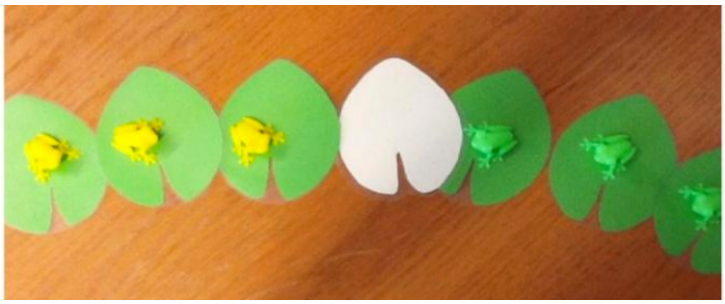
Javan GUI Scratchaajalle

Tehtävä: Sammakkolampi_1

Luo uusi java-projekti. Nimi voi olla vaikka “Sammakkolampi”. Anna Netbeansin luoda Main Class sen nimisenä kuin se ehdottaa.

Sammakkopelissä ideana on, että sammakot ovat kukin omalla lumpeenlehdellään. Lumpeenlehdet ovat jonona. Vasemmasta päästä jonoa yksi joukkue sammakoita koittaa päästä joen yli oikealle puolelle ja oikeasta päästä jonoa toinen joukkue koittaa päästä vasemmalle puolelle. Yksi lumpeenlehti jaksaa kantaa vain yhden sammakon painon ja sammakko jaksaa hypätä korkeintaan yhden sammakon yli. Sammakot eivät ymmärrä peruuttaa.



Kuva: Summamutikka-keskuksen loikkivat sammakot.
Kuvaajana Juulia Lahdenperä ja Inkeri Sundqvist.

Kertaus: Olioiden rakenne

Olio ja luokka

Olio, luokka, tai siis Scratchin hahmo, määritellään seuraavasti:

```
public class Hahmo {
    private int koko = 3;
    private String nimi;

    public Hahmo(String annettuNimi) {
        this.nimi = annettuNimi;
    }

}
```

Uloimpien { } -sulkujen alussa määritellään, että `Hahmo` niminen luokka (`class`) on julkinen (`public`). Luokan { } -sulkujen väliin kirjoitetaan kaikki, mitä luokka on tai osaa tehdä. Alimman } -sulun alle ei kirjoiteta mitään.

Luokassa tyypillisesti ensimmäisenä määritellään luokan tuntemat tiedot, muuttujat.

Muuttujien jälkeen voidaan tehdä luokan rakennusohje tai siis *konstruktori*, eli mitä tehdään kun luokka ensimmäisen kerran luodaan. Edellisessä esimerkissä rakennusohje sai myös taustatietoa: `(String annettuNimi)`. Taustatietoa voi antaa halutessaan, tai olla antamatta, jolloin merkitään vain tyhjät sulut: `()`.

Metodit

Tyypillisesti luokan rakennusohjeen jälkeen listataan kaikki luokan tuntemat komennot eli *metodit*. Voit verrata näitä Scratchin vastaanotettuihin viesteihin: “kun vastaanotan viestin siirrä, kasvata paikka-muuttujaa, jos se on kasvatettavissa”.

```
public class Hahmo {
    private int paikka;

    public class Hahmo() {
        this.paikka = 0;
    }

    public void siirra(int maara) {
        if ( voikoSiirtaa() ) {
            this.paikka = this.paikka + maara;
        }
    }

    private boolean voikoSiirtaa() {
        return true;
    }
}
```

Metodi voi olla julkinen (`public`) tai vain luokan tuntema (`private`). Se voi olla palauttamatta mitään lopputulosta (void) tai palauttaa esimerkiksi totuustiedon (`boolean`: `true` tai `false`). Jos jotain palautetaan, tulee metodissa olla `return` -komento. Näiden jälkeen tulee määrittää vielä komennon nimi, kuten “`siirra`”.

Myös metodi, kuten konstruktori, voi saada taustatietoa eli *parametreja*. Edellisessä esimerkissä parametrina annettiin `maara`, eli komennon kutsujan määrittelemä numero.

Perinnöllisyys ja lupaukset

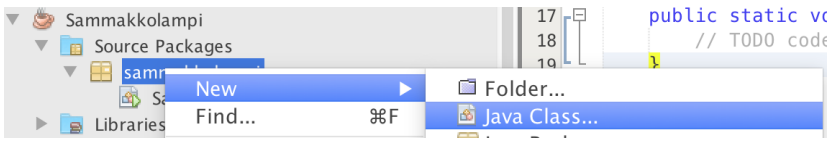
Luokkaan ei tarvitse käsin määrittää kaikkea, mitä luokka osaa. Se voi periä kaikki jonkin toisen luokan osaamat asiat `extends` -komennolla. Periä voi vain yhden luokan, mutta voi olla vaikka, että `Kissa` -luokka perii `Elain` -luokan, ja tällöin, jos `Persialainen` perii `Kissa` -luokan, niin se saa samalla myös kaikki `Elain` -luokan metodit.

```
public class Kissa extends Elain {}
```

Lupauksia voi antaa useita `implements` -komennolla. Lupaukset pitää myös pitää, ja esimerkiksi seuraavan tehtävän Runnable -lupaukseen pitää ohjelmoida `run` -metodi.

Tehtävä: Sammakkolampi_2

Tee projektiin uusi luokka nimeltä Lampinakyma ja luo sille Runnable -lupaus ja toistaiseksi tyhjä run -metodi.




```
public class Lampinakyma implements Runnable {

    @Override
    public void run() {

    }
}
```

Tehtävä: Sammakkolampi_3

Anna **`Sammakkolampi`** -luokan main-metodissa komento käynnistää graafinen ohjelma:

```
SwingUtilities.invokeLater( new Lampinakyma() );
```

Kirjastot

Javassa kaikkea ei tarvitse ohjelmoida itse vaan käyttöön voi pyytää “ohjekirjoja” eli *kirjastoja*. Tämä tapahtuu `import` -komennolla aivan luokan alussa:

```
import java.util.Scanner;
import javax.swing.JFrame;

public class Hahmo {

}
```

Eri kirjastot löytyvät eri osoitteista, kuten edellisen Scannerin osoite `java.util.` tai JFramen osoite `javax.swing.` Kirjastoja voi projektia luodessa lisätä tarvittaessa hehkulampusta.

```
    private JFrame ikkuna;
```

Ikkunat

Tehtävä: Sammakkolampi_4

Määrittele luokalle yksityinen (=private) JFrame -muuttuja, jotta mikä tahansa luokan sisäinen metodi eli komento voi antaa komentoja JFrame -ikkunalle.

```
public class Lampinakyma implements Runnable {
    private JFrame ikkuna;
}
```

Aseta **`run`** -metodissa ikkunalle arvoksi `new JFrame()`, eli määritä ensin, että edellinen `ikkuna` - muuttuja tulee vastaisuudessa olemaan jotain uutta:

```
this.ikkuna =
```

ja kirjoita sitten = -merkin jälkeen arvo.

Määritä samassa metodissa ikkunalle otsikko, koko, sulkumekanismi ja näkyvyys:

```
ikkuna.setTitle(“Valitse sopiva ikkunaotsikko.”);
ikkuna.setSize(400, 200);
ikkuna.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ikkuna.setVisible(true);
```

Valinnaisesti voit määrittää myös taustavärin. Ikkuna ei hoida tätä toiminnallisuutta, vaan siltä voi pyytää sisällön hallitsijan, jolle sitten voi antaa komennon:

```
ikkuna.getContentPane().setBackground(Color.BLUE);
```

Testaa, että ikkuna aukeaa, kun projekti ajetaan.

Komponentit

Ikkunaan voi laittaa eri tarkoituksiin tehtyjä komponentteja:

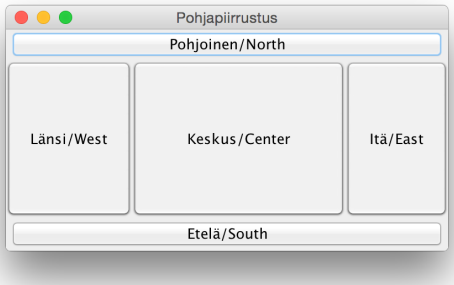
Etiketti	`JLabel("Sitruunapippuri");`
Nappula	`JButton("Klikkaa tästä.");`
Lyhyt tekstikenttä	`JTextField();`
Pitkä tekstikenttä	`JTextArea();`
Tyhjä alue uusille komponenteille	`JPanel();`

Lisääminen onnistuu komennolla `add` (lisää):

```
ikkuna.add( new JButton("Testinappula") );
```

Asettelu

Kaikilla ikkunoilla on automaattisesti BorderLayout -asettelu, jossa komponentteja voi asettaa reunoille ja keskelle:



Yhteen lokeroon voi asettaa vain yhden komponentin, mutta toisaalta JPanel -komponentin sisään voi laittaa useita. Kaikkia lokeroita ei tarvitse täyttää. Tiettyyn lokeroon lisääminen onnistuu seuraavasti:

```
ikkuna.add( new JLabel("OK"), BorderLayout.EAST );
```

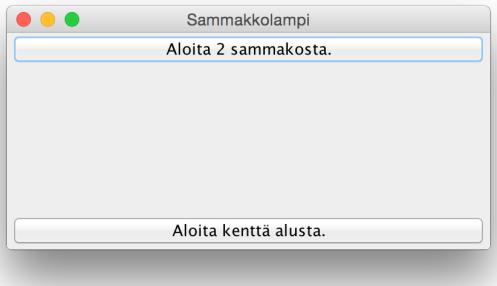
Tehtävä: Sammakkolampi_5

Luodaan kohta peliympäristö. Tee tätä tarkoitusta varten ensin uusi metodi **`Lampinakyma`** -luokkaan:

```
private void rakennaIkkuna() {

}
```

Luo **`rakennaIkkuna`**-metodissa *pääpiirteissään* seuraavan kaltainen ikkuna.



Kirjoita `rakennaIkkuna();` -metodikutsu **`run`** -metodin viimeiseksi komennoksi, kun JFrame -ikkuna on jo luotu ja määritelty.

Tehtävä: Sammakkolampi_6

Luo **Lampinakyma** -luokkaan toinenkin metodi, joka luo ja palauttaa JPanel -olion:

```
private JPanel rakennaLampi() {
    JPanel lampi = new JPanel();

    return lampi;
}
```

Lisäsit mahdollisesti aiemmin ikkunan CENTER -osaan tyhjän JPanel -olion **rakennaIkkuna** -metodissa. Korvaa se nyt komennolla rakentaa lampi:

```
ikkuna.add( rakennaLampi(), BorderLayout.CENTER );
```

Asetteluja on muitakin kuin reunoille asettelu. Toinen yksinkertainen asettelu on ruudukkoasettelu. Sille voi antaa syötteenä rivien ja sarakkeiden määrän (tässä järjestyksessä). JFrame ja JPanel -olioille asettelun voi asettaa `setLayout` -komennolla:

```
ikkuna.setLayout( new GridLayout(1,5) );
```

Tehtävä: Sammakkolampi_7

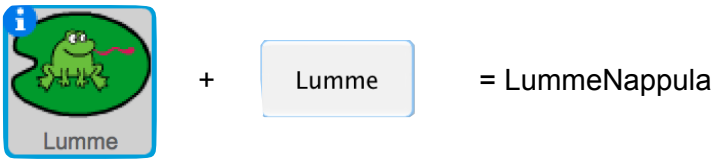
Lisää edelliseen **rakennaLampi** -metodiin lammen luomisen ja palautuksen väliin komento asettaa lammen asetteluksi ruudukkoasettelu (yksi rivi ja viisi saraketta).

Ruudukko ei vielä näy, mutta testaa, että ohjelma ei silti luo tässä vaiheessa mitään virheitä ajettaessa.

Perintä, osa 2

Ikkunan keskelle olisi tarkoitus luoda lumpeenlehtiä. Osalla voi olla sammakko matkalla oikealle, osalla vasemmalle. Osalla ei ole sammakkoa ollenkaan. Sammakkoihin liittyy niin paljon sääntöjä, ettei niitä kannata antaa jokaiselle lumpeelle erikseen. Lumpeesta kannattaa siis tehdä olio, jota voimme sitten kloonata useammaksi lumpeeksi.

Lumpeita pitäisi myös pystyä klikkaamaan, jotta niillä oleva sammakko ymmärtää hypätä. Tähän soveltuisi JButton -nappula, mutta sillä ei toisaalta ole lumpeiden muita ominaisuuksia.



Ongelman ratkaisee perintä. Javassa voimme tehdä lummenappulan, joka perii JButton -nappulalta kaikki sen ominaisuudet, mutta jolle voimme myös kirjoittaa uusia sääntöjä.

Javassa luokka pystyi tosiaan perimään toisen luokan extends -komennolla:

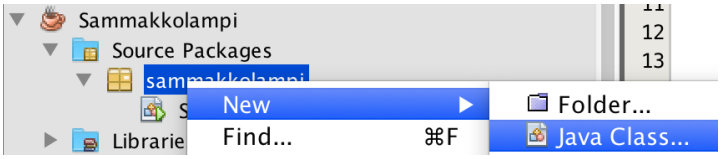
```
public class Kissa extends JButton {

}
```

“Extend” tarkoittaa sanatarkasti laajentamista. Perittyjen ominaisuuksien lisäksi luokalle tehdään lisää toimintoja, joten sen toimintaa ollaan laajentamassa.

Tehtävä: Sammakot_1

Luo Sammakkolampi -projektiin uusi luokka nimeltään Lumme.



Kirjoita Lumme -luokan määritelmään (`public class` -rivi), että se laajentaa (extends) JButton -luokan toimintaa.

Tehtävä: Sammakot_2

Lumme on tällä hetkellä vain JButton ilman lummemaisia ominaisuuksia. Kun se luodaan sillä pitäisi olla vähintään lumpeen ulkonäkö.

Luo:
- tyhjän lumpeen kuva
- kuva lumpeesta, jolla on sammakko matkalla oikealle
- kuva lumpeesta, jolla on sammakko matkalla vasemmalle

Nämä kuvat voi luoda vaikka sivustolla scratch.mit.edu, kun valitset sivun ylälaidasta Create (tai Luo) ja klikkaat sitten ylälaidan Costumes (tai Asusteet) -välilehteä. Voit valita välilehden ylälaidasta valmishahmon “pikku tyyppi” -ikonista tai piirtää oman “sivellin” -ikonista. Asusteita voi kopioida useampaa kuvaa varten.

Scratch-kuvan voit tallentaa, kun asusteen pikkukuvaa klikkaa oikealla painikkeella ja valitsee “save to local file” (tai “tallenna paikalliseen tiedostoon”). Huom! Jos kuva on vektorimuodossa, eli työkalut ovat oikeassa laidassa muuta kuva ensin bittikuvaksi alhaalta oikealta.

Tehtävä: Sammakot_3

Luo `Lumme` -luokalle konstruktori, joka ottaa vastaan tiedon sammakon suunnasta (-1, 1), tai sammakottomuudesta (0):

```
private int suunta;

public Lumme(int sammakollisuus) {
    this.suunta = sammakollisuus;
}
```

Kuvat

Javassa kuvia voi tuoda ohjelmaan monella tavalla, mutta nappuloihin niitä saa helposti ImageIcon -muodossa. Kuvatiedosto pitää ensin tallentaa tähän ImageIcon “-hahmoon” ja sen voi sitten antaa nappulalle piirrettäväksi.

```
ImageIcon kuva = new ImageIcon(“Kissa.png”);
JButton nappula = new JButton();
nappula.setIcon(kuva);
```

Kuvan voi toisaalta antaa nappulalle jo sitä luotaessa, vaikka käytämme seuraavassa tehtävässä edellistä menetelmää.

```
JButton nappula = new JButton( new ImageIcon(“Maisema.png”) );
```

Tehtävä: Sammakot_4

Aseta **Lumme** -luokan konstruktorissa kuvaksi toistaiseksi aina tyhjä lumpeenlehti. Konstruktorissa ei tarvitse luoda uutta JButton -nappulaa, koska Lumme *on* JButton. Voit siis kirjoittaa `this.setIcon...`

Palaa **Lampinakyma** -luokkaan ja lisää **rakennaLampi** -metodiin asettelun määrityksen perään, että lampeen lisätään viisi `Lumme` -oliota.

```
lampi.add( new Lumme(0) );
```

Testaa, että tyhjät lumpeet ilmestyvät näkymään, kun ohjelma nyt käynnistetään. Jos nappulat ilmestyvät ilman kuvia varmista, että kuvat ovat projektin kansiossa.

Kuvien koko

Lumpeet kyllä edellisessä tehtävässä “mahtuvat” näkyviin, mutta jos kuvien koko oli suuri eivät ne näy kokonaan. Kuvan koon muuttaminen on mahdollista ohjelmassa. Tässä yksi ratkaisu, joka on lyhyt vaikkei kenties paras:

```
ImageIcon ikoni = new ImageIcon();

try {
    Image kuva = ImageIO.read( new File(“Tyhja.png”) );
    ikoni.setImage( kuva.getScaledInstance(50, 100, Image.SCALE_SMOOTH) );
} catch (Exception e) {}

this.setIcon(ikoni);
```

Ratkaisussa ikoni on aluksi tyhjä ja kuva luetaankin ImageIO -työkalulla tiedostosta Image -olioon. Ikonille sitten asetetaan kuva (`setImage`), kun Image -oliosta pyydetään skaalattu versio (scaled instance). Skaalatessa ohjelma haluaa tietää leveyden, korkeuden ja skaalausmenetelmän.

Kuvan lukemiseen tiedostosta liittyy myös virhekäsittelyä, sillä Image -oliolle on tarjottava varmasti olemassa oleva tiedosto. Jos tiedostoa ei löydy siirtyy ohjelma yrityksen (try) jälkeen `catch` -osioon. Tämän ohjelman catch -osio voi nyt olla tyhjä ja kuvaa ei vain näytetä, jos ohjelman suorituksessa käy virhe.

Tehtävä: Sammakot_5

Lisää **Lumme** -luokkaan uusi metodi, johon kirjoitat sisään edellisen esimerkin mukaisen kuvan asetuksen:

```
private void asetaKuva() {

}
```

Saman luokan konstruktorissa poista aiempi tyhjän lumpeen kuvan asettaminen ja kirjoita tilalle `asetaKuva()` -metodikutsu.

Tehtävä: Sammakot_6

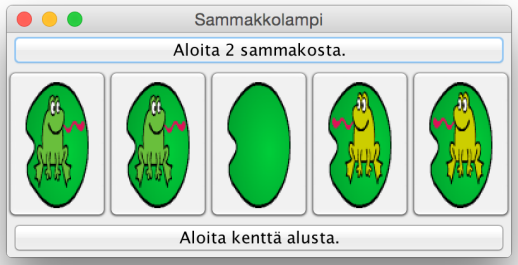
Muuta kuvien nimet muotoon:
- 0.png, kun lehti on tyhjä
- 1.png, kun sammakko katsoo oikealle
- -1.png, kun sammakko katsoo vasemmalle

Korjaa asetaKuva() -metodia niin, että tiedosto onkin: `new File(suunta + ".png")`

Tehtävä: Sammakot_7

Muuta Lampinakyma -luokan rakennaLampi -metodia niin, että viidestä lumpeesta kaksi ensimmäistä saa suunnaksi arvon 1, kolmas saa suunnaksi arvon 0 (eli sammakkoa ei ole) ja kaksi viimeistä saavat suunnaksi arvon -1.

Testaa, että sammakot ovat seuraavassa asetelmassa, kun käynnistät ohjelman.



Listat

Pelissä luomme useita lumpeenlehtiä. Yksittäisten lumpeiden käsitteleminen on hankalaa, joten luodaan niistä lista. Listojakin löytyy useita erilaisia eri tarkoituksiin, mutta yksi helpoimmista on ArrayList, joka kasvattaa pituuttaa tarvittaessa ja josta on helppo löytää lista-alkioita järjestysnumeron perusteella.

ArrayList -olioon voi lisätä vain sille alussa määritellyn mukaisia alkioita. Tässä esimerkissä listaan lisätään String-, eli merkkijonoalkioita:

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

nimilista.add("Emma");
nimilista.add("Jenna");
nimilista.add("Matias");
```

ArrayList -olion alkioiden määrän voi kysyä:

```
nimilista.size();
```

Tietyn alkion saa selville järjestysnumerolla:

```
nimilista.get(0);
```

Tehtävä: Lummeliikenne_1

Lampinakyma -luokan alussa on toistaiseksi vain yksi oliokohtainen muuttuja, `ikkuna`. Määritä sen alle nyt myös:

```
private ArrayList<Lumme> lummejono = new ArrayList<Lumme>();
```

rakennaLampi -metodissa luo `Lumme` -oliot omalla rivillään ja lisää ne sitten ikkunan lisäksi myös listaan.

```
Lumme sammakkolumme1 = new Lumme(1);
lampi.add(sammakkolumme1);
lummejono.add(sammakkolumme1);
```

Testaa, että sammakot yhä ilmestyvät oikein näkymään.

Tehtävä: Lummeliikenne_2

Lampi tuntee nyt lumpeet, ja niiden päällä mahdollisesti oleilevat sammakot, mutta sammakot eivät tunne lampea. Sammakot toisaalta tulevat tietämään, jos niitä klikataan, ja niiden tulisi lammen kautta selvittää voivatko ne hyppiä eteenpäin.

Korjaa uuden lumpeen luomista niin, että sille annetaan mukaan myös `this` -viite lampeen:

```
Lumme sammakkolumme = new Lumme(this, 1);
```

Korjaa vastaavasti **Lumme** -luokan konstruktoria niin, että se ottaa vastaan lampiviitteen ja tallentaa sen omiin muuttujiinsa

```
private Lampinakyma omaLampi;

public Lumme(Lampinakyma lampi, int sammakollisuus) {
    this.omaLampi = lampi;
```

Kuuntelijat

Nappuloilla on kyky reagoida niiden klikkaamisiin, mutta ohjelmoijan pitää silti lisätä niille kuuntelija. Samoin monilla muilla ikkunakomponenteilla on kyky vastata erilaisiin tapahtumiin.

Kuuntelija on olio, joka lupaa toteuttaa jonkin *rajapinnan* kuten vaikka `ActionListener`. Kuhunkin rajapintaan liittyy tietyt metodit, jotka luokan pitää tällöin sisältää. Tässä esimerkki klikkauksia kuuntelevasta oliosta:

```
import java.awt.event.*;
import javax.swing.JOptionPane;

public class HeiKuuntelija implements ActionListener {

    public void actionPerformed(ActionEvent ae) {
        JOptionPane.showMessageDialog(null, "Hei!");
    }
}
```

Kuuntelija voi olla erillinen luokka tai jokin ohjelman jo olemassa olevista luokista voi toteuttaa tämän toiminnallisuuden. Kuuntelija lisätään nappulalle seuraavalla tavalla tai sulkujen sisään kirjoitetaan `this`, jos kuuntelija on luokka itse.

```
JButton nappula = new JButton("Sano hei.");
nappula.addActionListener( new HeiKuuntelija() );
```

Tehtävä: Lummeliikenne_3

Tee **`Lumme`** -luokasta kuuntelija, eli kirjoita, että se implementoi ActionListener -ominaisuuden. **`Lumme`** on perinyt jo `JButtonin`, joten kirjoita implementointi `extends JButton` -kohdan jälkeen (älä laita pilkkua näiden väliin).

Tee luonnollisesti Lumpeelle myös **`actionPerformed`** -metodi, joka toistaiseksi sanoo vain "Hei!", kuten edellisessä esimerkissä näytettiin.

Kirjoita **`Lumme`** -luokan konstruktoriin, että sille lisätään kuuntelijaksi se itse.

```
this.addActionListener(this);
```

Kokeile, että lumpeenlehdet sanovat "Hei!", kun niitä klikataan.

Tehtävä: Lummeliikenne_4

Jotta sammakot voisivat loikkia lumpeelta toiselle, pitää lammen olla kykeneväinen selvittämään onko jokin tietty lumme tyhjä vai ei, ja sen pitää toisaalta olla myös kykeneväinen, muuttamaan lumpeen tilaa sammakolliseksi tai tyhjäksi.

Tee lumpeelle seuraavat metodit ja ohjelmoi metodiin harmaalla ohjeistettu toiminnallisuus.

```
public boolean onkoTyhja() {
    //palauta true, jos suunta on (==) 0, muuten palauta false
}

public void asetaSuunta(int uusiSuunta) {
    //aseta this.suunta arvoon uusiSuunta
    //kutsu sitten asetaKuva() -metodia
}
```

Tehtävä: Lummeliikenne_5

Seuraavaksi ohjelmoi **Lampinakyma** -luokkaan uusi metodi, joka ensin selvittää monettako lummetta klikattiin, kun klikkauksen tunteva lumme ilmoittaa itsensä:

```
public void suoritaHyppy(Lumme alkulumme, int suunta) {
    int monesko = lummejono.indexOf( alkulumme );

}
```

Järjestysnumeron 'monesko' selvittämisen jälkeen laske mihin sammakko olisi menossa, jos kohde olisi tyhjä:

```
monesko = monesko + suunta;
```

Jos numero on pienempi kuin 0 tai yhtä suuri kuin listan pituus, pysäytä metodin suoritus:

```
if (monesko < 0 || monesko >= lummejono.size()) {
    return;
}
```

Jos metodin suoritus ei loppunut, pyydä seuraavaksi listasta numeroa vastaava lumme, kysy siltä onko se tyhjä:

```
if ( lummejono.get(monesko).onkoTyhja() ) {

}
```

Jos lumme oli tyhjä:
- Anna `lummejono.get(monesko)` -oliolle käsky asettaa suunnaksi metodille annettu suunta.
- Anna `alkulumme` -oliolle käsky asettaa suunnaksi 0, eli muuta se tyhjäksi.

Esim.
```
alkulumme.asetaSuunta(0);
```

Huom! Sammakot eivät vielä liiku, sillä mikään ohjelman osa ei anna komentoa `suoritaHyppy`.

Tehtävä: Sammakkoliikenne_6

Nyt pitäisi siis suorittaa hyppy, *jos* jotain sammakkoa klikataan. Tai hyppyä pitää siis vähintään yrittää.

Tyhjennä **Lumme** -luokan **actionPerformed** -metodista muut komennot ensin. Kirjoita metodiin sitten, että jos lumpeelta löytyy sammakko, `omaLampi` -oliolle annetaan komento `suoritaHyppy(this, suunta)`. Sammakon löytymisen voi testata esim. testaamalla, että suunta ei ole (`!=`) nolla.

Muista, että komentoa annettaessa kirjoitetaan ensin komennettavan hahmon nimi (`omaLampi`), sitten piste ja sitten edellinen komento sulkuineen ja syötteineen.

Testaa, että sammakot osaavat hypätä seuraavalle lumpeelle, jos kyseinen lumme on tyhjä.

Tehtävä: Sammakkoliikenne_7

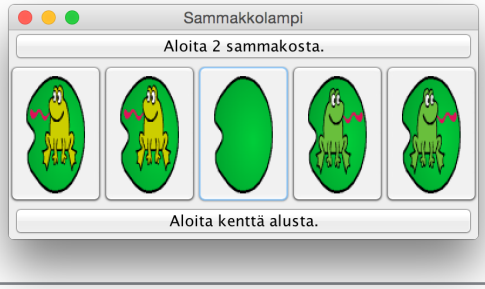
Täydennä `Lampinakyma` -luokan `suoritaHyppy` -metodiin onnistuneen hypyn perään `return` komento (suunnan asetusten jälkeen, ennen kuin if-lohko päättyy).

Lisää yhden hypyn yrityksen perään yritys hypätä edellä olevan sammakon yli. Voit siis laittaa yrityksen silmukkaan, joka suoritetaan kahdesti. Sisällytä yritykseen kaikki monesko = monesko + suunta; -rivistä toisen if-ehdon loppuun asti.

```
for (int yritys = 1; yritys <= 2; yritys++) {

}
```

Testaa jälleen pelin toiminta ja testaa myös saatko sammakot kulkemaan niiden haluamalleen puolelle lampea.



Tehtävä: Voitto_1

Luo `Lampinakyma` -luokkaan uusi metodi, joka tarkastelee ovatko sammakot päässeet kentän toiselle puolelle:

```
private void tarkistaVoitto() {

}
```

Tehtävä: Voitto_2

Voittoa tarkistaessa ohjelman tulee käydä kaikki listan lumpeet läpi järjestyksessä. Kirjoita siis *for each* -lohko:

```
for (Lumme vuorossaOleva : lummejono) {

}
```

For each käy annetusta listasta jokaisen alkion läpi ja kulloinkin käsiteltävänä olevaa alkiota kutsutaan lohkon sisällä “vuorossaOleva” -nimellä (tai minkä ohjelmoija kirjoittaa tuon tilalle). For each:n syntaksiin, eli kirjoitusasuun, kuuluu myös, että listan alkioiden tyyppi pitää määrittää. Tässä tapauksessa listalla oli `Lumme` -olioita.

Tehtävä: Voitto_3

Lumpeilla olevien sammakoiden suuntaa pitää tarkastella, joten kirjoita Lumme -luokalle seuraava metodi ja kirjoita harmaan ohjerivin tilalle ohjeen mukainen komento:

```
public int annaSuunta() {
    //palauta suunta -muuttujan arvo
}
```

Silmukat

Ohjelman komentoja voi käskeä suoritettavan monta kertaa for- ja while- silmukoilla. Scratchissa näitä voi verrata “toista _ kertaa” ja “toista kunnes _” -palikoihin.

For -silmukassa määritetään ensin alkupiste, sitten loppupiste ja sen jälkeen miten pitkiä askeleita tällä välillä otetaan. Javan ++ tarkoittaa, että muuttujan arvoa kasvatetaan yhdellä.

```
for (int laskuri = 0; laskuri <= 5; laskuri++) {
    System.out.println(“Rivi ” + laskuri + “.”);
}
```

While -silmukassa testataan jonkin silmukan ulkopuolisen muuttujan arvoa. Toistoa jatketaan kunnes väite ei pidä enää paikkaansa:

```
int numero = 1;
while (numero < 20) {
    numero = numero * 2;
}
```

Näiden lisäksi on myös siis olemassa *for each*, joka käy läpi kokonaisia listoja alusta loppuun.

Tehtävä: Voitto_4

Aiemman tarkistaVoitto -metodin pitäisi tarkistaa, että sammakoiden suunnat ovat jonossa alussa vasemmalle, sitten löytyy yksi tyhjä lumme ja lopuksi oikealle osoittavia sammakoita. Tarkistetaan siis, että lumpeita tarkastellessa suunta alkaa -1:stä ja sen jälkeen kasvaa aina korkeintaan yhdellä. Esim. seuraava olisi hyväksyttävä jono:

```
-1 -1  0  1  1
```

Korvaa seuraavassa *pseudokoodissa*, eli koodissa jota tietokone ei ymmärrä, kaikki harmaat rivit oikealla koodilla, joka tekee harmaalla kirjoitetun asian.

```
// aseta uusi kokonaislukumuuttuja ‘oletussuunta’ arvoon -1
for (Lumme vuorossaOleva : lummejono) {
    // jos vuorossaOleva.annaSuunta() ei ole oletussuunta {
        // kasvata oletussuunta -muuttujan arvoa yhdellä
        // jos vuorossaolevan suunta ei vieläkään ole oletussuunta {
            // palaa pois metodista (return)
        // }
    // }
}
```

Jos ohjelma pääsee for -silmukan loppuun pysähtymättä, ovat lumpeet hyväksyttävässä ja siis myös oikeassa järjestyksessä. Laita silmukan jälkeen pop up -ikkunaan voittoviesti pelaajalle.

```
JOptionPane.showMessageDialog(null, “Hienoa!”);
```

Tehtävä: Haastetaso_1

Haastetta peliin saa sammakoita lisäämällä ja samalla jonoa pidentämällä. Jos jonosta tulisi vaikka 21 lummetta pitkä, olisi jo kätevämpää, että yksittäisten lumpeiden lisääminen automatisoidaan. Tee seuraava metodi **`Lampinakyma`** -luokkaan.

```
private void lisaaLumpeet(JPanel kohdelampi, int maara) {

}
```

Tehtävä: Haastetaso_2

Lumpeet luotiin aiemmin seuraavasti:

```
Lumme sammakkolumme1 = new Lumme(this, 1);
lampi.add(sammakkolumme1);
lummejono.add(sammakkolumme1);
```

Käytä nyt edellisellä sivulla näytettyä for -silmukkaa kahdesti **`lisaaLumpeet`** -metodissa:

- Luo ensimmäisessä silmukassa lumpeita, joilla sammakko osoittaa suuntaan 1, `maara` verran kertoja. Aloitetaan siis silmukka numerosta 1 ja lopetetaan numeroon `maara`.
- Luo silmukan jälkeen yksi tyhjä lumme (jonka suunta oli siis nolla).
- Luo tämän jälkeen toisessa silmukassa lumpeita, joilla sammakko osoittaa suuntaan -1, jälleen `maara` verran kertoja.

Tehtävä: Haastetaso_3

Lisätään Lampinakyma -luokalle laskuri, joka tarkkailee tasoa.

```
private int tasolaskuri = 2;
```

Tehtävä: Haastetaso_4

Korvaa **`rakennaLampi`** -metodin pitkät rivit lumpeiden lisäilyjä äsken rakentamallamme metodin kutsulla. Annetaan kutsulle myös mukaan **`rakennaLampi`** -metodissa tehtävä lampi, joka tulee täyttää, sekä yhteen suuntaan liikkuvien sammakoiden määrä:

```
lisaaLumpeet(lampi, tasolaskuri);
```

Testaa, että sammakot tulevat paikoilleen kuten aiemminkin.

Tehtävä: Haastetaso_5

Varaudutaan tason vaihtoon:

1. Korjaa **`rakennaLampi`** -metodiin, että GridLayout ei sisällä 5 lummetta vaan 2* tasolaskurin verran plus 1 tyhjä.

2. Korjaa myös **`lisaaLumpeet`** -metodin alkuun, että lista tyhjennetään ennen uusien lumpeiden lisäystä. Tähän käy `lummejono.clear()` -komento.

3. Korjaa rakennaIkkuna -metodia niin, että ennen komponenttien lisäystä ikkunan sisältö tyhjennetään edellisistä nappuloista:

   ```
   ikkuna.getContentPane().removeAll();
   ```

   Komponenttien lisäyksen jälkeen “vahvista uusi sisustus”: `ikkuna.revalidate();`

Tehtävä: Haastetaso_6

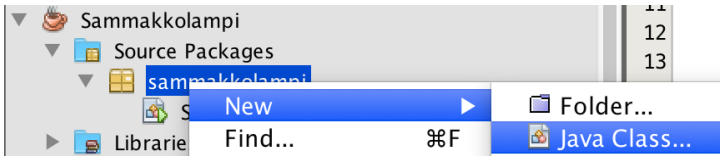
Lisätään **`tarkistaVoitto`** -metodiin viimeiseksi komennot rakentaa lampi jälleen yhtä suurempana. Kasvata tasolaskuria ensin yhdellä, kutsu sitten `rakennaIkkuna()` -metodia.

Testaa, että pääset seuraavalle tasolle.

Tehtävä: Ankkuripisteet_1

Pelissä on alusta asti ollut nappula, joka lupaa alkutasolle palaamisen. Tehdään nyt se.

Nappula tarvitsee toimiakseen kuuntelijan. Luo uusi luokka nimeltä AlkuKuuntelija:



Tehtävä: Ankkuripisteet_2

Kirjoita AlkuKuuntelijaan, että se toteuttaa (implements) rajapinnan `ActionListener` ja luo luokalle **`actionPerformed`** -metodi. Katso apua pari sivua taaempaa materiaalista.

Kirjoita luokkaan myös konstruktori, joka ottaa vastaan parametrina viitteen pelinäkymään sekä oliokohtainen muuttuja, joka muistaa viitteen. Tyypillisesti konstruktori kirjoitetaan muiden metodien yläpuolelle ja oliokohtaiset muuttujat vielä sen yläpuolelle:

```
private Lampinakyma peli;

public AlkuKuuntelija(Lampinakyma nakyma) {
    this.peli = nakyma;
}
```

Tehtävä: Ankkuripisteet_3

Kirjoita actionPerformed -metodiin, että klikkauksen tapahduttua (eli kun metodi suoritetaan) pelille annetaan komento `aloitaAlusta();`

Tämä ei tietenkään toimi niin kauan kuin peli, eli Lampinakyma ei sisällä komentoa **`aloitaAlusta()`**. Luo **`Lampinakyma`** -luokalle julkinen (`public`) metodi **`aloitaAlusta`**, joka ei ota vastaan parametreja (tyhjät sulut) ja joka ei palauta mitään (`void`).

Metodin **`aloitaAlusta`** tulee ensin asettaa `tasolaskuri` -muuttuja arvoon 2 (sammakkoa) ja sitten kutsua metodia `rakennaIkkuna`.

Tehtävä: Ankkuripisteet_4

Alkuun palaaminen ei kylläkään vielä toimi, sillä kuuntelijaa ei ole lisätty nappulalle.

Siirry **`Lampinakyma`** -luokkaan **`rakennaIkkuna`** -metodiin. Jos “alusta aloitus” -nappulan luominen on tehty `add` -komennon sisällä, tulee se nyt luoda ennen tuota riviä ja tallentaa esim. `JButton` -muuttujaan nimeltä `alkunappula`. Tämän jälkeen uusi (`new`) AlkuKuuntelija voidaan lisätä nappulalle komennolla `addActionListener( )`. Huomaa, että AlkuKuuntelijalle pitää syötteenä antaa (sulkujen sisään) Lampinakyma eli `this`.

add -komennossa lisätäänkin sitten juurikin tämä luotu `alkunappula`.

Alusta aloittamiseen tehdyn nappulan pitäisi nyt toimia pelissä.

Tehtävä: Ankkuripisteet_5

Kunkin kentän alusta aloittaminen on luultavasti vielä tärkeämpää pelaajalle, kun sammakot joutuvat aina tavalla tai toisella pattitilanteeseen, jossa kukaan ei voi liikkua.

Luo tätä toista nappulaa varten myös kuuntelija. Se voi olla nimeltään esim. **`NollausKuuntelija`**. Tee se samoin kuin **`AlkuKuuntelija`** tehtävissä Ankkuripisteet_1-2.

Tehtävä: Ankkuripisteet_6

Tehtävän Ankkuripisteet_4 mukaisesti tee myös `JButton nollausnappula` ennen sen `add` -komentoa ja lisää sille kuuntelijaksi juuri luotu **`NollausKuuntelija`**.

Tehtävä: Ankkuripisteet_7

Myös tälle nappulalle voisi tehdä **`Lampinakyma`** -luokkaan uuden metodin, mutta tämä ei ole tarpeen sillä **`Lampinakyma`** -luokalla on jo **`rakennaNakyma`** -metodi, joka tekisi juurikin halutun toimenpiteen. Tehdään nyt päätös, että rakennaNakyma -metodi voikin olla julkinen eli `public`.

Tämän muutoksen lisäksi kuuntelijan pitää vielä antaa komento rakentaa näkymä uudelleen.

Pelin pitäisi nyt toimia, testaa keksitkö logiikan päästä tasoissa eteenpäin?

Tehtävä: Ankkuripisteet_8

Mitä korkeammille tasoille pelissä pääsee sitä ahtaammaksi ikkuna voi käydä. Ohjelmalle voi kuitenkin onneksi antaa myös komennon muuttaa ikkunan kokoa.

Kirjoita rakennaLampi -metodiin komento asettaa ikkunan koko esimerkiksi setLayout -komennon jälkeen. Koko asetetaan setSize -komennolla, joka annetaan JFrame -muuttujalle (katso minkä niminen se sinulla on). Ikkunan korkeus voi pysyä vakiona, mutta laske ikkunalle leveys sen mukaan millä tasolla pelaaja on.

Jos haluat tuunata leveyden vastaamaan ensimmäisellä tasolla valitsemaasi leveyttä voit laskea sen jakamalla alkuperäisen leveyden luvulla 5 (lumpeiden alkuperäinen määrä) ja sitten kertomalla tämän lumpeiden sen hetkisellä määrällä eli tasolaskuri*2+1. Eli tässä lasketaan alkuperäisten lumpeiden leveys ja sitten lasketaan ikkunan leveys lumpeiden määrän mukaan.

Tehtävä: Ankkuripisteet_9

Lopuksi voisi vielä testata miten käy, kun pelin leveys menee ikkunaa leveämmäksi, mutta tämän testaamiseen voi mennä hetki.

Lopputulos on kuitenkin, että viimeiset sammakot jäävät ikkunan reunan taakse piiloon. Ikkunan leveydeksi pitääkin laittaa laskettu arvo tai maksimissaan näytön leveys. Näytön leveyden saa selville ja maksimin rajattua seuraavasti:

```
Dimension naytto = ToolKit.getDefaultToolkit().getScreenSize();
int laskettuLeveys = 400/5 * (tasolaskuri*2+1);
this.ikkuna.setSize( Math.min(naytto.width, laskettuLeveys) , 200);
```

Pelin pitäisi nyt toimia täydellisesti.