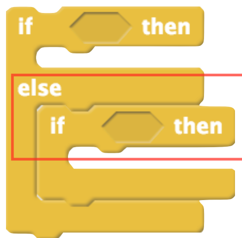
Javan GUI Scratchaajalle

Tehtävä: Sadonkorjuu_1

Kypsäksi kypsyneistä kasveista voisi saada myös rahaa, jos ne kerää pois ja myy.

Kerääminen tapahtuu klikkaamalla, joten tehdään uutta ohjelmaa Istutus -luokan actionPerformed -metodiin. Aiemmin siellä testattiin, jos kasvuvaihe == 0. Tee sen jälkeen else if, jossa testataan onko kasvuvaihe kuvalistan pituus - 1. Listan pituuden sait listanNimi.size() -komennolla.



`else if` eli "muuten jos" tarvitaan, sillä emme halua, että yhdestä klikkauksesta voi tapahtua kaksi asiaa peräkkäin. Teoriassa ensimmäinen if voisi kasvattaa kasvin valmiiksi, ja tällöin seuraavan rivin if voisi jo poimia sen.

```
if (kasvuvaihe == 0) {
    //tehtiin jotain
}
else if (hedelmaOnKypsa) {

}
```

Kirjoita tämän tilalle siis kasvuvaiheen vertailu.

Tehtävä: Sadonkorjuu_2

else if -tapauksessa (eli { } -sulkujen välissä) aseta kasvuvaihe takaisin arvoon 0 ja aseta kuvaksi oikea ikoni.

Tehtävä: Sadonkorjuu_3

Istutus tarkkailee hedelmän kypsymistä, mutta Puutarha huolehti rahakirstusta. Tee **Puutarha** -luokalle seuraava metodi:

```
public void ansaitseKultaa(int maara) {
```

```
}
```

setText -komento piti antaa graafisen ikkunan JLabel-etiketille, joka näyttää kullan määrän. Voit katsoa mallia **Puutarha** -luokkaan aiemmin ohjelmoidusta **loytyykoKultaa** -metodista.

Tehtävä: Sadonkorjuu_4

Kultaa ei tietenkään vielä ansaita vaikka äsken siihen ohjelmoitiinkin ohje. Komentoja pitää kutsua, että ne tapahtuvat.

Kirjoita **Istutus** -luokan **actionPerformed** -metodin else if -tapaukseen, että kasvuvaiheen alustamisen ja kuvan asetuksen lisäksi puutarhalle annetaan ansaitseKultaa(4) -komento. Saat toki itse päättää ansaitun kullan määrän. Muista, että kun komento annetaan jollekin luokan ulkopuoliselle hahmolle, tulee komennon eteen kirjoittaa komennettavan nimi ja sen perään piste.

Testaa, että pelissä saa nyt kultaa poimituista kasveista.

Lisenssi: CC BY-NC-SA

Tiedostot

Poikkeuskäsittely

Ennen kuin pääsemme edes tutustumaan tiedostoihin pitää meidän opetella vähän poikkeuskäsittelyä, sillä voihan aina olla, että pyydettyä tiedostoa ei edes ole olemassa. Tai sitten tiedoston osoite kirjoitettiin ohjelmalle vähän väärin.

Huomio: Javassa on *kirjoitusvirheitä*, *poikkeuksia* ja *virheitä*. **Kirjoitusvirheellistä** ohjelmaa ei voi ajaa, tai edes kääntää. **Poikkeus** (=exception) on toisaalta ohjelman suorituksen aikana syntyvä virhe, tai poikkeama siitä mitä ohjelmoija oli ajatellut ohjelman ulkopuolelta löytyvän. Poikkeuksen alkuperä voi olla esimerkiksi käyttäjä tai tiedosto. **Virhe** (=error) on täysin ennalta-arvaamaton häiriö ohjelman toiminnassa. Puhumme nyt poikkeuskäsittelystä, sillä virheisiin ei tyypillisesti ole syytä varautua.

Kun ohjelmassa tiedetään, että jostakin voi syntyä poikkeus, pitää kyseinen koodi laittaa “kokeilun kohteeksi” eli siis yritettäväksi (=try). Jos jokin virhe sitten toisaalta syntyisi, pitää sille olla oma “koppialue” eli ohje ohjelmalle, että mitä tehdään, jos virhe syntyi. (catch = napata, ottaa koppi)

```
try {

} catch (Exception e) {

}
```

Yleispätevin poikkeus, joka voidaan napata on juurikin Exception. Käytännössä tämä on “jos mitään meni juuri äsken pieleen, tee tämä …”.

Bonus: Napata voidaan myös esimerkiksi FileNotFoundException -poikkeuksen, eli tiedoston puuttumisen poikkeuksen, mutta sekin on perimmiltään Exception. Jos ohjelmassa halutaan varautua erilaisiin mahdollisesti syntyviin poikkeuksiin eri tavoin, voi catch {} -osioita kirjoittaa try {} -osion perään useampiakin.

Bonus: Kilinää_1

Ohjelmaan lisätään kohta ääni rahan keruuseen. Jos äänitiedostoa ei löydy, syntyy siitä poikkeus. Kirjoita Puutarha -luokalle uusi metodi:

```
private void lataaAani() {

}
```

Kirjoita metodin { } -sulkujen väliin jo try-catch -koodi seuraavaa tehtävää varten.

Kutsu lataaAani -metodia **run** -metodin lopuksi kasvimaan kyntämisen jälkeen.

Äänitiedostot

Äänien luomiseen tarvitaan useita olioita. Tarvitaan alkuperäinen äänitiedosto (`File`), äänilaitteisto (`AudioSystem`), äänen toistaja (`AudioInputStream`) sekä kasetti, jolle nauhoitetaan (`Clip`). Todennäköisesti teet jatkossa ääniä sen verran harvoin, että kopioit tarvittavan koodin aina vanhemmasta projektistasi tai netistä.

Alkuun tulee hakea käyttöön äänitiedosto:
```
File aanitiedosto = new File(“meow.wav”);
```

Sitten “valitaan” kasetti, jolle nauhoitetaan:
```
Clip kasetti = AudioSystem.getClip();
```

Tämän jälkeen äänitiedosto voidaan lisätä toistettavaksi:

```
AudioInputStream toistosyote = AudioSystem.getAudioInputStream(aanitiedosto);
```

Soittoon valmistavista työvaiheista viimeisenä nauhoite avataan kasetille:

```
kasetti.open(toistosyote);
```

Kun ääni on kasetilla voidaan se myöhemmin milloin vain soittaa. Kasetti pitää vain joka soiton aluksi kelata takaisin alkuun:

```
kasetti.setFramePosition(0);
kasetti.start();
```

set = aseta
frame = kehys
position = sijainti
start = aloita

Huom! Äänitiedostoa avatessa oletetaan, että pyydetty tiedosto myös löytyy. Näin ei aina ole, joten koko prosessi `File` -rivistä `kasetti.open` -riviin pitää sijoittaa try { } -sulkujen väliin.

Bonus: Kilinää_2

Luo esimerkiksi Scratchissa äänitiedosto rahan keruuseen liitettäväksi. Ääniä voi etsiä Äänet-välilehdellä ja tiedoston voi tallentaa, kun ääntä klikkaa hiiren oikealla painikkeella. Jos Scratchissa ei toimi oikeaklikkaus, paina pohjaan Shift-näppäin ja paina hiiren vasemmalla painikkeella.

Tallenna äänitiedosto projektin kansioon. Se löytyy tyypillisesti NetBeansProjects -kansiosta.

Kirjoita koko äänen luontiprosessi `File` -rivistä `kasetti.open` -riviin lataaAani -metodin try { } -sulkujen väliin. Lainaa nimenomaan seuraavat ohjeet:

```
import java.io.File;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

Poikkeusten jäljitys

catch { } -osioon voidaan ohjelmoida poikkeuskäsittely, mutta vähimmillään siellä kannattaa tulostaa poikkeuksen tapahtuminen ohjelmoijalle/käyttäjälle. Kun poikkeus on catch-osioon saapunut esimerkiksi nimellä e voidaan sille antaa komento tulostaa tietonsa:

```
e.printStackTrace();
```

Tämä voi johtaa esimerkiksi seuraavanlaiseen tulosteeseen:

```
run:
java.io.FileNotFoundException: me2ow.wav (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:138)
        at com.sun.media.sound.WaveFloatFileReader.getAudioInputStream(WaveFloatFileReader.java:164)
        at javax.sound.sampled.AudioSystem.getAudioInputStream(AudioSystem.java:1181)
        at puutarhapeli.Puutarha.lataaAani(Puutarha.java:51)
        at puutarhapeli.Puutarha.run(Puutarha.java:44)
        at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:311)
        at java.awt.EventQueue.dispatchEventImpl(EventQueue.java:749)
```

Näitä tulosteita luetaan **ylhäältä alas** kunnes ongelmakohta löydetään. Ylimmällä (punaisella) rivillä heti alussa kerrotaan, että tapahtui `FileNotFoundException`, eli tiedostoa ei löytynyt. Poikkeuksen nimi on usein se tärkein vinkki.

Seuraava huomattava vinkki on aina ylin punainen rivi, jolla on oman projektin koodia. Esimerkin tapauksessa `at puutarhapeli.Puutarha.lataaAani(Puutarha.java:51)`. Tämä paljastaa missä tiedostossamme ja millä rivillä poikkeus lopulta väistämättä syntyi.

Bonus: Kilinää_3

Kirjoita lataaAani -metodin catch { } -osioon äsken esitelty poikkeuksen tulostus.

Suorita ohjelma ja varmista, että poikkeusta ei tapahdu, eikä siis myöskään tulostu, heti projektin auettua. Jos poikkeus syntyy, varmista, että pyydetty tiedostonimi on oikein ja tiedosto on oikeassa kansiossa.

Bonus: Kilinää_4

Tällä hetkellä kasettinauhoite on ohjelman muistissa vain Clip-oliota ympäröivien { } -sulkujen välissä. Luo siis luokan alkuun private Clip-tyyppinen oliomuuttuja, jonka nimi on sama kuin lataaAani -metodissa luomasi Clip-olion.

Jotta lataaAani -metodi käyttäisi tätä uutta oliomuuttujaa, poista metodin `Clip kasetti =` -rivin alusta Clip-sana, jotta java ei määrittele metodiin uutta erillistä nauhoitetta.

Bonus: Kilinää_5

Kirjoita Puutarha -luokan ansaitseKultaa -metodiin komento kelata nauhoite alkuun ja sitten soittaa se. Katso komennot yltä.

Testaa nyt ja varmista, että ääni soitetaan aina (ei vain ensimmäisellä kerralla), kun pelaaja kerää työnsä hedelmiä.

Staattiset metodit

Olemme aiemmin jo käyttäneet staattisia luokkia, joista ei tarvitse luoda uusia (=new) olioita ennen käyttöä. Esimerkiksi SwingUtilities, jota käytämme ikkunan automaattikäynnistykseen, on tällainen "lähistöllä lymyilevä luokka".

Staattisia metodeja voi tehdä myös itse. Riittää, että metodin määrittelyyn lisää avainsanan static. Muuten luokka voi näyttää ihan normaalilta, vaikkakin jos siitä ei ikinä luoda uusia olioita voi sen konstruktori olla private.

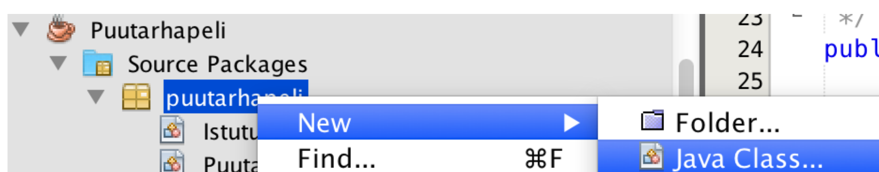
```
public class RuokaAutomaatti {
    private RuokaAutomaatti() {}

    public static String kerroMerkki() {
        return "Linkki";
    }
}
```

Staattisia metodeja voi olla järkevä tehdä silloin, kun ohjelmalle halutaan kirjoittaa hieman yleispätevämpiä metodeja ja metodin toimintaa ei tarvitse sitoa ohjelman toimintaan.

Tehtävä: Sadonkorjuu_5

Luo projektiin uusi javaluokka, joka on nimeltään vaikka Talvivarasto.



Tehtävä: Sadonkorjuu_6

Luo **Talvivarasto** -luokkaan staattinen ***tallennaPuutarha*** -metodi. Sen ei tarvitse palauttaa välttämättä mitään, joten edelliseen esimerkkiin verrattuna kirjoitetaan void (=tyhjyys) eikä String palautettavaksi arvoksi. { } -sulkujen väli voi vielä jäädä tyhjäksi.

Tehtävä: Sadonkorjuu_7

Peli voisi tallentaa itsensä aina, kun puutarhassa tapahtuu jotain, mutta se olisi melkoista suoritustehon haaskausta. Onkin parempi, että lisäämme ikkunalle kuuntelijan, joka tietää milloin ikkuna suljetaan. Kuuntelija voi tällöin käynnistää talvivarastoon tallennuksen.

Aivan aluksi ikkunaa pitää korjata niin, että sen sulkeminen ei pysäytä ohjelman suoritusta vaan vain sulkee ikkunan. Korjaa siis `EXIT_ON_CLOSE` tilalle `DISPOSE_ON_CLOSE`.

Huom! Tällä ohjeella ikkunan sulkeminen raksista johtaa tallennukseen, ohjelman pysäyttäminen muulla tavoin ei mahdollista tallentamista ennen sulkeutumista näin.

Tehtävä: Sadonkorjuu_8

Ikkunalle pitää nyt tehdä kuuntelija, joka tietää milloin ikkuna sulkeutuu. Tähän sopii `WindowListener`. **Puutarha** toteuttaa jo Runnable -toiminnallisuuden, mutta se voi ola myös WindowListener. Kirjoita Puutarha -luokkaan `implements Runnable` perään `WindowListener` pilkulla Runnable:sta erotettuna.

WindowListener lupaa jo 7 metodia ja NetBeans osaa lisätä ne meille automaattisesti. Valitse vasemman laidan hehkulampusta “add import” ja sitten “implement all abstract methods”.

Poista tiedoston loppuun syntyneistä metodeista `throw new …` -rivit.

Tehtävä: Sadonkorjuu_9

Yhtä tärkeää kuin kuuntelijan tekeminen on myös kuuntelijan lisääminen kohteelle. Kirjoita **Puutarha** -luokan ***run*** -metodiin, että luodulle JFrame:lle lisätään myös ikkunakuuntelija.

Ohje: Tarkista minkä nimisen JFramen itse loit ja anna sille `addWindowListener()` komento. Komennon sulkujen sisälle kirjoita `this`, eli lisättävä WindowListener on juurikin “tämä” Puutarha-luokka. Komennon tarkalla sijainnilla run -metodissa ei ole väliä, kunhan se annetaan uuden JFramen luonnin jälkeen.

Tehtävä: Sadonkorjuu_10

Lisää Puutarhan ***windowClosing*** -metodiin kutsu tallentaa peli. tallennaPuutarha -metodimme oli staattinen, joten metodin kutsu onnistuu näin:

```
Talvivarasto.tallennaPuutarha();
```

Tallentamiseen kylläkin tarvitaan puutarhan tietoja. Kirjoita kutsuun () -sulkujen väliin kultamäärä-viitteen nimi ja istutuslista-viitteen nimi (tarkista tiedoston alusta) pilkulla erotettuina.

Lopputulos on punainen virheelliseksi määritetty rivi, sillä Talvivaraston tallennaPuutarha ei osaa vielä ottaa noita tietoja vastaan. Korjaa **Talvivaraston *tallennaPuutarha*** -metodin () -sulkujen väliin, että se vastaanottaa arvot `int kulta` ja `ArrayList<Istutus> istutuslista` pilkulla erotettuina. Tai jos kullan määräsi oli double, niin toki `double kulta`.

Tiedostojen tallennus

Javassa oliot hoitavat myös tiedostojen tallennuksen. Tallennuksen osaavia olioita on useammanlaisia eri tarkoituksiin, mutta tutustumme nyt `FileWriter` -olioon. (file=tiedosto, writer=kirjoittaja)

FileWriter:n ohjeet voi ladata näin:

```
import java.io.FileWriter;
```

FileWriter:lle pitää sitä avatessa kertoa mihin tallennetaan. Tämän jälkeen valittuun tiedostoon voidaan liittää (=append), eli siis “liimata loppuun”, tekstiä. Tekstinkirjoittaja on fiksua aina lopuksi sulkea (=close). Huom! Tiedostojen käsittely voi aina luoda poikkeuksia, joten koko prosessi pitää laittaa kokeiltavaksi (=try).

```
try {
    FileWriter kirjuri = new FileWriter( “tiedostonimi.txt” );

    kirjuri.append(“tekstiä”);
    kirjuri.append(“lisää tekstiä”);

    kirjuri.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Bonus: Tiedoston kirjoitus tuhoaa aina automaattisesti tiedoston aiemmin sisältämät tekstit. Jos tekstiä haluaakin lisätä jo olemassa olevan tiedoston loppuun pitää FileWriter:a luotaessa myös kertoa, että on totta (=true), että lisätään loppuun:

```
    FileWriter kirjuri = new FileWriter( “tiedostonimi.txt”, true );
```

> Tehtävä: Sadonkorjuu_11
>
> Luo **Talvivaraston** tallennaPuutarha -metodissa FileWriter ja sulje se try {} -sulkujen välissä. Luo myös edellisen esimerkin kaltainen catch {} -poikkeuskäsittely.
>
> Voit itse valita, minkä nimiseen tiedostoon talvivarasto tallennetaan.

Tiedostot pitää kirjoittaa systemaattisesti niin, että ohjelma seuraavan kerran auetessaan myös ymmärtää mitä tiedostossa lukee. Ohjelma voi esimerkiksi lukea rivi kerrallaan tietoja. Kunkin rivin sisältämät tiedot voivat kuvata esimerkiksi jotain yhtä tapahtumaa tai esinettä. Rivin sisällä tiedot on voitu erotella mm. välilyönnein. Tietojen tulee tulla joka rivillä samassa järjestyksessä. Ohjelma ymmärtää poikkeuksia vain jos ne sille ohjelmoidaan myös lukupuolelle.

> Tehtävä: Sadonkorjuu_12
>
> Tiedoston ensimmäisellä rivillä voi lukea kullan määrä. FileWriterin luonnin jälkeen liitä tiedostoon append -komentoa käyttäen kullan määrä ja lisää (`+`) siihen rivinvaihto `“\n”`.
>
> **Testaa**, että tiedosto syntyy, kun testatessa ensimmäinen hedelmä kerätään.
>
> Bonus: Rivinvaihdon n tulee sanasta “new” eli uusi. Kenoviiva \ on erikoismerkeille varattu tuntomerkki, sen jälkeisen merkin ohjelma tulkitsee erilaisena kuin miltä se näyttää; n johtaa rivinvaihtoon, t sisennykseen (tabulator). Kenoviivalla ohjelmaan voi lisätä myös Unicode-merkkejä, joita normaali aakkosto ei sisällä.

Tehtävä: Sadonkorjuu_13

Kullan määrän liittämisen jälkeen tallennetaan kunkin istutuksen vaihe omalle rivilleen.

Jotta istutuksen vaihe voitaisiin kirjoittaa, tulee istutuksen pystyä kertomaan se. Kirjoita **Istutus** -luokalle seuraavanlainen metodi, joka palauttaa (return) kasvuvaiheeseen talletetun numeron.

```
public int annaKasvuvaihe() {

}
```

Tehtävä: Sadonkorjuu_14

Talvivarasto -luokassa tallennaPuutarha -metodissa luo kullan tallentamisen jälkeisillä riveillä istutuslistan läpikäyminen:

```
for (Istutus kasvi : istutuslista) {

}
```

{} -sulkujen sisällä liitä (append) tiedostoon kulloinkin vuorossa olevan kasvin kasvuvaihe sekä rivinvaihto eli “\n”. Kasvuvaiheen saa pyydettyä normaalilla metodikutsulla, eli komennettavan nimi annetaan ennen pistettä ja komento sen jälkeen:

```
kasvi.annaKasvuvaihe()
```

Testaa! että pelin suljettaessa syntyvä tiedosto sisältää nyt kullan määrän ja kasvuvaiheet, jokaisen omalla rivillään. Jätä tiedosto seuraavaa vaihetta varten tallelle.

Tiedostojen lukeminen

Tiedostojen lukemiseen voi käyttää samaa työkalua kuin käyttäjän syötteen lukemiseen. Scanner toimii kumpaankin. Tiedostoa lukiessa luettavaksi pitää vain antaa tiedosto ja poikkeusten ilmaantumiseen pitää varautua.

```
try {
    File tiedosto = new File(“tiedostonimi.txt”);
    Scanner lukija = new Scanner(tiedosto);
} catch (Exception e) {
    e.printStackTrace();
}
```

Edellinen ohjelma vasta avaa tiedoston skannattavaksi. Scanner osaa mm. komennot `hasNextLine()` ja `nextLine()`, eli kysymyksen, onko tiedostossa vielä seuraavaa riviä ja pyynnön lukea seuraava rivi. Rivejä voimme lukea niin kauan kuin (while) rivejä riittää:

```
while (lukija.hasNextLine()) {
    lukija.nextLine();
}
```

Tehtävä: Istutuskausi_1

Luo **Talvivarasto** -luokkaan uusi **avaaVarasto** -metodi. Myös sen tulee olla julkinen (public) ja staattinen, mutta se ei palauta tyhjää (void) vaan tiedoston rivit listana eli `ArrayList<String>`. Merkitse metodin () -sulkujen väliin, että metodi saa *parametriksi* kokonaisluvun nimeltä `istutustenMaara`.

Metodin {} -sulkujen sisällä luo ArrayList:

```
ArrayList<String> rivit = new ArrayList<String>();
```

Luonnin jälkeen palauta (`return`) metodista tämä `rivit` -lista.

Tehtävä: Istutuskausi_2

Listan luomisen ja sen palautuksen välissä luo Scanner -olio ja lue tiedostosta sillä kaikki rivit edellä annetuin ohjein. Avaa samanniminen tiedosto kuin äsken tallensit. Älä unohda try-catch -osiota.

Listalle rivin voi aina lisätä tyyliin: listannimi`.add(`lisättävä asia`)`. Täydennä listannimi ja lisättävä asia (eli tiedoston rivi), jotta luetut rivit myös päätyvät listalle.

Tehtävä: Istutuskausi_3

Jotta peli voisi avata tallenteen tulee sen siis tietää miten monta istutuksia pyydetään varastosta. **Puutarha** -luokassa määrä riippui ikkunaan asettelusta. Laitetaan nyt määrä Puutarha -luokankin muistiin sen oliomuuttujiin tiedoston alkuun:

```
private int riveja = 3;
private int sarakkeita = 4;
```

Päätä itse kuinka monta riviä ja saraketta puutarhaasi halusit.

Käy korjaamassa Puutarhan **kynnaKasvimaa** -metodia, siellä on `rivi<3` ja `sarake<4` kohdat. Kirjoita **numeroiden** tilalle muuttujat `riveja` ja `sarakkeita`.

Tehtävä: Istutuskausi_4

Lisää Puutarhan **run** -metodin alkuun, ennen JFramen luomista, talvivaraston avauskomento ja tallenna sen palauttama lista talvivarasto -muuttujaan:

```
ArrayList<String> talvivarasto = Talvivarasto.avaaVarasto(4);
```

Korjaa numeron neljä tilalle omien istutuksiesi määrä. Rivien ja sarakkeiden määrä on nyt tallessa muuttujissa, joten älä kirjoita numeroa vaan rivien ja sarakkeiden kertolasku.

Ohjelman pitäisi nyt toimia, vaikka varastolla ei mitään tehdäkään lukemisen jälkeen.

Huom! Tästä eteenpäin ohjelma ei toimi hetkeen, jos tallennetta ei löydy.

Tehtävä: Istutuskausi_5a

Korjataan ensin kullan säilyminen pelistä toiseen. Korjaa **rakennaNakyma** -metodia niin, että se saa parametrina () -sulkujen sisällä tekstiä (`String`) esim. nimeltä `varastokulta`.

Korjaa **run** -metodissa `rakennaNakyma();` -riviä niin, että sille annetaan sulkujen väliin varastokulta. Sulkujen välissä talvivarastolistalta pitää siis pyytää sen nollas alkio:

```
talvivarasto.get(0)
```

Tehtävä: Istutuskausi_5b

rakennaNakyma -metodin alussa sijoita (=) `kulta` -muuttujaan varastokullan arvo. Tekstistä saa numeron komennolla `Integer.parseInt(varastokulta)`, tai `Double.parseDouble(varastokulta)`, jos kulta oli ohjelmassasi desimaaliluku.

Saman metodin alussa kullan määrä kirjoitetaan JLabeliin. Korvaa numeron kohdalle oikea `kulta` -muuttujan arvo tyyliin:

```
"Sinulla on " + kulta + " kultaa."
```

Tehtävä: Istutuskausi_6a

Korjaa nyt **kynnaKasvimaa** -metodia niin, että se saa parametrina () -sulkujen sisällä String-listan (`ArrayList<String>`) esim. nimeltään `varasto`.

Korjaa vastaavasti **run** -metodin `kynnaKasvimaa();` -riviä niin, että sille annetaan sulkujen väliin koko talvivarasto.

Tehtävä: Istutuskausi_6b

Istutuksia tehdessä meillä ei ole vielä mitään työkaluja istuttaa talvehtineita kasveja. Muokataan **Istutus** -luokkaa niin, että istutusta luodessa myös sen kasvuvaihe annetaan valmiina. Korjaa kyseisen luokan konstruktoria (`public Istutus(Puutarha pelilogiikka)`) niin, että Puutarhan lisäksi sille annetaan sulkujen sisällä `String varastotieto`.

Palaa **Puutarha** -luokkaan ja lisää kaksoissilmukalle istutuksien laskuri, jotta `talvivarasto` -listasta voidaan antaa kullekin istutukselle oikea varastotieto. Aloitetaan istutusten laskeminen nyt yhdestä, sillä listan nollannessa alkiossa oli kullan määrä.

Täydennä ohjelmaasi seuraavalla tavalla. Vaaleamman harmaat rivit ovat vertailua varten eivätkä ne ole tässä täydellisesti kirjoitettuja, älä muuta niitä.

```
int laskuri = 1;
for (int rivi=0; …
    for (int sarake=0; …
        Istutus istutus = new Istutus(this, talvivarasto.get(laskuri));
        istutukset.add(…
        kasvimaa.add(…
        laskuri++;
    }
}
```

Tehtävä: Istutuskausi_6c

Korjataan vielä istutusta niin, että se tiedon vastaanottamisen lisäksi käyttää sitä.

Istutus -luokan konstruktorissa on rivi, jolla kasvuvaihe asetetaan nollaan. Kirjoita nollan tilalle, että varastotieto -merkkijonosta luodaan kokonaisnumero:

```
Integer.parseInt(varastotieto)
```

Korjaa myös setIcon -komentoa hieman alempana, että kuvaksi ei aseteta nollatta kuvaa kuvalistasta vaan `this.kasvuvaihe` mones kuva.

Testaa, että peli avaa nyt viimeisimmän tallenteensa tilanteen uudelleenkäynnistettäessä. Jos tallenteesi on hävinnyt luo Talvivarastossa nimetty tiedosto projektin kansioon ja siihen 1. riville: 10.0 ja istutusten määrän mukaisesti seuraaville riveille: 0

Bonus: Istutuskausi_7

Jos tiedostoa ei löydy avattavaksi, ohjelma tulostaa tällä hetkellä konsoliin `e.printStackTrace()` -virheviestin catch-osiossa. Normaali pelaaja ei tarkkaile konsolia, joten hänelle voitaisiin antaa pop up -viesti.

Lisää catch-osioon `JOptionPane.showMessageDialog(null, "kirjoita viesti tähän")` -viesti. Viestissä voi lukea vaikkapa: "Ei tallennetta. Peli aloitetaan alusta." Nyt voit myös poistaa `e.printStackTrace()` -komennon halutessasi.

Tehtävä: Istutuskausi_8

Puutarha voisi teoriassa sisältää kaksi eri tapaa aloittaa istutuskausi; aloittamalla pelin alusta tai tallenteen löytyessä viime pelin pohjalta. Ohjelmointi on kuitenkin työläämpää, jos puutarhaa istuttaessa pitää koko ajan valvoa kummin istutus tehtiinkään.

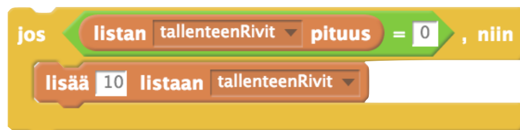
Talvivaraston avaaVarasto -metodi voi palauttaa alkuasetelmaa vastaavan varastotilanteen, vaikka tiedostoa ei löytynytkään. Tällöin koko puutarhan voi luoda varaston palauttaman tiedon pohjalta vaikka varasto oli käytännössä tyhjä.

1) Luo **Talvivarasto** -luokkaan uusi metodi, joka saa tiedokseen istutusten määrän ja tallenteen rivit:

```
private static void taydennaPuuttuvatVarastotiedot(int maara,
ArrayList<String> tallenteenRivit) {

}
```

2) Kirjoita edelliseen metodiin, että kullan määrä aluksi kirjoitetaan, jos sitä ei löydy tallenteesta. Kirjoita lisäyskomento tekstinä, esim. `tallenteenRivit.add("10");` Listan pituuden saat `tallenteenRivit.size()` -komennolla.



3) Toista sen jälkeen metodissa istutusten kasvuvaiheen tallennerivejä, kunnes listan pituus on istutusten määrä + 1. (+1 tulee kullan määrän rivistä.)

```
while (rivit.size() < maara+1) {
    rivit.add("0");
}
```

Tehtävä: Istutuskausi_9

Lisää **avaaVarasto** -metodin loppuun ennen return -riviä seuraava metodikutsu:

```
taydennaPuuttuvatVarastotiedot(istutustenMaara, rivit);
```

Jos listasi nimi **avaaVarasto** -metodissa on jotain muuta kuin `rivit`, kirjoita oman listasi nimi sen tilalle.

Pelin pitäisi nyt toimia vaikka tallennetiedostoa ei löytyisi.

Seuraavaksi pelissä voisi olla enemmän kasveja. Laita sähköpostia, jos haluat seuraavat ohjeet rajapinnoista: jenna.tuominen(at)cs.helsinki.fi