paramétere (ilyenkor a tömb még nem tartalmaz semmit),
  - meg lehet adni egy `int` értékeket tartalmazó tömbbel, hogy mit tároljon kezdetben,

valamint legyenek a következő műveletei:

- `add()` : egy új elem hozzáadása a tömb végére,
- `add()` : elem beszúrása az adott indexre (túlterhelt változat),
- `concat()` : egy másik `IntList` tartalmának hozzáfűzése az aktuálishoz,
- `get()` : az első elem lekérdezése
- `get()` : adott indexű elem lekérdezése (túlterhelt változat),
- `set()` : adott indexű elem beállítása, amennyiben van olyan indexű elem,
- `remove()` : az első elem törlése,
- `remove()` : adott indexű elem törlése (túlterhelt változat),
- `indexOf()` : elem indexének megkeresése, ha nem található, akkor `-1` ,
- `size()` : a jelenleg tárolt elemek száma,
- `clear()` : az összes elem törlése,
- `toArray()` : az osztályban tárolt értékeket egyetlen tömbként történő visszaadása,
- `show` : szöveggé alakítás.

és legyen egy olyan, `read()` nevű osztályszintű metódusa, amellyel szövegből tudjuk beolvasni a tárolni kívánt elemeket. Az értékeket ebben a beolvasni kívánt szöveges reprezentációban egyszerűen szóközökkel választjuk el. Ellenőrizzük azt is, hogy a szövegben valóban számok szerepelnek! Ha ez nem teljesül és nem tudjuk beolvasni az összes elemet, akkor adjuk vissza `null` referenciát!

Ügyeljünk arra, hogy az osztály semmilyen más egyéb eleme ne legyen a külvilág számára elérhető!

- Készítsünk egy olyan programot, amely a szabványos bemenetet soronként olvassa, majd annak befejeződésekor fordított sorrendben kiírja az addig beolvasott sorokat a szabványos kimenetre! Használjuk a `java.util.ArrayList` (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) osztályt!
- Írjunk egy `IntTree` osztályt, amely egy egész számokat rendezetten tároló bináris fát ábrázol! A bináris fa értékek olyan láncolata, ahol minden értéknek nulla, egy vagy két rákövetkezője lehet, és csak nulla vagy egy megelőzője. A láncolást `IntTree` objektumokra vonatkozó referenciákkal oldjuk meg, ezek fognak mutatni az adott fabeli csomópont bal- és jobb részfáira. Az csomópontokban ezenkívül még egy `int` értéket is eltárolunk. Ha nincs megelőző, akkor a fa gyökeréről beszélünk.

  Legyenek a következő műveleteink:

  - `insert()` , amely megkap egy `int` értéket és beilleszti a fába, annak értékétől függően. Ha a beszúrandó elem kisebb, a fa gyökerében található elemnél, akkor illesszük be a bal részfába (rekurzió). Ha nincs még bal részfánk, akkor hozzuk létre azzal az elemmel a gyökerében! Ugyanezt végezzük el a jobb részfára, ha a beszúrandó elem nagyobb, vagy egyenlő, mint a gyökérben levő!
  - `contains()` , amely eldönti, hogy a paramétereként megadott elem megtalálható-e a fában! Ezt egy logikai értékkel adja vissza: igaz, ha igen, hamis, ha nem! (Ez is rekurzív metódus lesz.)
  - egy olyan konstruktor, amely megkapja `int` értékek egy tömbjét és azokat beszúrja a fába.
  - egy olyan konstruktor, amely egy elemből létrehoz egy olyan fát, amelyben csak egyetlen gyökérelem van (és nincsenek rákövetkezői).
  - `toArray()` , amely egy `int` értékeket tartalmazó tömbben visszaadja azokat az értékeket, amelyeket a fában tárolunk! Ha szeretnénk kihasználni, hogy a beszúrást rendezetten végeztük, akkor érdemes ezt a tömböt úgy felépíteni, hogy először a bal részfa elemeit vesszük, aztán a

gyökérben levő elemet, majd a jobb részfa elemeit. Az így összeállított tömb is rendezett lesz, ezt nevezik inorder bejárásnak. (Természetesen ez a metódus is rekurzív.)

- `show`, amely megadja egy fa szöveges alakját. Itt most elegendő, ha csak a tárolt elemeket jelenítjük meg egy felsorolásban.
- `equalsTo()`, amely eldönti az objektumról és a paraméteréről, hogy a kettő megegyezik-e. Két bináris fát akkor tekintsünk egyenlőnek, ha ugyanazokat az elemeket tartalmazzák!

Linkek

Kapcsolódó forráskódok (http://oktnb127.inf.elte.hu/javagyak/04/tree/master)