Javan GUI Scratchaajalle

Tehtävä: Kasvukausi_1

Kun puutarha nyt näyttää oikealta, sitä olisi hyvä alkaa ohjelmoida toimivaksi.

Tehdään aluksi istutukselle kuuntelija, jotta sitä klikkaamalla multaan voi asettaa kasvin kasvamaan. Istutus voi oikeastaan olla oma kuuntelijansa, joten kirjoita sille, että se toteuttaa ActionListenerin: `implements ActionListener`

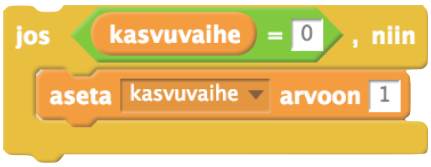
Lupauksen kirjoittaminen ei jälleen riitä vaan tee Istutus -luokalle myös seuraava metodi, tai valitse rivin alun hehkulampusta “implement all abstract methods”. Jos käytit hehkulamppua, poista `throw new` -rivin tekstit.

```
@Override
public void actionPerformed(ActionEvent ae) {

}
```

Tehtävä: Kasvukausi_2

Kirjoita tämän metodin sisään kutakuinkin:



Jos kasvuvaihe kasvaa yhdellä, aseta istutukselle myös uusi kuva setIcon -komennolla. Kuva on nyt listassa, joten `get` -komentoa käyttäen pyydä indeksissä *kasvuvaihe* olevaa kuvaa.

Testaa ja mieti miksi istutus ei vielä onnistu?

Tehtävä: Kasvukausi_3

`actionPerformed(ActionEvent ae)` -metodin ohjelmointi ei vielä tarkoita, että mikään käyttäisi sitä. Istutukselle pitää siis asettaa kuuntelijaksi se itse. Anna istutuksen konstruktorissa sille (`this`) addActionListener() -komento, jossa sulkujen välissä kuuntelijaksi asetetaan se itse (`this`).

Testaa, että istutus toimii nyt.

Bonus: Kasvukausi_4

Halutessasi voit klikkauksen jälkeen asettaa ohjelman kysymään pelaajalta haluaako hän varmasti istuttaa kasvin. Tee tällöin kasvuvaiheen kasvatuksen ja kuvan vaihtamisen ympärille toinenkin if:

```
if ( JOptionPane.showConfirmDialog(this, "Kysymys") == JOptionPane.OK_OPTION ) {

}
```

Viitteet ohjelman sisällä

Ohjelman osaset toimivat harvoin aivan itsekseen. Pelissämme haluamme seuraavaksi asettaa istuttamiselle hinnan ja antaa kasveille kasvuaikaa (yö-nappula). Yksittäisten istutus-nappuloiden ei ole missään mielessä järkevää hallinnoida omia kukkaroitaan. Samaten, jos yö-nappula vaikuttaa vain siihen itseensä, eivät kasvit voi ikinä kasvaa. Ohjelman eri osien tuleekin olla tietoisia toisistaan.

Komentoja, ja siis tietoa, toisille osille ohjelmaa voi antaa vain, jos tietää kyseisesten osien nimet.

Tarkastellaan seuraavaa kahta luokkaa:

```
1           public class Kaniyhdyskunta {
2               private ArrayList<String> repliikit;
3
4               public Kaniyhdyskunta() {
5                   repliikit = new ArrayList<String>();
6
7                   Kani rumpali = new Kani(repliikit);
8                   soitaAanite();
9               }
10
11              public void soitaAanite() {
12                  for (int i=0; i<repliikit.size(); i++) {
13                      System.out.println( repliikit.get(i) );
14                  }
15              }
16          }
```

```
1           public class Kani() {
2               public Kani(ArrayList<String> puheenvuorot) {
3                   puheenvuorot.add("Hmph.");
4               }
5
6               public void mokota() {
7
8               }
9           }
```

Nimi voi vaihtua tässä välissä. Vertaa lempinimiin. Eri hahmot voivat kutsua samaa hahmoa eri nimellä, mutta viesti menee silti perille.

Kaniyhdyskunnan rivillä 7 uutta Kani-oliota luotaessa sille annetaan mukaan viite tehtyyn listaan:

```
            new Kani(repliikit);
```

Tällöin toki Kani-oliolla pitää olla sellainen konstruktori (=rakentaja), joka ottaa vastaan listan:

```
            public Kani(ArrayList<String> puheenvuorot) {
```

Kun Kani-otuksemme on saanut viitteen repliikit-listaan, voi se antaa listalle komentoja:

```
            puheenvuorot.add("Hmph.");
```

Jos kanin olisi tärkeää myös muistaa tuo sille annettu lista, pitäisi sen myös laittaa se muistiin. Javassa viite on "voimassa" vain { } -sulkujen välisellä alueella. Esimerkiksi kanin mokota-metodissa lista ei olisi enää käytettävissä. Talteen listan saa, kun luo **oliomuuttujan**. Oliomuuttujaa ympäröivät { } -sulut ovat voimassa koodin alusta loppuun:

```
1           public class Kani() {
2               private ArrayList<String> puheenvuorot;
3
4               public Kani(ArrayList<String> puheenvuorot) {
5                   puheenvuorot.add("Hmph.");
6                   this.puheenvuorot = puheenvuorot;
7               }
…
9+          }
```

Entä sitten, jos kani haluaisi kutsua Kaniyhdyskunnan soitaAanite -metodia? Kyseinen metodi on julkinen (public), mutta yhdyskuntia voisi silti olla useita ja komennon oikeaa osoitetta ohjelma ei voi arvata ohjelmoijan ajatuksista.

Jotta kani tuntisi oikean yhteisön, voisi sen viitteen antaa sille jo konstruktoriin. Siellä viitteen voisi tallentaa oliomuuttujaan:

```
private Kaniyhdyskunta naapurusto;

public Kani(ArrayList<String> puheenvuorot, Kaniyhdyskunta naapurusto) {
    this.naapurusto = naapurusto;
}
```

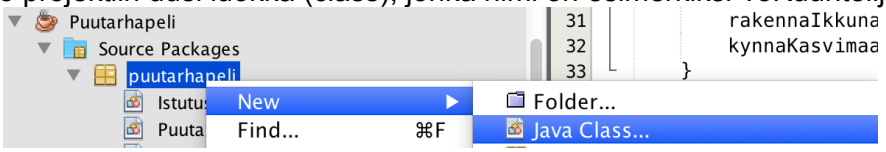
Tällöin kania luotaessa pitää toki antaa sulkujen välissä myös yhdyskunnan viite. Tässä tapauksessa olio lähettäisi viitteen itseensä eli this (=tämä):

```
Kani rumpali = new Kani(repliikit, this);
```

Tehtävä: Kasvukausi_5

`Puutarhanakyma` saa jatkossa olla pelin “aivot” ja hoitaa pelin logiikan, eli miten pelin säännöt toimivat eikä miltä se näyttää. Jotta yö-nappula voisi kuitenkin kutsua sitä, kun sen klikkaus tapahtuu, pitää sillä ensin olla kuuntelija.

Luo projektiin uusi luokka (class), jonka nimi on esimerkiksi YoKuuntelija.



Kirjoita YoKuuntelijan määritysriville (public class), että se toteuttaa (implements) ActionListenerin. Luo sille myös tämän lupauksen myötä seuraava komento:

```
public void actionPerformed(ActionEvent ae) {

}
```

Tehtävä: Kasvukausi_6

Klikkauksen tapahduttua YoKuuntelijan pitää ottaa yhteyttä pelin logiikkaan. Sen pitää siis myös tuntea se nimeltä. Luo YoKuuntelija -luokalle konstruktori, joka ottaa vastaan Puutarhanakyman:

```
public YoKuuntelija(Puutarhanakyma logiikka) {

}
```

Tee **YoKuuntelija** -luokalle myös oma private Puutarhanakyma-tyyppinen oliomuuttuja ja tallenna siihen konstruktorissa annettu Puutarhanakyma-viite. Katso mallia yltä, jossa Kaniyhdyskunta -tyyppinen viite tallennetaan muistiin.

Tehtävä: Kasvukausi_7

Puutarhanakyma -luokassa lisää yönappulalle (se, jossa lukee “Nuku yö.” tai vastaavaa) juuri luotu kuuntelija `addActionListener()` -komennolla. Kirjoita komennon sulkujen väliin uusi (new) YoKuuntelija ja anna sille `this` -viite.

Tehtävä: Kasvukausi_8

Klikkauksesta ei vieläkään tapahdu mitään. Tee **Puutarhanakyma** -luokkaan uusi metodi, jota kuuntelija voi kutsua:

```
public void nukuYo() {

}
```

Lisää **YoKuuntelija** -luokan actionPerformed -metodiin tämän metodin kutsu. Kenelle seuraava metodikutsu pitää antaa? Vinkki: viitteen komennettavaan pitäisi löytyä YoKuuntelija -luokan oliomuuttujista ja mikä luokka ymmärtää nukuYo-komennon?

```
komennettava.nukuYo();
```

Tehtävä: Kasvukausi_9

Pelissä istutukset kasvavat yön aikana. Luo **Istutus** -luokalle metodi, jota voimme kohta kutsua. Listan pituuden voit pyytää `listanNimi.size()` -komennolla. Javassa “ja” kirjoitettiin: &&

Kun kasvuvaihetta on **kasvatettu** yhdellä, aseta istutuksen kuvaksi kasvuvaihetta vastaava kuva. Ota mallia Istutus -luokan actionPerformed -metodista.

Pohdi: Miksi `pituus - 1` ? Kuvitellaan, että vaiheita on kolme, eli kasvuvaihe saa arvot 0-2. Listan pituus on toisaalta 3, joten pituus-1 on 2. Kasvi ei kuitenkaan saa enää kasvaa, kun se on vaiheeltaan 2. Jotta kasvua voi vielä tapahtua pitää kasvuvaiheen olla < 2.

Listojen läpikäyminen

Listoja käytettäessä halutaan usein pyytää vuorotellen jokainen listan alkio syyniin. Voisimme tehdä perinteisen for-silmukan, jos kysyisimme listalta ensin sen koon (=size):

```
ArrayList<Integer> lista = new ArrayList<Integer>();
int koko = lista.size();
```

Listoja voi kuitenkin läpikäydä vielä yksinkertaisemmin for-each syntaksilla:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
//tässä välissä listaan voi lisätä alkioita

for (Integer alkio : lista) {
    System.out.print(alkio + ", ");
}
```

Tarkastellaan neliöityä aluetta. Kaksoispisteen jälkeen on määritetty minkä nimisestä listasta alkioita otetaan. Aluksi on toisaalta määritetty, että listasta saadaan käsiteltäväksi Integer -tyyppisiä alkioita, joista { } -sulkujen sisällä käytetään kustakin vuorollaan alkio -nimeä.

Tehtävä: Kasvukausi_9

Luo **Puutarhanakyma** -luokan nukuYo -metodiin for-silmukka, joka käy kaikki istutukset läpi. Kunkin istutuksen kohdalla kutsu sen kasva -metodia.

Huomaa for-silmukassa, että listalla ei ole edellisen esimerkin kaltaisesti Integer-olioita vaan Istutus-olioita. Niitä ei tarvitse myöskään kutsua nimellä “alkio” vaan voit päättää sen itse.

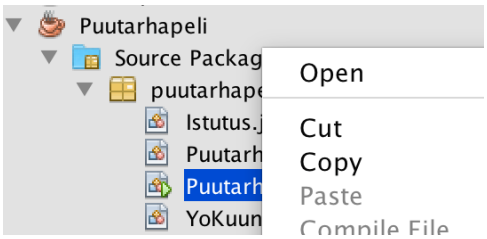
Nyt voit vihdoin testata, että pelissä kasvit kasvavat yön aikana.

Tehtävä: Kasvukausi_10

On hieman tyhmää, että Puutarhanakyma -niminen olio hoitaa pelin logiikkaa. Nimensä mukaisesti sen tulisi hoitaa ulkonäköä. Tyypillisesti grafiikka ja logiikka eroitellaan ohjelmissa eri luokille, mutta pienessä pelissämme ne voivat nyt olla yhdessä. Annetaan luokalle kuitenkin nyt vähän kuvaavampi nimi.

Voit valita nimen itse, mutta esimerkiksi seuraava toimii. Uudelleennimeä luokka klikkaamalla sitä oikealla painikkeella ja valitse refactor > rename.

Puutarhanakyma -> **Puutarha**



Tehtävä: Rahakirstu_1

`Puutarha` hallinnoi pelaajan varoja. Luo sille oliomuuttuja (tyypillinen private-rivimme), johon voi säilöä kultaa. Nimi voi olla vaikka “kulta” ja tyypiksi voit valita kokonaisluvun (int) tai desimaaliluvun (double).

Anna pelaajalle jokin määrä kultaa heti aluksi. Määrän voi määrittää oliomuuttujan määritysrivillä tai konstruktorissa.

Tehtävä: Rahakirstu_2

Kun pelaaja istuttelee uusia kasveja, tulisi istutuksen aina tietää onko kultaa tarpeeksi. Tai oikeammin, se on pelin “aivojen” tehtävä. Istutuksen pitääkin tuntea Puutarha, jotta se voi esittää sille kysymyksen varallisuudesta.

1) Korjaa **istutuksen** konstruktoria (=rakentaja eli public Istutus() {…} -rivit), niin että se ottaa vastaan Puutarha -viitteen:

```
public Istutus(Puutarha pelilogiikka) {

}
```

2) Tee **Istutus**-luokkaan myös oliomuuttuja, johon voit tallentaa tämän viitteen jatkokäyttöä varten. Katso mallia yltä, jossa Kaniyhdyskunta -tyyppinen viite tallennetaan muistiin.

3) Korjaa **Puutarha** -luokkaa niin, että aina istutusta luodessa sille annetaan myös viite itse puutarhanakyma-olioon, eli `this` -olioon.

Totuusarvot

Javassa on numeroita (int ja double), tekstimuuttujia (String) ja mitä ihmeellisempiä valmiita ja keksittyjä olioita (mm. JButton, ArrayList ja Kissa). Käytettävissä on kuitenkin vielä yksi hyvin yksinkertainen tieto; jokin voi olla totta tai valetta.

Javan totuusmuuttuja on “boolean”. Sillä voi olla arvo true (totta) tai false (epätotta).

```
boolean olenWanda = false;
if (olenWanda) {
    System.out.println(“Olen kala nimeltä Wanda.”);
}
```

Palautusarvot

Scratchissa viestien lähetykset ja lohkot olivat vain komentoja; niiden johdosta ohjelma suoritti jotain komentojasarjoja, mutta alkuperäinen viestin lähettäjä tai lohkon käyttäjä ei saanut ikinä mitään vastausta.

Javassa kaikki komennot voivat saada myös vastauksen, jos niin määritetään.

Toistaiseksi olemme kirjoittaneet usein metodien määritykseen, että ne palauttavat “void”. Eli ne palauttavat vain “tyhjää”. Voisimme myös pyytää vastaukseksi jonkin arvon, numeron tai totuusarvon ..tai jotain ihan muuta. Vastauksen palauttaminen toimii return (=palauttaa) -komennolla:

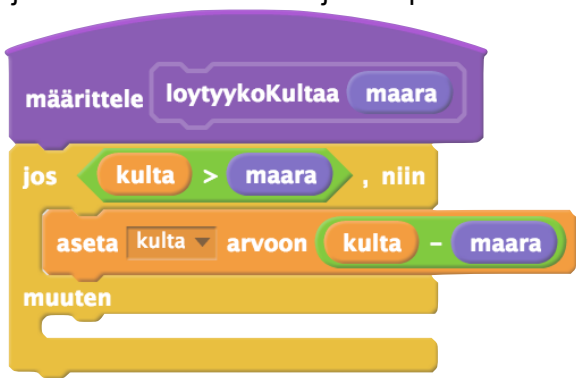
```
1   public static void main (String[] args) {
2       if (paistaakoAurinko()) {
3           System.out.println(“Aurinkorasvaa iholle!”);
4       }
5   }
6
7   public static boolean paistaakoAurinko() {
8       //ohjelma toimii pohjoisnavalla kesällä
9       return true;
10  }
```

Käytännössä, kun ohjelma pääsee riville 2, se huomaa tuntemattoman arvon “paistaakoAurinko()”. Onnekseen se löytää myös, miten ohjelmoija on määrittänyt tälle arvon riveillä 7-10. Ohjelman suoritus siirtyy sinne. Kun siellä sitten ohjelma ensi kertaa törmää return -komentoon, lähtee vastaus takaisin riville 2. Tässä vaiheessa vastaus sijoitetaan: `if (true) {` eli “jos tosi on totta”. Tosi on aina totta, joten ohjelma tulostaa rivin 3 tekstin.

> Tehtävä: Rahakirstu_3
>
> Jotta istutus voisi varallisuuskysymyksiä puutarhanakymalle esittää, on **Puutarha**-luokkaan vielä kirjoitettava jokin komento, jota kutsua. Tämä komento voi palauttaa totuusarvon “totta, on varaa” tai “epätotta, ei ole varaa”. Tee seuraavanlailla määritetty metodi:
>
> ```
> public boolean loytyykoKultaa (int maara) {
>
> }
> ```

Tehtävä: Rahakirstu_4

Kirjoita metodiin if-else -ohjelmanpätkä.



Jos kultaa on tarpeeksi, kirjoita kullan vähentämisen jälkeen: `return true;` muuten kirjoita: `return false;`

Jos haluat olla tarkempi rahan suhteen, kirjoita >= vertailu.

Tehtävä: Rahakirstu_5

Istutuksen tulisi nyt kysyä rahatilannetta ennen uuden kasvin istuttamista. Korjaa Istutus -luokan actionPerformed -metodia niin, että kasvuvaiheen ja kuvan muuttamisen ympärillä on vielä yksi if (sisempänä kuin `if (kasvuvaihe == 0)`). Tässä if-testissä kysy puutarhalta, onko istutukseen varaa.

Ota mallia edellisen sivun paistaakoAurinko-tapauksesta.

Tämän korjauksen jälkeen istutuksia ei pitäisi voida tehdä enempää kuin mihin on varaa, mutta rahatilanne ei vielä päivity ikkunassa.

Tehtävä: Rahakirstu_6

Puutarha -luokan rakennaNakyma -metodissa luotiin aiemmin JLabel, jossa näkyi kullan määrä. Tee luokkaan private oliomuuttuja, johon voi tallentaa viitteen tuohon JLabeliin.

Täydennä saman luokan loytyykoKultaa -metodiin tuon JLabelin tekstin päivitys. Jos kultaa käytettiin, kutsu `setText("Uusi teksti")` -käskyä. Yhdistä + -merkillä yhteen järkevä lause:

komennettava. 

Bonus: Rahakirstu_7

Jos teit aiemmin kulta -muuttujasta desimaaliluvun (double), ikkunan tekstissä näkyy nyt ostosten jälkeen esim. 8.0

Voit suorittaa matemaattisia toimenpiteitä staattisen Math -luokan avulla. setText -metodikutsuun voit kulta -muuttujan tilalle kirjoittaa:

```
Math.round(kulta);
```

Bonus: Rahakirstu_8

Jos teit Kasvukausi_4 bonustehtävän ja olet jo kyllästynyt sen jatkuvasti antamiin kyselyihin, voimme rajoittaa vielä ohjelmaa niin, että se kysyy vahvistusta vain ensimmäisellä kerralla.

Tee ensin **Puutarha** -luokkaan private oliomuuttuja, johon voi tallentaa totuusarvon (boolean), ja jonka nimi on vaikka `onkoJoIstutettu`. Anna sille alussa arvo `false`.

Bonus: Rahakirstu_9

Tee Puutarha -luokkaan uusi metodi:

```
public boolean onkoJoIstutettu() {
```


```
}
```

Totuusarvojen oikoreittejä

Edellisessä tehtävässä ei oikeastaan tarvitsisi testata onko muuttujan arvo true. Käytännössä “jos” tarvitsee vain tiedon “totta, suoritetaan tämä” tai “epätotta, ei suoriteta tätä”. Sille voi siis antaa ihan vain boolean-muuttujan alun alkaenkin:

```
boolean paistaakoAurinko = false;
if (paistaakoAurinko) {
    System.out.println(“Olet varjossa.”);
}
```

Toisaalta tehtävässä palautetaan aina päinvastainen totuusarvo kuin mikä meille on tallennettu. Onko jo istutettu on epätotta silloin kun istutus on todella eka istutus, ja päinvastoin. Totuuden kääntämiseen löytyy “looginen laskutoimitus” eli huutomerkki:

```
public void vaihdaTotuutta(boolean totuus) {
    return !totuus;
}
```

Bonus: Rahakirstu_10

Siirry **Istutus** -luokkaan ja etsi **actionPerformed** -metodi. Muokkaa nyt `if (JOptionPane…` -riviä. Tavoite olisi, että testi päästää ohjelman läpi istuttamiseen, jos istutuksia on jo *tai* JOptionPaneen vastaa “kyllä”. Muokkaa siis if -rivistä seuraavanlainen:

```
if (pelilogiikka.onkoJoIstutettu() || JOptionPane.show…) {
```

Kirjoita `pelilogiikka` -viitteen tilalle oman Puutarha-viitteesi nimi.

Ohjelman pitäisi nyt kysyä vahvistusta vain, kun aivan ensimmäinen istutus tehdään.