Minden feladatot kötelező beküldeni!

A beküldési határidő: október 5.

A GitHub-ra kell feltölteni, az órán megbeszéltek szerint!

1. Készítsünk egy Pont nevű osztályt, amely egy kétdimenziós pont modellezésére szolgál. A nagyon lebutított kis osztályunkban most mindössze két adatot tárolunk erről a modellezendő pontról: a két koordinátáját (valósak), amelyek **privát** láthatóságúak legyenek!
   Az osztályunk tartalmazza a Pont „életre keltésére", megkonstruálására szolgáló KONSTRUKTORT, amivel be tudjuk állítani létrehozáskor a két koordinátát.
   Majd írjunk mind a két koordinátához beállító és lekérdező metódust is („Getter és Setter").
   Ezután hozzunk létre egy Main nevű osztályt, amely csak a „fő függvényt" tartalmazza(psvm+TAB). A fő függvényben „keltsük életre" a Pontokat: „hozzunk a világra" 4 darab pontot (ezt a new kulcsszóval tehetjük meg). Ezt követően módosítsuk(a módosítást a „Setter-rel tudjuk csak megtenni, mivel a koordináta privát láthatóságú") az első két pont y koordinátáját az *eredeti érték+5*-re (az eredeti értéket ne mi írjuk be direkt módon, hanem kérdezzük le…azért van a getter), és a második két pont x koordinátáját *előző érték-3.4*-re.
   A módosított koordinátákat írjuk ki a képernyőre (persze megint a lekérdező metódust „Getter" használjuk a println-ben).

2. (*Geometria: n-oldalú szabályos sokszög.*) Egy *n*-oldalú szabályos sokszögben minden oldal azonos hosszúságú és minden szög azonos nagyságú (fokú), azaz a sokszög egyenlő oldalú és egyenlő szögű. Tervezzen egy RegularPolygon nevű Java osztályt, amely tartalmaz
   - egy int típusú, n nevű privát adatmezőt, amely a sokszög oldalainak a számát definiálja, alapértelmezés szerint 3-as értékkel;
   - egy double típusú, side nevű privát adatmezőt, amely az oldal hosszát tárolja, alapértelmezés szerint 1-es értékkel;
   - egy double típusú, x nevű privát mezőt, amely a sokszög középpontjának *x*-koordinátáját definiálja, **alapértelmezés szerint 0 értékkel(ezt az alpaértelmezett konsruktor állítja be->lásd lentebb a pédát!!);**

- egy double típusú, y nevű privát mezőt, amely a sokszög középpontjának *y*-koordinátáját definiálja, **alapértelmezés szerint 0 értékkel(ezt az alpaértelmezett konsruktor állítja be->lásd lentebb a pédát!!);**
- egy paraméter nélküli konstruktort, amely egy szabályos sokszöget hoz létre az **alapértelmezett értékekkel,** ez egész pontosan az alábbi módon fog kinézni:

  public RegularPolygon(){

  this.n=3;

  this.side=1;

  this.x=0;

  this. y=0;

  }

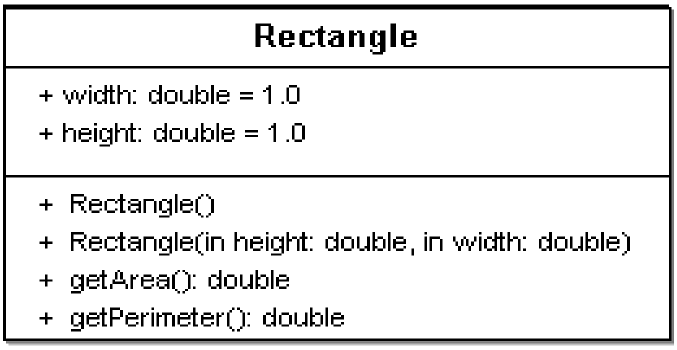
  Továbbá egy konstruktort, amely a megadott oldalszámmal és oldalhosszal hoz létre egy szabályos sokszöget, amelynek a középpontját a (0, 0) pontba helyezi; (public RegularPolygon(int n, double side)…a másik két értéket nem paraméterrel állítjuk be, alapértelmezetten nullát írunk a konstruktor tözsébe: erre gondolok: this.x=0; this. y=0;)

- egy konstruktort, amely a megadott oldalszámmal, oldalhosszal, *x*- és *y*-koordinátákkal hoz létre egy szabályos sokszöget;
- lekérdező és beállító metódusokat mind a négy adatmezőhöz;(get és set)
- a getPerimeter() metódust, amely visszaadja a sokszög kerületét;
- a getArea() metódust, amely visszaadja a sokszög területét.
- toString() metódust, amely kiírja a sokszöget, az általatok kreatívnak tartott módon

Implementálja az osztályt! Írjon egy Teszt nevű osztályt, ebben a **main** metódust, amelyben létrehoz három RegularPolygon objektumot, egyet a paraméter nélküli, egyet a RegularPolygon(6, 4), egyet pedig

a RegularPolygon(10, 4, 5.6, 7.8) konstruktor használatával! Írassa ki mindegyik objektum kerületét és területét!

3. (*A **Rectangle** osztály.*) Tervezzen egy **Rectangle** nevű Java osztályt, amely egy téglalapot reprezentál.



Az osztály tartalmazzon

- egy width és egy height nevű, double típusú mezőt, amely téglalap szélességét és magasságát adja meg; az alapértelmezés szerint mind awidth, mind a height értéke 1 legyen;
- egy paraméter nélküli konstruktort, amely egy alapértelmezett téglalapot hoz létre;(lásd első feladat esetén adott mintát)
- egy konstruktort, amely megadott width és height értékekkel hoz létre egy téglalapot;
- egy getArea() nevű metódust, amely az aktuális téglalap területének az értékét adja vissza;
- egy getPerimeter() nevű metódust, amely az aktuális téglalap kerületének az értékét adja vissza!
- Ne felejtsen el gondoskodni a toString() metódus megírásáról sem!

Implementálja az osztályt! Írjon egy teszt programot, amely létrehoz két Rectangle típusú objektumot — egy 4 szélességű és 40 magasságú, valamint egy 3.5 szélességű és 35.9 magasságú téglalapot!

Írassa ki az egyes téglalapok szélességét, magasságát, területét és kerületét ebben a sorrendben a standard kimenetre!

4. Tekintse az alábbi Java kódot:

```
public class Ital {
    protected String név;
    protected String kiszerelés;
    private  static int ár;
    protected Date gyártásiDátum;
}
```

Egészítse ki az osztályt egy konstruktorral, amelynek segítségével a név és kiszerelés adattagjának kezdőérték adható! A gyártásiDátum az ital létrehozását/gyártásának idejét tárolja, amit nem paraméterként adunk át a konstruktornak, hanem annak belsejében, a lefutásakor hozunk létre(lásd órai példa). Az ár statikus adattag alapértelmezetten 10 értéket vegyen fel.

a. Írja meg az adattagokhoz tartozó lekérdező metódusokat!
b. Írja meg az ár beállításához tartozó beállító metódust.
c. Definiálja felül a toString() metódust úgy, hogy az ital adatait az alábbi formában adja vissza (idézőjelek nélkül): „<név>, <kiszerelés>, <ár> Ft” (például „Coca-Cola, 5 dl, 150 Ft”)!
d. Definiálja felül az equals metódust, amely akkor tekint egyenlőnek két italt, ha a nevük, kiszerelésük és az áruk is megegyezik.
e. Írjon egy statikus getÁrEuróban() metódust, amely visszaadja, hogy az ital alapértelmezetten forintban értendőt ára mennyi euró.

5. (*Algebra: másodfokú egyenletek.*) Tervezzen egy `QuadraticEquation` nevű Java osztályt egy $ax^2 + bx + c = 0$ másodfokú egyenlethez! Az osztály tartalmazzon
   - `a`, `b` és `c` privát adatmezőket, amelyek a három együtthatót reprezentálják;
   - egy konstruktort megadott `a`, `b` és `c` paraméterekkel;
   - a három `get` metódust `a`-hoz, `b`-hez és `c`-hez;
   - egy `getDiscriminant()` nevű metódust, amely visszaadja diszkriminánst, amelynek értéke $b^2 - 4ac$.
   - a `getRoot1()` és `getRoot2()` nevű metódusokat, amelyek az egyenlet két gyökét adják vissza. Ezek a metódusok csak akkor használhatók, ha a diszkrimináns értéke nem negatív. Ha a diszkrimináns negatív lenne, ezek a metódusok adjanak vissza `0` értéket!

- Írja meg a toString() metódust is!

Implementálja az osztályt!

Írjon egy Test nevű osztályt, ebbe a **main** metódust, amelyben létrehoz 3 tetszőleges `QuadraticEquation objektumot`, majd megjeleníti a diszkriminánson alapuló eredményt! Ha a diszkrimináns pozitív, írja a két gyököt! Ha a diszkrimináns `0`, írja ki a (közös) gyököt! Egyébként írja ki a „`The equation has no roots.`” üzenetet!

6. (*Algebra: 2×2-es lineáris egyenletrendszer.*)

Tervezzen egy `LinearEquation` nevű Java osztályt egy 2×2-es lineáris egyenletrendszerhez:

$$ax + by = e$$
$$cx + dy = f$$

, amiből

$$x = \frac{ed - bf}{ad - bc} \quad \text{és} \quad y = \frac{af - ec}{ad - bc}.$$

Az osztály tartalmazzon

- `a`, `b`, `c`, `d`, `e` és `f` privát adatmezőket;
- egy konstruktort megadott `a`, `b`, `c`, `d`, `e` és `f` paraméterekkel;
- hat `get` metódust (`a`-hoz, `b`-hez, `c`-hez, `d`-hez, `e`-hez és `f`-hez);
- egy `isSolvable()` nevű metódust, amely igaz értéket ad vissza, ha *ad* - *bc* nem `0`;
- `getX()` és `getY()` metódusokat, amelyek az egyenletrendszer megoldását adják vissza.

Implementálja az osztályt! Írjon egy teszt programot, amelyben létrehoz 3 különböző egyenletrendszer objektumot, majd megjeleníti mindhárom esetben a megoldást. Ha *ad* - *bc* értéke `0`, írja ki a „`The equation has no solution.`” üzenetet!

II. Rész - Statikus metódusok készítése

Hozzunk létre egy Metódusok osztályt, amely tartalmazza a „main” statikus függvényt. Egészítsük ki a következő 14 osztályszintű(STATIKUS) metódussal az osztályunkat. Ezen metódusok, az órán tanultakhoz hasonlóan STATIKUS metódusok, hogy a main-ben meghívhatók legyenek. Minden metódust hívjatok meg legalább egyszer a main-ben is, tesztelés végett:

1. Írjunk eljárást, amely paraméterként kap három egész számot. Írjuk ki őket növekvő sorrendben!
2. Írjunk eljárást, amely paraméterként kap három valós számot. Határozzuk meg és írjuk ki a három adott valós szám minimumát és abszolút értékeinek maximumát!
3. Írjunk eljárást, amely paraméterként kap négy valós számot: a, b, c, d. Írjuk ki a négy számot az adott sorrendben majd, ha d ≥ 0, az a, c, b, d sorrendben, egyébként az a, b, d, c sorrendben!
4. Adott három szigorúan pozitív valós szám: a, b, c. Írjunk függvényt, amely paraméterként megkapja ezeket a számokat és eldönti, hogy képezhetik-e ezek a számok egy háromszög oldalait (legyen a függvénynek visszatérítési értéke: boolean típusú).
5. Írjunk függvényt, amely visszaadja, hogy hány szökőév volt/lesz két különböző évszám között! (a két évszámot paraméterként adjuk át)

   Útmutatás ■ A szökőév osztható 4-gyel és nem osztható 100-zal, vagy osztható 400-zal
6. Írjunk eljárást, amely paraméterként megkap egy dolgozatra adott jegyet, és kiírja a dolgozat szöveges értékelését az érdemjegy alapján (Használjunk **switch** szerkezetet)!

7. Számítsuk ki két természetes szám egész hányadosát ismételt kivonásokkal!

```
Algoritmus Osztás(a,b,hányados):
  hányados ← 0{ bemeneti adatok: a, b, kimeneti adat: hányados }
  Amíg a ≥ b végezd el:
    hányados ← hányados + 1
    a ← a - b
  vége(amíg)
Vége(algoritmus)
```

8. Adva van egy nullától különböző természetes szám (n). Tervezzünk algoritmust, amely eldönti, hogy az adott szám prímszám-e vagy sem!

Megoldás ■ Ki kell dolgoznunk annak a módját, hogy megállapíthassuk, hogy a szám prím-e. A megoldás első változatában *a prímszám definíciójából indulunk ki:* egy szám akkor prím, ha pontosan két osztója van: 1 és maga a szám. Első ötletünk tehát az, hogy az algoritmus számolja meg az adott szám osztóit, elosztva ezt sorban minden számmal 1-től *n*-ig.

A döntésnek megfelelő üzenetet az osztók száma alapján írjuk ki.

```
Algoritmus Prím_1(n,válasz): { bemeneti adat: n, kimeneti adat: válasz }
  osztók_száma ← 0
  Minden osztó=1,n végezd el:
    Ha maradék[n/osztó] = 0 akkor
      osztók_száma ← osztók_száma + 1
    vége(ha)
  vége(minden)
  Ha osztók_száma = 2 akkor{ vagy: válasz ← osztók_száma = 2}
    válasz ← igaz
  különben
    válasz ← hamis
  vége(ha)
Vége(algoritmus)
```

További változatok ■ Vegyük észre, hogy az osztások száma fölöslegesen nagy. Ezt a számot csökkenteni lehet, mivel ha 2 és *n*/2 között nincs egyetlen osztó sem, akkor biztos, hogy nincs *n*/2 és *n* között sem, tehát eldönthető, hogy a szám prím. De, ha tovább gondolkozunk, arra is rájövünk, hogy elég a szám négyzetgyökéig keresni a lehetséges osztót, hiszen ahogy az osztó értékei nőnek a négyzetgyökig, az [*n*/*osztó*] hányados értékei csökkennek szintén a négyzetgyök értékéig. Ha egy, a négyzetgyöknél nagyobb osztóval elosztjuk az adott számot, hányadosként egy kisebb osztót kapunk, amit megtaláltunk volna előbb, ha létezett volna ilyen.

```
Algoritmus Prím_2(n,válasz): { bemeneti adat: n, kimeneti adat: válasz }
  válasz ← igaz
  Minden osztó=2,[√n] végezd el:
    Ha maradék[n/osztó] = 0 akkor
      válasz ← hamis
    vége(ha)
  vége(minden)
Vége(algoritmus)
```

9. Generáljuk és írjuk ki az első *n* Fibonacci-számot!

A Fibonacci-számokat az alábbi rekurzív összefüggéssel definiáljuk:

$F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$, ahol $i \geq 2$ (*)

Minden Fibonacci-szám tehát a megelőző kettőnek az összege (kivétel az első kettő, amelyek adottak).

A Fibonacci-számok sorozata: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

A megoldás lépéseinek leírása ■ A megoldásban a (*) összefüggést alkalmazzuk, amelyből kiderül, hogy az első *n* Fibonacci-szám generálásához elegendő három változó: *a*, *b* és *c*.

A számsor egy új tagját (*c*) úgy állíthatjuk elő, hogy a már kiszámolt elemeket egyszerűen balra „toljuk” egy pozícióval. Ily módon rendre megkapjuk *a* és *b* új értékét, majd ezek ismeretében kiszámíthatjuk *c* új értékét. A fenti lépéseket addig ismételjük, amíg a feladat által kért számú elemet elő nem állítottuk!

```
Algoritmus Fibonacci(n):
  a ← 0
  b ← 1
  Ha n = 1 akkor
    Ki: a
  különben
    Ha n = 2 akkor
      Ki: a, b
    különben
      c ← a + b
      Ki: a, b, c
      k ← 3
      Amíg k < n végezd el:
        a ← b
        b ← c
        c ← a + b
        Ki: c
        k ← k + 1
      vége(amíg)
    vége(ha)
  vége(ha)
Vége(algoritmus)
```

A fenti algoritmus generálja és kiírja egyenként a Fibonacci-sorozat első *n* elemét, amelyeket három változó segítségével állapít meg. Lássunk egy olyan algoritmust, amely mindössze két segédváltozót használ ahhoz, hogy kiszámítsa és kiírja a Fibonacci-sorozat *n* elemét.

```
Algoritmus Fibonacci(n):
  a ← 1
  b ← 0
  Minden k=1,n végezd el:
    Ki: b
    b ← a + b
    a ← b - a
  vége(minden)
Vége(algoritmus)
```

10. Adott az *n* természetes szám, amelynek legfeljebb 9 számjegye van. Hozzuk létre és írjuk ki azt a számot, amely az eredeti szám számjegyeit fordított sorrendben tartalmazza.

Elemzés ■ Legyen $n = 321$, melynek számjegyei előállításuk sorrendjében: 1, 2 és 3. Az ezekből a számjegyekből alkotott új szám 123.

Tudjuk, hogy $123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$.

Szükségünk lesz egy segédváltozóra (*újszám*), amelyben a keresett számot fogjuk generálni. Az *újszám* kezdeti értéke nulla, majd minden lépésben *újszám* új értékét a következőképpen határozzuk meg: *újszám* régi értékét szorozzuk 10-zel és hozzáadjuk az éppen előállított számjegyet.

újszám := 0
újszám := 0 · 10 + 1 = 1
újszám := 1 · 10 + 2 = 12
újszám := 12 · 10 + 3 = 123

A leírt módszer *Horner-séma* néven ismert. (Találkozunk még vele a polinom értékének kiszámításakor.)

```
Algoritmus Fordított_szám(n,újszám): { bemeneti adat: n, kimenet: újszám }
  újszám ← 0
  Amíg n ≠ 0 végezd el:
    újszám ← újszám * 10 + maradék[n/10]
    n ← [n/10]
  vége(amíg)
Vége(algoritmus)
```

11. Írjunk függvényt, amely paraméterként kap egy 0 és 12 közötti egész számot és visszaadja annak faktoriálisát! (Azért csak ekkoráét, mert a 12 faktoriálisa még tárolható egy unsigned long típusban.)
12. Írjunk eljárást, amely megtalálja és kiírja az összes k-val osztható számot, amelyek két adott szám (n1 és n2 …ezeket az eljárás paraméterei) között találhatók!
13. Írjunk függvényt, amely megkeresi azt a legkisebb Fibonacci-számot, amely nagyobb mint egy adott n szám (az n számot paraméterként adjuk át, az eredményt visszatérítési értékként)!
14. Írjunk eljárást, amely megkeresi azokat az 1000-nél kisebb számokat, amelyek egyenlők számjegyeik köbének összegével! Ezeket írjuk a standard kimenetre.