

Javan GUI Scratchaajalle

Graafisen ikkunan luonti

Aiemmin ohjelmoimme luokat ja metodit itse. Graafisia ohjelmia tehdessä kaikkea ei kannata tehdä itse, osan voi ottaa valmiina. Näitä "valmishahmoja", eli javaksi **olioita**, käytetään samoin kuin muitakin tekemiämme olioita. Javassa ikkuna itsessään on myös olio. **Vertaa:**

Olion luonti:

```
Kissa aatu = new Kissa();
```

Ikkunan luonti:

```
JFrame ikkuna = new JFrame();
```

eli yhä kirjoitetaan: `millainen mikä = mitä();`

Ikkunaa luodessa pitää myös:

- kertoa ikkunan yläpalkkiin nimi
`.setTitle("Haltijapalvelu");`
- kertoa koko
`.setSize(400,400);`
- kertoa ohje sulkemiseen
`.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- kertoa, että ikkuna on näkyvässä!
`.setVisible(true);`
- valinnainen: myös ikkunan taustaväriä voi kertoa
`.getContentPane().setBackground(Color.BLACK);`

```
set = asettaa
title = otsikko
size = koko
default = oletus
close = sulkea
operation = toimenpide
exit = poistua
visible = näkyvä
true = totta
get = saada
content pane = sisältöruutu
```

Tehtävä: Haltija_1

Luo uusi java-projekti. Nimi voi olla vaikka "Haltijapalvelu". Anna NetBeansin luoda *Main Class* sen nimisenä kuin se ehdottaa. Pelissä tullaan tekemään haltijoiden tilauspalvelu.

Tehtävä: Haltija_2

Luo syntyneen luokan (=tiedoston) main-metodissa ensin uusi JFrame-olio eli ikkuna.

NetBeans alleviivaa punaisella edellisen rivin, koska se ei tiedä miten JFrame tehdään. Importtaa JFrame:n teko-ohje tiedoston alussa, ennen `public class Haltijapalvelu` -riviä: `import javax.swing.JFrame;`

Ikkunan luonnin jälkeen anna sille yllä ohjeessa mainitut komennot, jotta se tietää millainen sen tulisi olla. Huomaa, että komennon kirjoittaminen ei nyt riitä vaan se pitää antaa nimenomaan ikkunalle, siis tyyliin näin:

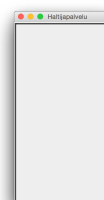
```
ikkuna.setTitle("Ikkunan nimi");
```

Käynnistä peli ja katso, että ikkuna myös syntyy.

Huom! Sulje ikkuna, jotta voit testata peliä aina myös uudelleen.

Tehtävä: Haltija_3

Muuta ikkunan koon asettavaa komentoa, jotta saat vieressä näkyvän muotoisen ikkunan.



Ikkunakomponenttien lisäys

Ikkunaan voi lisätä sisältöjä, kuten Scratchissa esiintymislavalle voi lisätä hahmoja. Javassa esiintymislavalle pitää kylläkin antaa myös komento, että sen pitää *ottaa hahmo vastaan* lavalle.

Voisimme tehdä vaikka JLabelin, joka lisätään (add=lisätä) lavalle:

```
JLabel etiketti = new JLabel("Lava on huollossa.");
ikkuna.add(etiketti);
```

Jotta ohjelma ymmärtäisi mistä JLabelista ohjelmoija nyt puhuu, pitää tiedoston alussa myös käydä "lainaamassa" JLabelin (tai muiden esiteltävien luokkien) teko-ohje:

```
import javax.swing.JLabel;
```

JLabel on etiketti, jonka voi "liimata" ikkunaan. Ikkunaan voi lisätä myös muita komponentteja:

Etiketti	JLabel("Sitruunapippuri");
Nappula	JButton("Klikkaa tästä.");
Lyhyt tekstikenttä	JTextField();
Pitkä tekstikenttä	JTextArea();

Tehtävä: Haltija_4

- 1) Kokeile vuorotellen miltä mikin komponentti näyttää ikkunassa. Luo siis ensin komponentti ja käske sitten ikkuna ottamaan se vastaan näkymään. Jos ikkunaan ei tule mitään, raahaa hiirellä se eri kokoiseksi. > Java piirtää sen uudelleen.
- 2) Mitä tapahtuu, jos koitat laittaa ikkunaan kaksi komponenttia yhtä aikaa?
- 3) Miten lyhyt ja pitkä tekstikenttä eroavat toisistaan? Vinkki: Koita kirjoittaa niihin paljon tekstiä.

Kokonaisen kirjaston lainaaminen

Äsken jouduit koko ajan pyytämään ohjelmallesi käyttöön eri luokkia, kuten JLabel, JTextField, ... Jos haluat kerralla pyytää käyttöön vaikka kaikki javan graafiseen käyttöliittymään liittyvät javax-komponentit, voit kirjoittaa tähden:

```
import javax.swing.*;
```

Muista, että kaikki ohjelman käyttöön otettavat luokat pitää importata tiedoston alussa.

Kuvat graafisessa käyttöliittymässä

Scratchissa hahmolla on kuva automaattisesti ja sitä voi vaihtaa asusteissa. Javassa asusteellekin pitää tehdä oma hahmo. Tämän hahmon nimi on ImageIcon (=kuvaikoni).

```
ImageIcon kuva = new ImageIcon("Tiedostonimi.png");
```

Yksi komponentti, joka ymmärtää kuvia on JButton. Kuvan luonnin lisäksi esim. nappulalle pitää myös antaa komento asettaa (=set) kuva käyttöön.

```
komennettavaHahmo.setIcon(kuva);
```

Tehtävä: Haltija_5

Poista alkuun muut komponentit ikkunasta ja lisää nyt vain yksi JButton, jossa voi lukea vaikka "Tilaa haltija". Anna JButton-oliolle fiksu nimi!

Pikaisen testin jälkeen lisää nappulalle kuva.

- Tee esim. Scratchissa kuva hahmon asusteissa.
- Muuta kuva lopuksi bittikarttamuotoon oikeasta alakulmasta.
- Klikkaa hahmoa hiiren oikealla painikkeella ja valitse "tallenna paikalliseen tiedostoon". Jos oikealla painikkeella klikkaus ei toimi, paina Shift-näppäin pohjaan ja klikkaa vasemmalla painikkeella.
- Tallenna kuva NetBeansProjects -kansioon Haltijapalvelu-kansioon. (Ubuntun vieraskäyttäjäksi kirjautuneena se löytyy guest-kansiosta.)



Palaa ohjelmoinnin puolelle. Luo ImageIcon -hahmo ja komenna JButton -nappulaa asettamaan se kuvaksi.

Muuta ikkunan koko uuteen kuvaan sopivaksi ohjelman koodissa.

Bonus: Haltija_6

Saatoit äsken luoda yhdellä rivillä kuva-hahmon ja käskeä nappulan toisella rivillä ottamaan se vastaan. Tämä ei ole välttämätöntä. Jos kuva vain kerran annetaan nappulalle ja ohjelma ei sen jälkeen välitä kuvasta, voidaan kuva luoda suoraan nappulalle. Kuvalle ei tarvitse tällöin antaa edes nimeä.

Luo siis kuva `setIcon`-komennon sulkujen sisällä.

Automaattikäynnistys

Nyt on kyllä tyhmää, jos ikkunan kokoa pitää aina muuttaa, että se näkyy. Tehdään ohjelmastamme automaattikäynnistettävä eli javaksi `Runnable`. Ensin meidän pitää tutustua rajapintoihin.

Rajapinnat

Rajapinnat ovat lupauksia siitä mitä joku hahmo voi tehdä. Esimerkiksi Orava voi luvata olevansa *keräilevä* tai ohjaaja voi luvata olla *avustava*. Mikään ei toki estä Oravaa lupaamasta moniakin asioita. Se voi olla keräilevän lisäksi avustava.

Lupaukset pitää kuitenkin määrittää tietokonetarkasti:

- minkä nimisen komennon hahmo ymmärtää
- mitä taustatietoja komentoon on määritelty
- mitä komento palauttaa
- voiko kuka vain antaa komennon vai vain hahmo itse, eli: `public` / `private`

Haltijapalvelussamme haluamme seuraavaksi tehdä Ikkunan, joka lupaa olevansa *ajettava* eli `Runnable`. Tämä luvataan näin:

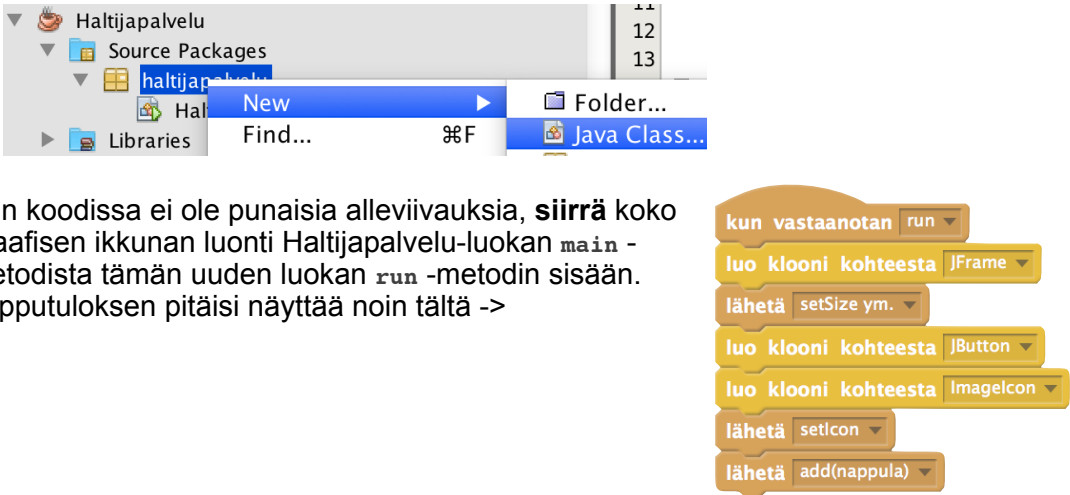
```
public class Ikkuna implements Runnable {  
  
}
```

Tyhjä lupaus ei tietokoneelle riitä. `Runnable`-lupaukseen on määritelty, että lupauksen tekevän luokan pitää sisältää `run` -komento:

```
public void run() {  
  
}
```

Mitä komennossa tehdään ei `run` -komentoa kiinnosta, mutta me haluamme luoda siinä ikkunan.

Tehtävä: Haltija_7
Luo projektiin uusi "Ikkuna" -luokka, joka lupaa olevansa `Runnable`. Tee sille `run` -metodi.



Kun koodissa ei ole punaisia alleviivauksia, **siirrä** koko graafisen ikkunan luonti `Haltijapalvelu`-luokan `main` -metodista tämän uuden luokan `run` -metodin sisään. Lopputuloksen pitäisi näyttää noin tältä ->

Nyt ohjelma ei toimi, koska `Haltijapalvelu` ei käynnistä `Ikkuna`-luokkaa. Jatketaan siis materiaalissa eteenpäin.

SwingUtilities

`SwingUtilities` on luokka, joka osaa mm. käynnistää luotettavasti graafisia käyttöliittymiä, jotka sisältävät `Runnable` -lupauksen `run` -metodin.

Tyypillisesti olemme tehneet aina uuden (=new) "kloonin" eli olion, kun olemme tehneet esim. `JFrame`ja, `JButton`eita ja `ImageIconeita`. `SwingUtilities` on kuitenkin *staatinen* luokka eli se lymyilee aina lähistöllä ja odottaa, jos sille tulisi komentoja. Sille voi siis antaa komentoja ihan vain mainitsemalla ensin sen nimen: `SwingUtilities`.

Kuvitellaan, että olemme luoneet `Esiintymislava` -luokan, joka on `Runnable`:

```
public class Esiintymislava implements Runnable {  
    //tänne on ohjelmoitu mm. run-metodi  
}
```

Komento, jolla saamme `SwingUtilities` luokan käynnistämään käyttöliittymämme (main-metodissa):

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

Tehtävä: Haltija_8
Laita `Haltijapalvelu`-luokan `main`-metodin sisään, eli Scratchin vihreän lipun alle, `invokeLater`-komento. Laita käynnistettäväksi sulkujen sisään uusi `Ikkuna`-kloonin.
Testaa, että `Haltijapalvelu` käynnistyy.

Kuuntelijat = Listener

Nappulan löytyminen ei riitä. Sitä voi painella, mutta mitään ei tapahdu. Javassa meidän pitää lisätä nappulalle kuuntelija, jotta ohjelma ymmärtää, että nappulaa oikeasti pitää tarkkailla.

Kuuntelijoita on moneen eri tarkoitukseen, tässä muutama:

<code>ActionListener</code>	Kuuntelee mm. nappuloiden tapahtumia (=action).
<code>MouseListener</code>	Kuuntelee hiiren klikkauksia ja hiiren saapumista/lähtemistä esim. nappulan alueelta.
<code>MouseMotionListener</code>	Kuuntelee hiiren liikettä (=motion).
<code>MouseWheelListener</code>	Kuuntelee hiiren rullan (=wheel) pyörimistä.
<code>KeyListener</code>	Kuuntelee näppäimistöä. (JTextField tai JTextArea eivät tarvitse tätä.)
<code>ComponentListener</code>	Kuuntelee mm. ikkunan siirtoja.

Kuuntelijoiden käyttö

Kaikki äsken mainitut kuuntelijat ovat oikeasti vain rajapintoja, eli listoja lupauksista. Jos haluamme oikeasti kuunnella nappulan klikkauksia, meidän tulee tehdä luokka, joka osaa tehdä kaikki ActionListenerin määrittämät lupaukset.

1) Uuden luokan tulee luvata olevansa ActionListener:

```
import java.awt.event.*;

public class NappulaKuuntelija implements ActionListener {

}
```

2) Uuden luokan tulee myös pitää lupauksensa. ActionListener-lupauksiin kuuluu tällä kertaa vain yksi metodi. Metodien sisällä voi sitten halutessaan suorittaa komentoja.

```
//nappulan klikkaus
public void actionPerformed(ActionEvent ae) {

}
```

Bonus-tietoa: ActionEvent

`actionPerformed` (action = toiminto, perform = suorittaa, näytellä) komennon sisällä voi esimerkiksi tulostaa tekstiä `System.out.println("Minua klikattiin.");` tai luoda uuden kissan `new Kissa("Mauku")`.

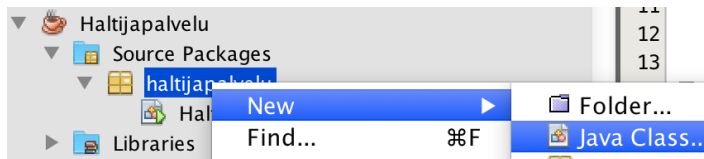
Klikkauksesta syntyvä tapahtuma voi myös ensin kysyä lisää tietoa tapahtumasta. Yllä olevassa esimerkissä `actionPerformed`-metodi saa taustatiedokseen `ActionEvent` -olion nimeltään `ae`. (event=tapahtuma) Siltä voi esimerkiksi kysyä klikkauksen parametreja:

```
System.out.println( ae.paramString() );
```

Erilaiset kuuntelijat antavat käytettäväksi erilaisia tapahtumia. Esimerkiksi `MouseListener` -rajapinnan metodit antavat taustatiedoksi `MouseEvent` -olion. `MouseEvent` -olionta voi pyytää mm. hiiren sijaintia suhteessa kuunneltavaan komponenttiin tai onko vaikka Shift-painettuna, jolloin hiiren klikkaus voi tarkoittaa eri asiaa.

Tehtävä: Asiakaspalvelu_1

Tee projektiin uusi luokka. Sen nimi voi olla vaikka TilausKuuntelija.

**Tehtävä: Asiakaspalvelu_2**

Kirjoita luokan määrittelmään (`public class` -rivi), että se lupaa olla ActionListener.

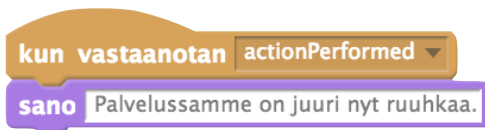
Netbeans alleviivaa nyt luokan nimen, koska se ei täytä lupaustaan. Klikkaa hehkulamppua rivin vasemmassa laidassa ja valitse "Implement all abstract methods".

Poista Netbeansin automaattisesti kirjoittamasta metodista `throw new` -rivi.

Tehtävä: Asiakaspalvelu_3

Jos kuuntelijan metodit ovat tyhjä, ne eivät tietenkään tee mitään vaikka jotakin tapahtumia kuultaisiinkin.

Kirjoita `actionPerformed` -metodiin `{ }` -sulkujen väliin haluamasi viestin tulostus.

**Bonus-tietoa: @Override**

Kun luokkaan kirjoitetaan metodeita, jotka täyttävät rajapinnan määrittämiä lupauksia tai korvaavat jonkun perityn luokan metodin (tästä lisää myöhemmin), niiden alkuun on tapana kirjoittaa `@Override` -rivi. Sen tarkoitus on muistuttaa ohjelmoijalle, että alla oleva metodi täyttää jonkin lupauksen tai korvaa jonkin aiemman luodun metodin.

Bonus-tietoa: throw new Exception

Äskeisen tehtävän aikana Netbeans loi tekemiinsä metodeihin `throw new UnsupportedOperationException` -rivejä. Suora käänös olisi, että java tällöin heittää uuden tukemattoman toimenpiteen poikkeuksen. Ohjelmaa suoritettaessa välillä syntyy virheitä vahingossa, mutta niitä voi myös tarkoituksella luoda, tai siis *heittää* kuin kapuloita rattaisiin. Aiemmat ohjelmamme ovat usein kaatuneet tällaisiin virheisiin, mutta graafiset käyttöliittymät voivat usein selvitä niistä. Tällöin ne voivat kylläkin olla näkyvissä vaikka jokin niiden sisällä olisikin jo pahasti rikki.

Halutessasi voit kokeilla heittää seuraavaa virhettä main-metodin `invokeLater`-komennon jälkeisellä rivillä:

```
throw new NullPointerException("Ähäkutti.");
```

Virheitä voi luoda, mutta ohjelman voi myös ohjelmoida varautumaan niihin. Tähän käytetään `try-catch` -menetelmää, eli ensin yritetään jotain (`try`) ja sitten otetaan koppi (`catch`), jos virhe syntyy. Virheillä on monia erilaisia nimiä, jotta koppia otettaessa erilaiset virheet voidaan ratkaista erilailla.

Kuuntelijan lisääminen ohjelmaan

Kuuntelijan luokan tekeminen ei riitä. Kuuntelijasta pitää 1) luoda olemassa oleva kloonin ja 2) se pitää lisätä jollekin aiemmin luodulle ikkunan komponentille. Esimerkiksi näin:

```

JButton nappula = new JButton("Olen kissa.");
NappulaKuuntelija kuuntelija = new NappulaKuuntelija();
nappula.addActionListener(kuuntelija);

```

Tehtävä: Asiakaspalvelu_4

Ikkuna -luokassa ikkuna luodaan tällä hetkellä **run** -metodissa. JButton -nappulan kuvan lisäyksen (setIcon) jälkeen luo uusi TilausKuuntelija ja lisää se nappulalle. Tee ja lisää TilausKuuntelija, kuten edellisen esimerkin NappulaKuuntelija.

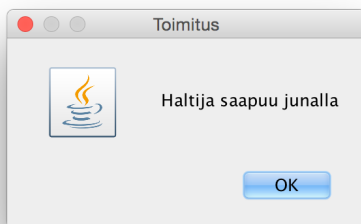
Käynnistä ohjelma ja testaa, että nappulaa klikatessa Netbeansin ohjelmansuoritusriville ilmestyy kuuntelijaan kirjoittamasi viesti.

Pop up -viestit

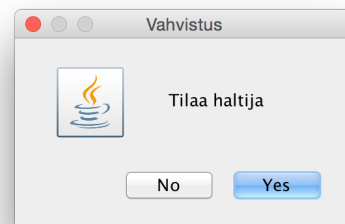
Ohjelmien käyttäjät harvoin vilkuilevat konsolista ohjelmien kirjoittamia viestejä. Käyttäjille viestit kannattaakin antaa pop up -ikkunoissa.

Erilaisia pop up -ikkunoita:
(Ubuntussa ja Windowsissa ikkunat näyttävät erilaisilta)

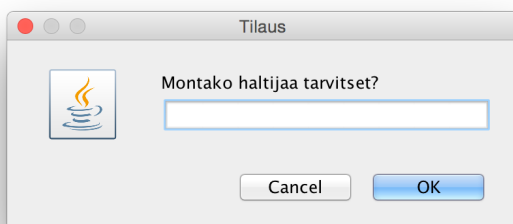
messageDialog = viestivuoropuhelu



confirmDialog = vahvistusvuoropuhelu



inputDialog = syöttövuoropuhelu



Tällaisia pop-up -ikkunoita osaa tehdä JOptionPane. Se on myös staattinen luokka, joka vain lymyää ohjelman lähistöllä, siitä ei tarvitse tehdä oikeaa kloonin, jotta sitä voi komentaa.

Yksinkertainen viesti (show = näyttää):

```
JOptionPane.showMessageDialog(null, "Haltija saapuu junalla.");
```

Kaikkiin viesteihin voi myös lisätä otsikon ja viestityypin:

```
JOptionPane.showMessageDialog(null, "Haltija matkalla.", "Toimitus",
JOptionPane.INFORMATION_MESSAGE);
```

Viesti

Ikkunan otsikko

Tehtävä: Asiakaspalvelu_5

Ohjelmoi **Tilauskuuntelija** -luokkaan, että normaalin tekstin kirjoittamisen sijasta tehdäänkin pop up -ikkuna. Kirjoita haluamasi viesti `showMessageDialog` -kutsuun.

Testaa ohjelman toiminta.

Bonus: Asiakaspalvelu_6

Viesti-ikkunan voi ohjelmoida myös niin, ettei alkuperäisessä ikkunassa työskentelyä voi jatkaa ennen kuin viestiin on vastattu.

Kun loimme viestin, sille annettiin "**viesti**", mutta myös `null`. `null` -viite (=nolla) tarkoitti, ettei viesti-ikkuna tiennyt mihin alkuperäiseen ikkunaan se liittyi. Sen tilalle voitaisiin kuitenkin kirjoittaa myös alkuperäisen ikkunan viite, jolloin se jäätyisi, kunnes vastaus on saatu. Tämän viitteen saa `ActionEvent`iltä. Se tietää aina mikä komponentti johti tapahtumaan:

```
public void actionPerformed(ActionEvent ae) {
    JOptionPane.showMessageDialog( (Component)ae.getSource(), "Heipä hei!" );
}
```

Testaa, että alkuperäinen ikkuna nyt todella muuttuu passiiviseksi kunnes viestiin vastataan.

Bonus-tietoa: casting eli muuntelu

Edellisen tehtävän `getSource` (`get=saada, source=lähde`) -metodi antoi sen *objektin*, josta tapahtuma syntyi. `showMessageDialog` -tarvitsi kuitenkin *ikkunakomponentin* (`component`) viitteeksi. Onneksi nappulamme oli objektin lisäksi myös ikkunakomponentti, joten pystyimme muuntamaan annetun objektin komponentiksi.

`(Component)` jonkin toisen asian edessä on vähän niin kuin naamiaiset. Toiselle komponentille laitetaan ylle `Component`-vaate. Tämä ei tietenkään aina toimi, esimerkiksi nappulaamme ei voisi millään tavalla pukea `(Kissa)` -asusteeseen, koska se ei ole `Kissa`.

Pukeminen on aina mahdollista ohjelmakoodissa, mutta jos yhtä oliota koitetaan pukea vaatteeseen, johon se ei sovi, tulostuu konsoliin `ClassCastException` (luokan muunto poikkeus) ohjelman ollessa käynnissä. Ohjelma voi kaatua virheeseen tai vähintään alkaa toimia oudosti. Tämän johdosta luokkia muunneltaessa tulee aina olla hyvin varma tekemisistään ja ohjelmaan on ehdottoman hyvä tällöin lisätä myös virheenkäsittely.