Javan GUI Scratchaajalle

Käsin aseteltuja komponentteja

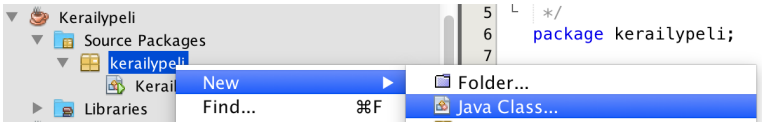
Tehtävä: Pelialue_1

Seuraavaksi tehtävä peli vastaa Scratch-kerhossamme usein tehtyä keräilypeliä. Pelissä on nuolinäppäimillä ohjailtava pelihahmo, jonka tulee kerätä aarteita eri puolilta ikkunaa. Pelihahmoa jahtaa vihollinen ja peli päättyy, kun vihollinen saavuttaa pelihahmon.

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

Tehtävä: Pelialue_2

Luo projektiin uusi luokka, joka on nimeltään esim. Pelialue.



Tehtävä: Pelialue_3

Tee `Pelialue` -luokasta automaattisesti ajettava eli tee siitä `Runnable` ja kirjoita sille toistaiseksi tyhjä `run` -metodi.

```
public class Pelialue implements Runnable {

    @Override
    public void run() {

    }
}
```

Anna Kerailypeli -luokan main-metodissa komento käynnistää graafinen ohjelma:

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

Tehtävä: Pelialue_4

Ohjelmoi `Pelialue` -luokka niin, että se perii `JFrame` -luokan toiminnallisuuden (`extends`). Huomaa, että perintä pitää kirjoittaa ennen implements -määrettä.

Kirjoita `run` -metodiin, että tälle (`this`) Pelialue -ikkunalle annetaan nimi, koko, sulkemismekanismi (*`EXIT_ON_CLOSE`*) ja näkyvyys. Etsi komennot tarvittaessa aiemmasta Haltijapalvelu -monisteesta (GUI, osa 2). Testaa, että ikkuna aukeaa, kun ohjelma käynnistetään.

Tehtävä: Hahmo_1

Luo projektiin uusi `Hahmo` -luokka. Luo sille konstruktori, jossa se ottaa parametrina vastaan String -muotoisen tiedostonimen (hahmon kuva).

Ohjelmoi `Hahmo` -luokka niin, että se perii `JPanel` -luokan toiminnallisuuden (`extends`). Tällöin saamme hahmosta ikkunaan sijoitettavan komponentin, jota ei kuitenkaan voi klikata (ellei sille sitten ohjelmoisi MouseListener -rajapintaa).

Tehtävä: Hahmo_2

Etsi pelihahmolle kuva, jossa tausta on läpinäkyvä. Jos käytät Scratch:n kuvia, muuta kuva bittikarttatilaan ennen tallentamista.

Sijoita kuvatiedosto projektin kansioon. Netbeansissa normaalisti luodut projektit löytyvät käyttäjän kotihakemistosta NetBeansProjects -kansiosta.

Kuvat

Haltijapalvelussa opettelimme käyttämään ImageSprite -luokkaa. Sillä voi helposti piirtää komponentteja nappuloihin, mutta tarvitsemme tällä kertaa raskaampia työkaluja. Keräilypelissä ohjelman pitää pystyä testaamaan koskettavatko kaksi hahmoa toisiaan. Jos hahmot olisivat neliöitä olisi tämän testaus helpohkoa, mutta luonnollisempaa olisi testata menevätkö epämääräisen muotoiset kaksi kuvaa päällekkäin. Yksittäisten pikselien kyselyn mahdollistaa BufferedImage.

BufferedImage -olion luontiin tarvitaan File -muotoinen, kuvan sisältävä olio. Tiedoston osaa avata oikeaan muotoon ImageIO -työkalu (staattinen luokka, jota ei tarvitse erikseen projektiin luoda):

```
try {
    File tiedosto = new File("Tiedostonimi.png");
    BufferedImage kuva = ImageIO.read(tiedosto);
} catch (Exception e) {

}
```

Kuten kenties jo huomasitkin, kuvan avaus on sijoitettu try-catch -lohkoon. Kuvatiedoston puuttuminen on ilmeinen virhe, joka voi tapahtua, ja ohjelma pitää ohjelmoida niin, että se tietää mitä tehdä myös virheen sattuessa.

Mitä catch -lohkossa virheen sattuessa sitten tehdäänkään on ohjelmoijan päätettäessä. Sinne voi yksinkertaisimmillaan kirjoittaa virheviestin käyttäjälle tai kuvan löytymisen ja ohjelman toimimisen ollessa kriittisempää sinne voi ohjelmoida jonkin vaihtoehtoisen ikonin ohjelmaan lisäämisen.

Tehtävä: Hahmo_3

Luo Hahmo -luokkaan oliokohtainen muuttuja: `private BufferedImage kuva`. Oliokohtaiset muuttujat tehdään tyypillisesti konstruktorin yläpuolelle, heti `public class` -rivin alle.

Itse konstruktorissa luo oliomuuttuja `kuva` annetusta tiedostonimestä. (Älä siis luo uutta kuva-muuttujaa konstruktorissa vaan anna aiemmalle oliokohtaiselle muuttujalle vain arvo = -merkillä.)

Jos kuvan luomisessa tapahtuu virhe, ilmoita siitä jotenkin käyttäjälle. Kirjoita `System.out.println()` -viesti, tai pelaajalle huomattavasti paljon paremmin näkyvä:

```
JOptionPane.showMessageDialog(null, "Kirjoita tähän virheviesti.");
```

Override

JPanel itsessään ei sisällä mitään tapaa piirtää kuva, mutta se osaa kyllä piirtää itsensä. Projektissa nyt luotu Hahmo -luokka on JPanel (sen perittyään), joten myös se osaa piirtää itsensä tyhjänä. Perittyjä ominaisuuksia voi kuitenkin parannella.

Luokalla jo aiemmin olemassa olevan metodin voi kirjoittaa uudelleen parempana. Nämä merkitään yleensä koodiin rivillä: `@Override`

Näin tehtiin myös rajapintoja (ActionListener ym.) tehtäessä, mutta perittyjen ominaisuuksien tapauksessa uusi toteutus todella **korvaa** aiemman. Rajapintojen tapauksessa aiempaa toiminnallisuutta ei ollut.

Graafisilla elementeillä yksi korvattava ominaisuus on piirto. Alkuperäinen komennon määritelmä pitää kirjoittaa täsmälleen samanlaisena, jotta kääntäjä on mukana juonessa:

```
@Override
public void paintComponent(Graphics g) {

}
```

Metodi toimii jo näin kirjoitettuna, mutta se ei luonnollisestikaan tee mitään, sillä komentoja ei metodissa vielä ole. Ohjelmoija voi sijoittaa metodin sisälle kaikki haluamansa komennot, mutta käytännössä tämän komennon yhteydessä kannattaa keskittyä piirtämiseen.

Metodi antaa ohjelmoijalle käyttöön parametrina myös Graphics -olion. Ohjelmoijan ei tarvitse ihmetellä mistä se ohjelmaan saadaan luotua, vaan Java luo sen automaattisesti graafisia ikkunoita luodessa ja sen saa näin tarvittaessa käyttöön ohjelman sisältä. Näin yksinkertaisesti sen saa kylläkin vain perittyjä metodeja korvatessa.

Tehtävä: Hahmo_4

Kirjoita **`Hahmo`** -luokalle uusi `paintComponent` -metodi.

Metodin sisällä anna graafiselle g -työkalulle seuraava komento piirtää komponentin yläkulmaa (0, 0) kuva tietyn levyisenä ja korkuisena. Kuvan "huoltajaksi" laitetaan itse JPanel eli `this`.

```
g.drawImage(kuva, 0, 0, 50, 50, this);
```

Huomaa, että kuva ei vielä näy pelissä, koska pelialueelle ei olla lisätty vielä yhtään hahmoa.

Käsin asettelu

Tyypillisesti ohjelmat kannattaa luoda niin, että niillä on jokin valmisjärjestelijä, Layout, kuten aiemmin käyttämämme BorderLayout tai GridLayout. Tällöin ohjelma toimii varmimmin, minkä muotoiseksi käyttäjä sen nyt sitten venyttääkin.

Ikkunan komponentit on kuitenkin mahdollista asettaa paikoilleen myös käsin. Tällöin valmisjärjestelijästä pitää luopua (kaikilla komponenteilla on automaattisesti BorderLayout järjestelijä) ja komponentin luomisen lisäksi sen paikka ja koko pitää määrittää.

```
JPanel alueIlmanJarjestysta = new JPanel();
alueIlmanJarjestysta.setLayout(null);

JButton nappula = new JButton("Sekamelska");
nappula.setBounds(40, 80, 30, 100);
alueIlmanJarjestysta.add(nappula);
```

Tehtävä: Hahmo_5

Siirry takaisin `Pelialue` -luokkaan ja poista siltä järjestelijä `run` -metodissa. Lisää sille tämän jälkeen yksi Hahmo -olio sen kuvan kera, jonka aiemmin loit. (Kuvatiedoston nimi annetaan tekstisyötteenä.) Valitse hahmolle jokin satunnainen paikka ja määrittele kooksi sama kuin hahmossa piirretyllä kuvalla.

Testaa, että hahmo näkyy ikkunassa, kun ohjelma käynnistetään.

Tehtävä: Hahmo_6

Luo `Pelialue` -luokalle uudet oliokohtaiset (private) muuttujat: omaX, omaY, aarreX, aarreY, vihuX ja vihuY. Anna näille myös jotkin alkuarvot sen mukaan minkä kokoinen itse peli on. Aarteen satunnainen paikka koodataan myöhemmin.

Muuta pelihahmon setBounds -komennon syötteitä niin, että paikkaa ei määritellä siinä käsin (numeroa kirjoittamalla) vaan syötteeksi annetaan `omaX` ja `omaY`.

Tehtävä: Hahmo_7

Tee pelialueelle `private Hahmo pelihahmo` -muuttuja oliokohtaisiin muuttujiin ja korjaa run-metodia niin, että siellä ei määritellä uutta Hahmo -muuttujaa vaan `new Hahmo` tallennetaan oliokohtaiseen muuttujaan.

Tehtävä: Hahmo_8

Luo peliin myös hahmot: aarre ja vihu. Etsi siis niille kuvat ja lisää ne näkymään.

Näppäinkuuntelija

Useat graafisen käyttöliittymän tekstikomponentit ymmärtävät niihin kirjoitettuja tekstejä automaattisesti. JFramen tai JPanelin kaltaisilla komponenteilla ei kuitenkaan ole mitään syytä tulkita käyttäjän painamia näppäimiä millään lailla. Näppäin kuuntelijan voi kuitenkin aina lisätä tarpeen niin vaatiessa.

Luokka voi luvata toteuttavansa (implements) KeyListener -rajapinnan.

Tällöin luokalta tulee löytyä seuraavat metodit:

```
public void keyPressed(KeyEvent ke) {
}

public void keyReleased(KeyEvent ke) {
}

public void keyTyped(KeyEvent ke) {
}
```

Metodeista voi käyttää haluamiaan, loput voi jättää sisällöltään tyhjiksi. Pressed vastaa hetkeä, kun jokin näppäimistön näppäin painetaan pohjaan, Released sitä, kun näppäin vapautetaan. Typed vastaa näppäimen koko painallusta alusta loppuun. Typed kuuntelee vain näkyviä merkkejä, ei esimerkiksi nuolinäppäimiä.

Painetun näppäimen voi testata esimerkiksi näin:

```
public void keyPressed(KeyEvent ke) {
    if (ke.getKeyCode() == KeyEvent.VK_RIGHT) {

    }
}
```

Lisää näppäinten koodeja voi etsiä oraclen sivuilta googlettamalla “keyevent java api”. Huomaa, että KeyListener ei toimi, jos ikkunassa jollakin muulla komponentilla on *fokus*.

Tehtävä: Pelihahmo_1

Kirjoita `Pelialue` -luokan määritelmään, että se toteuttaa rajapinnan KeyListener (edellisestä rajapinnasta pilkulla erotettuna).

Kirjoita tai lisää automaattisesti luokalle edellä mainitut metodit `keyPressed`, `keyReleased` ja `keyTyped`. Poista metodeista `throw` -komennot, jos loit metodit automaattisesti.

Tehtävä: Pelihahmo_2

Luo Pelialue -luokalle vielä neljä uutta oliokohtaista, totuusarvoista muuttujaa. Nämä sitä varten, että peli voi muistaa mitkä nuolista ovat pohjassa ja mitkä eivät.

```
private boolean ylos = false, alas = false;
private boolean oikealle = false, vasemmalle = false;
```

Lisää tälle luokalle (`this`) näppäinkuuntelijaksi se itse (`this`) `run` -metodissa.

Tehtävä: Pelihahmo_3

Kirjoita keyPressed -metodiin neljä seuraavanlaista testiä (katso mallia yltä):

> “Jos painettu näppäin on tietty nuolinäppäin niin tätä vastaava oliomuuttuja asetetaan arvoon true.”

Eri nuolinäppäimet ovat: *VK_UP*, *VK_DOWN*, *VK_RIGHT* ja *VK_LEFT*.

Testaa, että testit toimivat ja kirjoita niihin hetkeksi vaikkapa `System.out.println("!")`.

Tee vastaava toiminnallisuus keyReleased -metodiin, mutta aseta siellä arvot pois päältä, eli arvoon `false`. Huom! Älä liikuttele vielä hahmoa, tehdään se kohta paremmin.

Ajastin

Java-projektiin voi lisätä `Timer` -olion. Tällainen ajastin osaa lähettää “hälytyksiä” tietyin aikavälein. Hälytyksiä tulee olla vastaanottamassa `ActionListener` -kuuntelija, johon on voitu ohjelmoida mitä ajastimen hälyttäessä tulee tehdä. Seuraavassa esimerkissä ajastin luodaan, kun peli käynnistyy, ja 100 ms välein se kirjoittaa tekstin: 100 ms meni juuri.

```
import javax.swing.Timer;

public class Peli implements ActionListener {
    public void run() {
        Timer ajastin = new Timer(100, this);
        ajastin.setRepeats(true);
        ajastin.start();
    }

    public void actionPerformed(ActionEvent ae) {
        System.out.println("100 ms meni juuri.");
    }
}
```

`Timer` -oliota tehdessä sille tulee antaa kellon lyöntitiheys millisekunneissa sekä ActionListener, joka ottaa vastaan viestit ajan kulumisesta. Tässä esimerkissä luokka itsessään on ActionListener. Ajastin luontinsa jälkeen voidaan komentaa esimerkiksi tekemään toistuvia (repeat = toista) hälytyksiä. Ajastimen ollessa valmis se käynnistetään (start).

Huomaa, että Javalle on tehty useita eri Timer -toteutuksia. Tässä materiaalissa käytämme `javax.swing.` -osoitteesta löytyvää ajastinta.

Tehtävä: Pelihahmo_4

Tee `Pelialue` -luokasta myös `ActionListener` eli aseta se implementoimaan kyseinen rajapinta luokan alussa ja tee sille `actionPerformed` -metodi.

Tehtävä: Pelihahmo_5

Luo `Timer` -olio `Pelialue` -luokan `run` -metodissa, aseta sen hälytykset toistuviksi ja käynnistä ajastin. Testaa esimerkiksi `System.out.println("!")` -viestillä (`actionPerformed` -metodissa), että ajastin varmasti toimii.

Tehtävä: Pelihahmo_6

Kirjoita `actionPerformed` -metodiin, että jos `oikealle == true`, niin `omaX` -arvoa kasvatetaan. Ohjelmoi metodin loppuun myös, että tuosta testistä huolimatta **jokaisen** hahmon `setBounds` asetetaan uudelleen. Tällöin mm. `omaX` -arvo päivittyy peli-ikkunassa.

Testaa, että oikealle liikkuminen toimii nyt. Voit nopeuttaa hahmoa muuttamalla `omaX` -arvoa kerralla enemmän tai lyhentämällä ajastimen aika-askelta. Hidastaa voit luonnollisesti päinvastoin.

Tehtävä: Pelihahmo_7

Toteuta `actionPerformed` -metodiin myös loput suunnat yksittäisinä `if` -lauseina. Laita nekin ennen `setBounds` -komentoja.

Testaa, että pelihahmo liikkuu halutusti nuolinäppäimillä. Mihin suuntaan y-akseli kasvaa Javan ikkunoissa?

Tehtävä: Koskettaako_1

Scratchissa kahden hahmon kosketus oli helppo testata. Javassa näin …ei ole. Jos hahmot olisivat täydelliset neliöt tai ympyrät voisi kosketusta testata hahmon sijainnin ja koon/säteen mukaan. Kuvien kanssa pitää mennä yksittäisiin pikseleihin asti. Hahmon voi komentaa testaamaan onko se jossain tietyssä pikselissä näkyvä vai läpinäkyvä.

Kirjoita `Hahmo` -luokkaan uusi metodi:

```
public boolean oletkoTassa(int ikkunaX, int ikkunaY) {
    return true;
}
```

Tehtävä: Koskettaako_2

Paikan testauksessa tulee ottaa huomioon, että kysyttävä koordinaatti on luodun ikkunan vasemman yläkulman suhteen, se ei vastaa hahmon kuvan pikseleitä, jollei hahmo sitten ole aivan vasemmassa yläkulmassa. Hahmo voi selvittää kuitenkin oman sijaintinsa `getBounds` -komennon avulla. Komento palauttaa Rectangle -olion, jolta edelleen voidaan kysyä sen sijainti (ja koko).

Piirrä ikkuna ja satunnaiseen kohtaan sijoitettu hahmo paperille ja pohdi miksi koordinaatti hahmon suhteen voidaan laskea seuraavasti:

```
Rectangle hahmoruutu = this.getBounds();
double xKuvassa = ikkunaX - hahmoruutu.getX();
double yKuvassa = ikkunaY - hahmoruutu.getY();
```

Rectangle palauttaa sijaintinsa liukulukuna, joten talletetaan koordinaatti tällä kertaa `double` -arvoisena.

Kirjoita edellä olevat rivit `oletkoTassa` -metodiin ennen `return` -riviä, sillä kysyttävän pikselin koordinaatti pitää varmasti laskea ennen kuin vastaus kysymykseen “oletko tässä” voidaan antaa.

Tehtävä: Koskettaako_3

Ohjelmoi koordinaattien laskemisen alle seuraavanlainen toiminnallisuus Javaksi muunnettuna:

```
jos ( xKuvassa tai yKuvassa on < 0 tai
jompikumpi on ≥ kuvan leveys tai korkeus koordinaatista riippuen ) {
    return false;
}
```

Eli käytännössä, jos kysyttävä koordinaatti on kuvan ulkopuolella niin kuva ei varmasti ole siinä näkyvissä. Huomaa, että tarvitset tähän neljä testiä, jotka voi erotella saman if -lauseen sisällä `||` merkeillä, jotka Javaksi merkitsevät: “tai”

Kuvan leveyden voit *kovakoodata* eli katsoa aiempaa minkä levyiseksi/korkuiseksi kuva tehtiin ja kopioida saman numeron tähän kuvan leveydeksi ja korkeudeksi.

Jos `xKuvassa` ja `yKuvassa` sattuivat kuvan kohdalle, palauttaa metodi viimeisellä rivillään yhä arvon `true`. Toistaiseksi siis oletamme hahmon olevan jokaisessa kuvan kohdalle sattuvassa pikselissä.

Bonus: *Kovakoodatessa* ohjelmaa voi olla myöhemmin työlästä muuttaa sillä sama arvo pitää tällöin muistaa ja jaksaa kopioida jokaiseen tarvittuun paikkaan. Vaihtoehtoisesti esim. kuvan koolle voisi tehdä oman oliokohtaisen muuttujan luokan alussa:

```
private int koko = 50;
```

ja nimen `koko` voisi kirjoittaa koodissa tarvittuihin paikkoihin. Tällöin kokoa muutettaessa vain yhteen paikkaan muutoksen tekeminen riittäisi.

Värit ja kuvat

Väri on Javan normaaliin tapaan sekin olio. Värin voi muodostaa mm. seuraavilla tavoilla:

```
Color sininen = Color.BLUE;
Color violetti = new Color(100, 0, 100);
```

mutta kun kuvilta kysytään väriä ne palauttavat yhden ainoan numeron. Värin voi luoda myös sillä kunhan muistaa kertoa värille, että kuvasta haettu väri sisältää myös läpinäkyvyyden arvon:

```
//näitä rivejä täytyy edeltää kuvan luonti jossain kohtaa koodia
int variarvo = kuva.getRGB(x, y);
Color kuvanVari = new Color(variarvo, true);
```

Väriltä voi vuorostaan kysyä sen punaisen, vihreän, sinisen ja näkyvyyden (alpha) arvoja:

```
int lapinakyvyys = kuvanVari.getAlpha();
```

Tehtävä: Koskettaako_4

Kirjoita edellisen tehtävän kuvan rajoja tarkastelevan testin jälkeen vielä, että kuvasta todella testataan yksittäisen pikselin näkyvyys. Kuva voi toki olla myös erikokoinen kuin minkä kokoiseksi se on piirretty, joten koordinaatit pitää skaalata vielä sen suhteen. Jotta skaalaus olisi tarkka tulee lasku laskea liukuluvuilla. Javan voi pakottaa tähän, kun laskuun lisätään yksi liukuluku, esim. ylimääräinen 1.0 :lla kertominen.

```
int koko = 50;
xKuvassa = 1.0 * kuva.getWidth() / koko * xKuvassa;
yKuvassa = 1.0 * kuva.getHeight() / koko * yKuvassa;
```

Aseta kooksi oman pelin hahmoasi vastaava koko. Tässä vaiheessa on helpointa, jos kaikki pelin hahmot ovat samankokoisia.

Tämän jälkeen koodaa (Javaksi):

```
Color pikseli = new Color(kuva.getRGB((int)xKuvassa, (int)yKuvassa), true);
jos ( pikseli.getAlpha() == 0 ) {
    palauta false
}
muuten {
    palauta true
}
```

Tässä koodissa koordinaatit tulee jälleen castata eli muuntaa kokonaisluvuiksi, jotta ne toimivat kuvan liukuluvuttomiin pikseleihin. Poista alta tarpeeton `return true;` -komento.

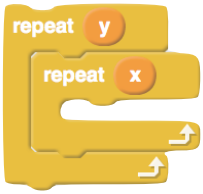
Tehtävä: Koskettaako_5

Kirjoita **`Pelialue`** -luokkaan uusi metodi: `private void` **`testaaOsumat()`**

Lisää `testaaOsumat();` -metodikutsu **`actionPerformed`** -metodiin ennen `setBounds()` -komentoja, mutta hahmon paikan muuton jälkeen.

Tehtävä: Koskettaako_6

Kirjoita testaaOsumat -metodiin tuplasilmukka, jossa käydään läpi kaikki ikkunan y- ja x-arvot läpi `for` -silmukoissa. y- ja x- arvot alkavat nollasta ja loppuvat yhtä vajaa ikkunan korkeuteen/leveyteen. Kysy ohjelmastasi miten korkea ja leveä `Pelialue` -ikkunasi on komennoilla `this.getHeight()` ja `this.getWidth()`.



Satunnaisuus

Viime kerrasta, kun arvoit Javassa satunnaisia lukuja, voi olla hetki. Tässä muistutus.

```
Random noppa = new Random();
int ekaArvo = noppa.nextInt(10);
int tokaArvo = noppa.nextInt(6);
```

Javassa pitää siis ensin luoda noppa -olio ja siltä voi sitten kysyä jonkin satunnaisen numeron nollasta annettuun ylärajaan. Noppaa voi toki heittää useamman kerran ja sen sivujen määrää voi muuttaa, koska miksi ei?

Nopalta voi oikeastaan kysyä myös muita satunnaisia asioita, voit tutustua niihin googlettamalla “random java api”. Oraclen sivuilla “Method Summary” otsikon alta löytyy yllä olevan komennon lisäksi muitakin kiinnostavia komentoja.

Tehtävä: Koskettaako_7

Silmukoiden sisällä, eli jokaisen ikkunan pikselin kohdalla, kysy koskettavatko pelihahmo ja aarre toisiaan. Javaksi muotoiltuna kysytään siis käytännössä tunnustavatko molemmat hahmot olevansa yhtä aikaa saman pikselin kohdalla:

```
if ( pelihahmo.oletkoTassa(x, y) && aarre.oletkoTassa(x, y) ) {

}
```

Jos hahmot sattuivat kohdakkain, kirjoita testin sisään, että aarteelle arvotaan uusi paikka. Millä välillä aarteen koordinaatit voivat vaihdella?

Aarteiden pitäisi nyt olla kerättävissä, kun testaat peliä.

Perintä

Olemme aiemmin luoneet luokkia, jotka ovat perineet valmisluokkia kuten JFrame tai JButton. Mikään ei kuitenkaan estä luomasta luokkaa, joka perii toisen projektiin luodun luokan toiminnot.

Joskus on kuitenkin myös tarpeen käyttää perityn luokan metodia. Tämä on mahdollista perivästä luokasta. Perityn luokan konstruktoria voi käyttää perivän luokan konstruktorin ensimmäisellä rivillä:

```
super();
```

Perityn luokan metodeja voi käyttää: `super.komennonNimi();`

Tehtävä: Aarresäkki_1

Luo projektiin uusi luokka, joka on nimeltään esimerkiksi Pelihahmo.

Aseta Pelihahmo perimään **Hahmo** -luokan toiminnallisuus, sillä pelihahmo on myös hahmo. Näin kuvien määrittelyjä tai pikselien tarkistuksia ei tarvitse tehdä erikseen Pelihahmolle.

Perivällä luokalla tulee olla vähintään yksi samanlainen konstruktori kuin perityllä luokalla. Tee siis Pelihahmolle:

```
public Pelihahmo(String tiedostonimi) {
    super(tiedostonimi);
}
```

Tehtävä: Aarresäkki_2

Luo **Pelihahmo** -luokalle oliokohtainen muuttuja: `private int pisteet = 0;`

Luo luokalle myös uusi metodi `public void annaPiste()`, jossa pisteitä kasvatetaan yhdellä.

Tehtävä: Aarresäkki_3

Kirjoita **Pelihahmo** -luokalle uusi paintComponent -metodi (katso mallia metodin muotoiluun Hahmosta). Käske hahmoa heti metodin ensimmäisenä komentona käyttää `super.paintComponent(g);` -metodia. Tällöin kuva piirtyy ilman, että Pelihahmoon sitä tarkemmin kirjoitetaan.

Piirrä pistemäärä esimerkiksi hahmon vasempaan yläkulmaan:

```
g.setColor(Color.BLACK);
g.drawString("" + pisteet, 1, 10);
```

Pisteiden eteen pitää kirjoittaa tyhjä `""`, jotta pistemäärä olisi `drawString` -komennon vaatimassa merkkijono-muodossa.

Tehtävä: Aarresäkki_4

Edelliset muokkaukset **Pelihahmo** -luokkaan eivät luonnollisestikaan toimi, jos mikään pelin hahmoista ei ole `Pelihahmo`.

Muokkaa **Pelialue** -luokan oliomuuttujia niin, että pelihahmo onkin `Pelihahmo` -tyyppinen. Korjaa samaten **run** -metodiin, että `pelihahmo` -muuttujaan ei talleteta uutta `Hahmo` -oliota vaan uusi `Pelihahmo` -olio.

Testaa, että pelihahmon yläkulmaan nyt ilmestyy nolla.

Tehtävä: Aarresäkki_5

Lisää `testaaOsumat` -metodiin `pelihahmo` -oliolle komento antaa piste, kun pelihahmo koskee aarteeseen.

Testaa peliä ja korjaa ilmenevä ongelma lisäämällä Scratchin “pysäytä tämä skripti” -komentoa vastaava `return;` sopivaan kohtaan ohjelmakoodia.

Tehtävä: Viholinen_1

Lisää `testaaOsumat` -metodiin pelihahmon ja aarteen kosketuksen testaamisen jälkeen toinen testi, jossa testaat pelihahmon ja vihun osumista kohdakkain. Jos kohtaaminen tapahtuu, voit sulkea pelin komennolla `System.exit(0);`

Halutessasi voit myös avata pelaajalle pop up -ikkunan ennen sulkemista. Kun ensimmäiseksi parametriksi annetaan itse ikkuna, ei pelaaja voi enää aktivoida sitä ja pelata ennen viestin OK -nappulan klikkaamista.

```
JOptionPane.showMessageDialog(this, "Vihollinen sai sinut kiinni!");
```

Tehtävä: Viholinen_2

Luo **`Pelialue`** -luokkaan uusi metodi:

```
private void liikutaVihollista() {

}
```

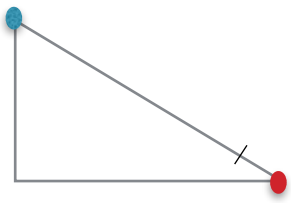
Lisää `liikutaVihollista();` -metodikutsu **`actionPerformed`** -metodiin juuri ennen osumien testausta.

Tehtävä: Viholinen_3

Vihollisen olisi tarkoitus aina liikkua suoraan kohti pelihahmoa. Scratchissa pystyi osoittamaan kohti jotain ja liikkua sitten nenän osoittamaan suuntaan. Kumpaakaan ei voi tehdä Javassa yhdellä komennolla.

Tässä vaiheessa siis perehdymme hieman matematiikkaan, jotta saamme selville oikean reitin. Jos matematiikka ei kiinnosta, voit omantuntosi mukaan napata vain laskukaavat vaikkakin silloin seuraavan oman pelin, johon ei saa valmiita kaavoja, tekeminen voi olla haastavampaa. Voi toki myös odottaa, että näitä käsitellään koulussa ensin.

1. Kuvittele tilanne, jossa pelihahmo on alla olevan kolmion vasemmassa yläkulmassa ja vihu oikeassa alakulmassa. Vihu voi edetä esim. 10 askelta poikkiviivan kohdalle:



2. x- ja y-suuntaisten etäisyyksien neliöiden summa on Pythagoraan lauseen mukaan sama kuin hahmoja suoraan yhdistävän etäisyyden neliö. Hahmojen välinen etäisyys on siis:

```
double etaisyys = Math.sqrt(Math.pow(vihuX-omaX,2) + Math.pow(vihuY-omaY,2));
```

3. Esimerkiksi 10 askeleen matka on tästä etäisyydestä `10/etaisyys` verran. Voimme siis skaalata liikuttava matkan sillä:

```
vihuX -= (int) (10/etaisyys * (vihuX-omaX));
vihuY -= (int) (10/etaisyys * (vihuY-omaY));
```

Testaa, että vihollinen seuraa nyt pelihahmoa ja aseta sen nopeus peliin sopivaksi.

Taustakuva

Javassa ikkunan elementeille on helppo asettaa taustaväri, taustakuva on asia erikseen. Java ei sinänsä tunne käsitettä “ikkunan taustakuva”, mutta Java-ohjelmaan voi luoda ikkunaelementin, jossa näkyy kuva. Käytännössä voit siis käyttää ikkunan kokoista JPanelia tai JLabelia, johon on asetettu näkymään kuva.

JLabel
Plussat: kuvan voi liittää valmiilla komennolla
Miinukset: JLabel ei ole “Container” (~astia) eli sen alueelle ei voi lisätä esim. hahmoja

JPanel
Plussat: JPanel on “Container” eli se on luotu sisältämään juurikin esim. muita hahmoja
Miinukset: kuvaa ei voi liittää yhdellä komennolla vaan JPanelin paintComponent pitää kirjoittaa uusiksi

> Tehtävä: Viimeistely_1
>
> Lisätään peliin vielä taustakuva. Tässä vaiheessa voi huomata, että pääsemme kuvan lisäämisessä harvinaisen helpolla, sillä myös tausta voi olla hahmo ja hahmo osaa jo piirtää itsensä ikkunaan.
>
> Etsi peliin taustakuva ja lisää se projektin kansioon.
>
> Lisää `Pelialue` -luokan `run` -metodiin uuden Hahmo -olion ikkunaan lisääminen. Anna olion parametriksi taustakuvan tiedostonimi. Aseta taustahahmo kohtaan (0, 0) ja koko ikkunan levyiseksi ja korkuiseksi. Pitääkö taustahahmo lisätä ennen vai jälkeen muita hahmoja, että se tulee taaimmaiseksi?

> Tehtävä: Viimeistely_2
>
> Taustakuva ilmestyi luultavasti varsin pienenä.
>
> Hahmon koko on toistaiseksi vain kovakoodattu sen paintComponent -metodiin. Hahmo voi kuitenkin kysyä sielläkin minkä kokoiseksi se on luotu. Kirjoita piirrettävän kuvan kooksi numeroiden sijasta:
>
> `(int)this.getBounds().getWidth()` ja `(int)this.getBounds().getHeight()`

Javakomentojen mekka

Erityisesti Javan graafiseen käyttöliittymään löytyy niin paljon käytettävissä olevia elementtejä, että niitä ei vain voi muistaa. Javasta löytyy onneksi hyvin paljon tietoa verkosta.

Oracle Help Center ylläpitää javakomentojen kattavaa listaa, jossa listataan kaikille Javan luokille ja rajapinnoille niiden metodien ja konstruktorien yksityiskohdat, niiden perimät luokat ja niiden toteuttamat rajapinnat sekä tavat käyttää niitä. Yksinkertaisin tapa löytää apua on hakea verkosta esim. “jbutton java api”, vaihdat vain ensimmäisen sanan tilalle sen luokan nimen mistä haluat tietoa.

Entä jos sitten ei tiedäkään mistä luokasta haluaa tietoa? Verkosta löytyy tällöinkin paljon tietoa englanniksi. Parhaimpien hakutulosten joukossa on usein stackoverflow.com tuloksia, missä voi sekä kysyä apua että tarjota sitä. Jo kysyttyjen asioiden joukosta löytyy usein apu omaankin ongelmaan.