

PROCESSING

Published : 2012-12-04
License : GPLv2

ESITTELY

1. JOHDANTO
2. ESIMERKKEJÄ
3. PROCESSINGIN ASENTAMINEN

1. JOHDANTO

Processing on merkittävä kehitysympäristö, jossa ohjelmointikieltä käytetään multimediateoksien tekemiseen tietokoneella. Sen toimintafilosofian ja helppokäyttöisyyden lisäksi Processingin viehättävyys perustuu sen käyttötarkoitusten moninaisuuteen: näihin kuuluvat kuvat, äänet, verkkosovellukset, matkapuhelimet ja elektronisten laitteiden interaktiivinen ohjelmointi.

Taiteilijoiden taiteilijoille suunnittelema Processing tuo yhteen ammattimaisten ja amatöörimäisten käyttäjien yhteisön, johon kuuluu graafisia suunnittelijoita, videokuvaajia, painotaloja, arkkitehtejä ja verkkosuunnittelijoita. Sitä käyttävät myös taideopettajat, jotka tahtovat opettaa oppilailleen ohjelmoinnin käyttöä taiteeseen. Processingin kehittäjät tekivät sen alusta asti opetusvälineeksi.

Processing on sekä luovuuden että ohjelmoinnin ympäristö.

PIIRRÄ JA LUO TIETOKONEKODILLA

Multimedian luomiseen tarkoitettuna ohjelmana Processingin erikoisuutena on tietokoneen komentojen käyttö piirtämiseen, 2D tai 3D -animaatioiden tekemiseen, äänitaiteen luomiseen ja ympäristönsä kanssa vuorovaikutuksessa olevien digitaalisten olioiden suunnitteluun.

Kynän, siveltimen, hiiren tai piirtoalustan kaltaisten vapaan käden työkalujen käyttöön tottuneille taiteilijoille voi olla yllättävää, että muotoja, värejä ja liikkeitä voi luoda vain kirjoittamalla tietokoneelle käskyn.

Tämä taiteellisen ilmaisun tapa käyttää koodin avulla tietokoneen ominaisuuksia (nopeutta, toimintojen automaattisuutta ja moninkertaistamista, vuorovaikutteisuutta jne.) luodakseen alkuperäisiä töitä, joita ei olisi voinut tehdä muuten, ja joiden luominen olisi vaatinut enemmän aikaa perinteisten menetelmien tai monimutkaisten ohjelmien avulla.

Processingin avulla voidaan myös ohjelmoida mikrokontrollereita, jotka ovat vuorovaikutuksessa ympäristönsä kanssa. Yhdistettynä ääntä, lämpöä tai liikettä aistiviin sensoreihin nämä halvat mikrokontrollerit voivat vuorostaan luoda kuvia, aktivoida robottikäden, lähettää viestejä internetissä... Tietenkin riippuen tekemästäsi ohjelmasta.

Tässä käyttöoppaassa autetaan sinua kehittämään itseilmaisun kykyä opetellesasi ohjelmoimaan Processingilla. Ainoastaan mielikuvituksesi rajoittaa luomuksiasi.

OPPIMISYMPÄRISTÖ

Processing on helppokäyttöinen, se on alusta alkaen suunniteltu opettamaan ohjelmoinnin perusteita visuaalisessa ympäristössä.

Processingin opetustavoite tekee siitä loistavan työkalun ohjelmoinnin opiskelemiseen ei-ohjelmoijille tai aloitteleville ohjelmoijille. Monet opettajat käyttävät sitä opettaakseen oppilailleen ohjelmoinnin käsitteitä ja käytäntöä.

Toivottavasti tämä opas lisää Processingiin kohdistuvaa mielenkiintoa opettajien, kouluttajien ja laajemmassa mittakaavassa ammattilaisten ja harrastajien keskuudessa.

VAPAA OHJELMISTO

Processing on vapaa ohjelma. Se toimii Windowsissa, Linuxissa, Macissa (ja missä tahansa muussa käyttöjärjestelmässä, jossa toimii Java). Siitä on myös versioita matkapuhelimiin ja mikrokontrollereihin.

Vapaana ohjelmana Processingin etuna on vapaaehtoisten ohjelmoijien anteliaisuus. He ovat luoneet käyttäjille kierrätettäviä ohjelmanpätkiä (jotka tunnetaan tietokoneslangissa kirjastoina). Yli sata kirjastoa laajentaa ohjelman mahdollisuuksia äänen, videon ja vuorovaikutteisuuden kentillä.

HISTORIAA

Processing on suunniteltu Aesthetics Computation Group (ACG) -laboratoriossa, joka kuuluu MIT Media Lab -laitokseen. Sen suunnittelivat Ben Fry ja Casey Reas vuonna 2001. Tämä ohjelma on enemmän tai vähemmän projektin Design By Numbers lopputulos. Projektin loi laboratorion johtaja, taideohjelmoija John Maeda. Ohjelmointikieltä käsittelevässä kirjassaan Maeda painottaa kuvankäsittelyohjelmoinnin yksinkertaisuutta ja toiminnan taloudellisuutta.

Monia tämän ensimmäisen projektin piirteitä on näkyvillä Processing-ympäristössä: ohjelmistoliittymän yksinkertaisuus, jossa ensisijalle asetetaan kokeellisuus ja oppiminen, ja monet funktiot, jotka toimivat molemmissa ympäristöissä. Processingin suunnittelijat eivät kätke tätä historiaa.

Processingin nykyinen ja tässä käyttöoppaassa käytetty versio on 1.5.1.

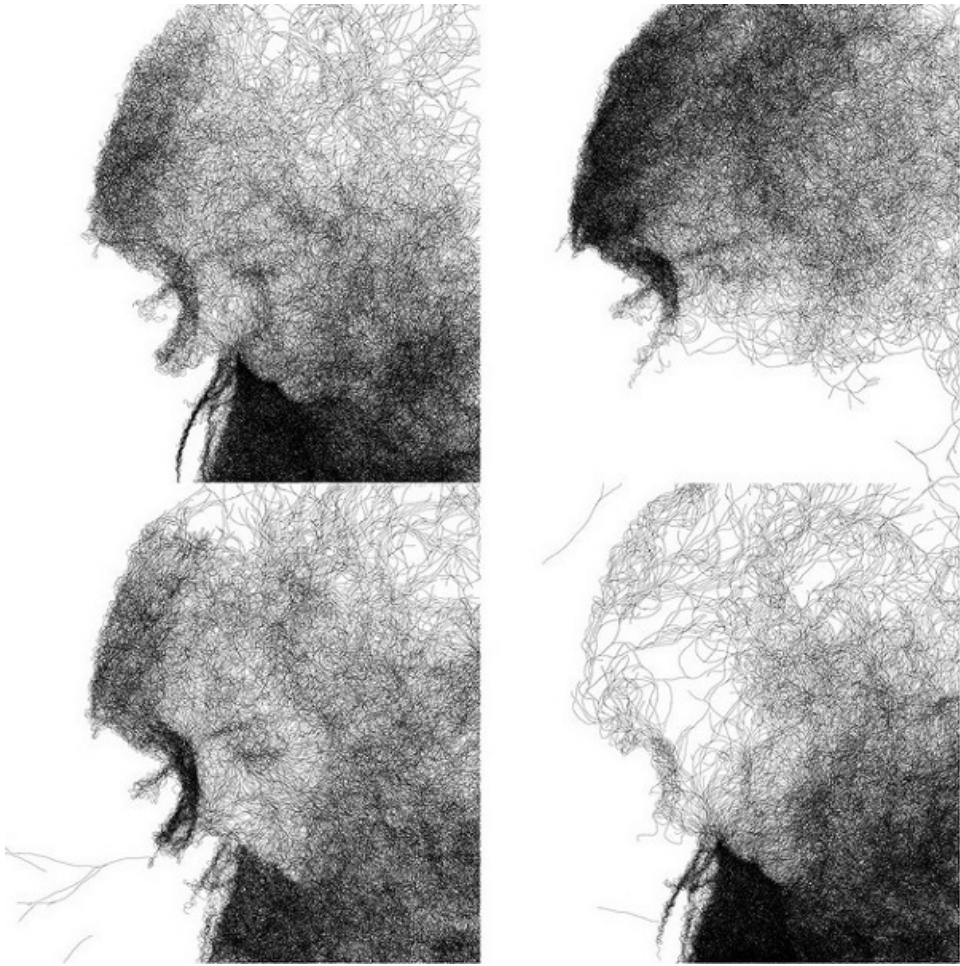
2. ESIMERKKEJÄ

Multimedian luomiseen sopiva ohjelmointikieli Processing mahdollistaa piirtämisen, kaksi- ja kolmiulotteisten animaatioiden tekemisen, taiteilun äänellä, sekä ympäristön kanssa vuorovaikuttavien elektronisten laitteiden suunnittelemisen. Kymmenet tuhannet taiteilijat, suunnittelijat, arkkitehdit, tutkijat ja yritykset käyttävät sitä luomaan hämmästyttäviä projekteja monilla alueilla:

- Processingia on käytetty mainontaan, elokuvien alkuteksteihin, videoihin ja sarjakuviin.
- Processingia on käytetty tieteellisen tiedon visualisointiin monilla monimutkaista tietoa vaativilla alueilla, kuten ympäristön, liikenteen, talouden ja humanististen tieteiden tutkimuksessa.
- Processingia on käytetty luomaan musiikkia ja muuttamaan ääniä.
- On mahdollista laittaa Processing tuottamaan äänitehosteita tanssijoiden liikkeisiin perustuen.
- Arkkitehtuurin alueella Processing mahdollistaa tilan esittämisen, sitä käytetään arkkitehtuuriprojekteissa automatisoimaan rakennusten suunnittelu kahdessa tai kolmessa ulottuvuudessa.

Tässä luvussa esitellään joitain esimerkkejä Processingin käytöstä eri ympäristöissä.

MYCELIUM



Piirteitä ilmestyy ruudulle. Niistä muodostuu vähitellen kasvot. Tämä animaatio simuloi sienten ja bakteerien rihmamaisten osien kehitystä.

Ryan Alexanderin luomus vuodelta 2010:
<http://onecm.com/projects/mycelium/>

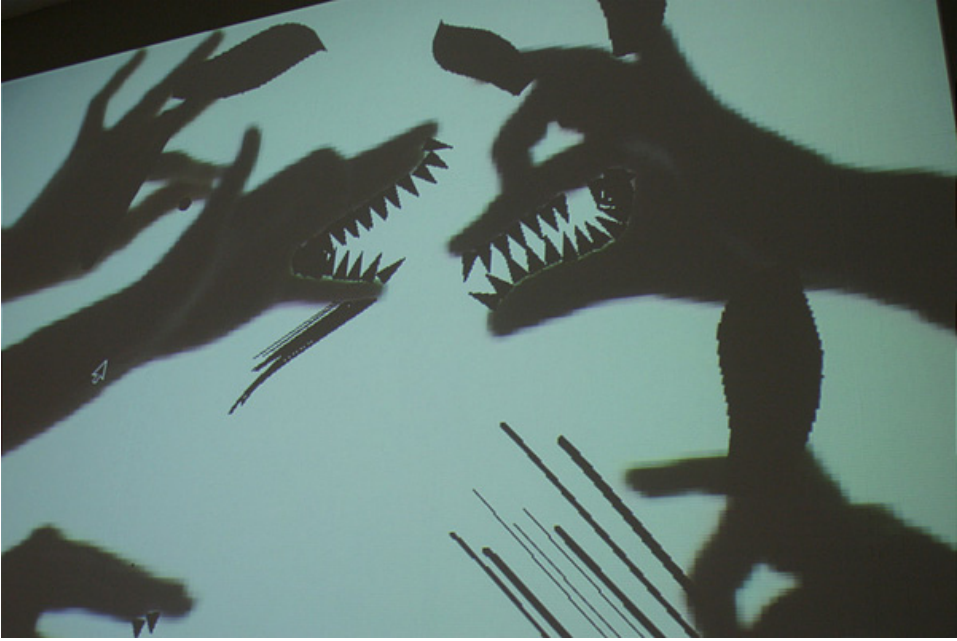
NYTIMES 365/360

NY Times visualisoi New York Timesin verkkosivujen informaatiota. Ohjelma käyttää verkkosivun tietoa ja visualisoi tiedon tärkeyden ja suhteen muuhun tietoon. Lopulta ilmestyy korkean tarkkuuden kuva, joka on valmis tulostettavaksi.

Jer Thropen luomus vuodelta 2009:

<http://blog.blprnt.com/blog/blprnt/7-days-of-source-day-2-nytimes-36536>

SHADOW MONSTERS

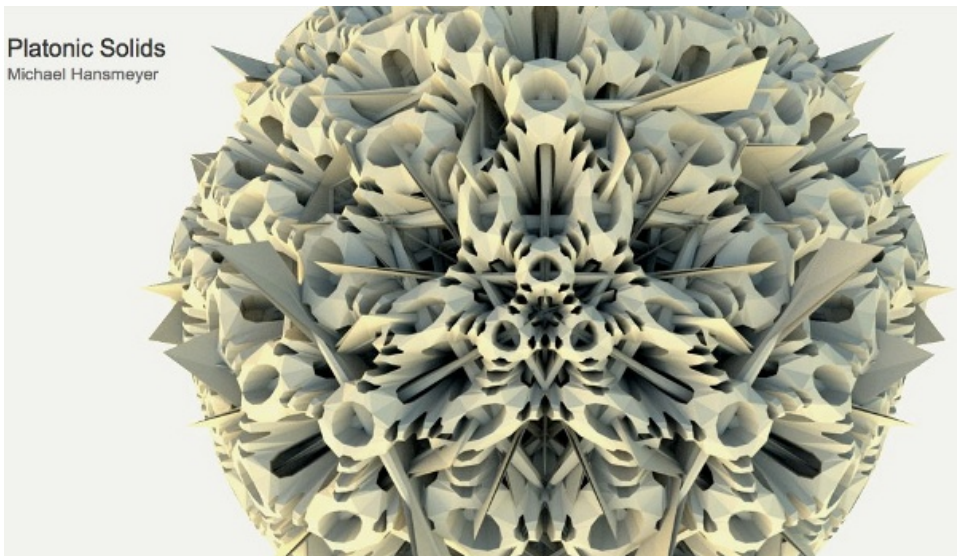


Shadow Monster on interaktiivinen installaatio, joka toimii varjojen avulla. Yleisö näkee varjojensa muuttuvan hirviöiksi ja oudoiksi olennoiksi.

Philip Worthingtonin luomus vuodelta 2005:

<http://worthersoriginal.com/wiki/#page=shadowmonsters>

PLATONIC SOLIDS



Ernst Haeckel kirjoitti 1800-luvulla kirjan luonnossa esiintyvistä taiteellisista muodoista. Tämä ohjelma luo geometrisiä muotoja tuon kirjan inspiroimana.

Michael Hansmeyerin teos: <http://www.michael-hansmeyer.com/html/solids/p0s.html>

CHAMP D'OZONE



Champ d'Ozone on installaatio, joka esiteltiin Pariisissa vuonna 2007 osana Pariisin ilmastoa esittelevää näyttelyä. Se tehtiin yhteistyössä Airparifin kanssa. Se on holograafinen projektio virtuaalisista pilvistä, joiden väri muuttuu jatkuvasti riippuen Pariisin ilmanlaadusta.

HeHen teos vuodelta 2007: <http://hehe.org.free.fr/hehe/champsdozone/>

COP15 GENERATIVE IDENTITY



Yhdistyneiden Kansakuntien ilmastokonferenssin logo suunniteltiin lontoolaisessa studiossa Processingilla. He kehittivät logon perustuen vuorovaikutusvoimiin. Logo ilmentää konferenssin keskusteluiden monimutkaisuutta.

Lontoon studio okdeluxen luomus vuodelta 2009:

<http://www.okdeluxe.co.uk/cop15/>

BODY NAVIGATION

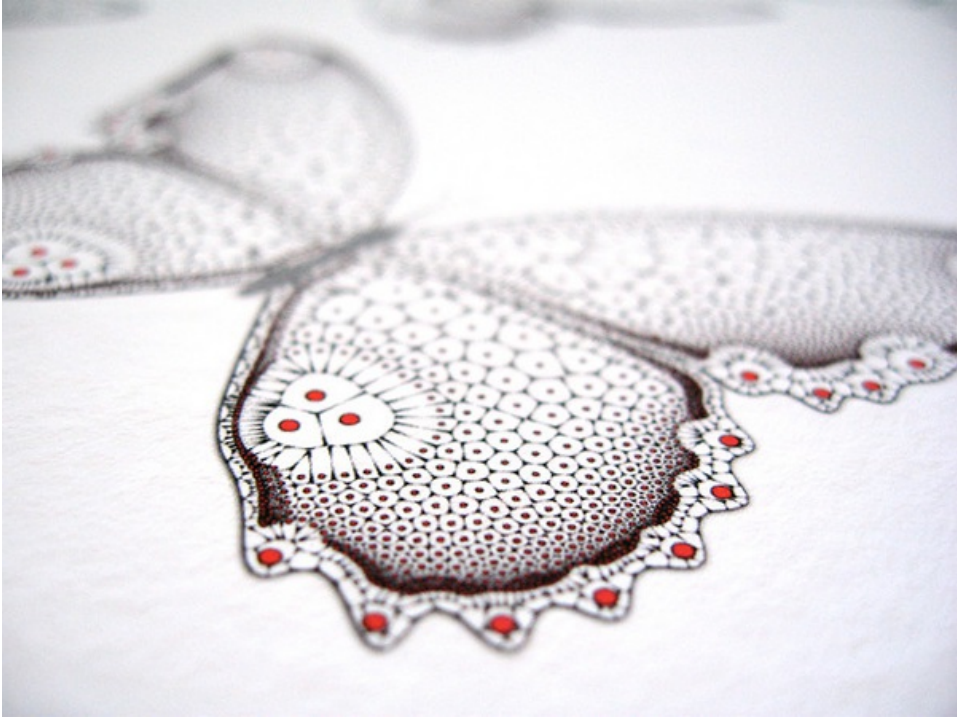


Body navigation on performanssilava. Se luotiin yhteistyönä koreografi Tina Tarpgaardin ja kehittäjä Jonas Jongejanin välillä. Tanssijoita kuvataan infrapunakameralla, joka seuraa heidän liikkeitään, ja Processing kehittää heti kuvat heidän ympärilleen.

Jonas Jongejanin ja Ole Kristensenin luomus vuodelta 2008:

<http://3xw.ole.kristensen.name/works/body-navigation/>

FLIGHT 404



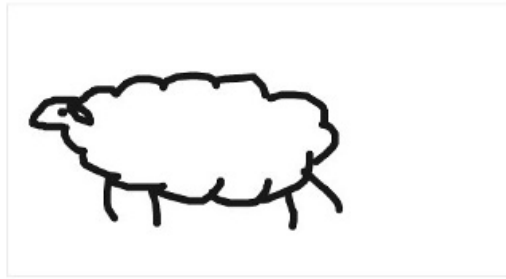
Robert Hodgin tutkii koodin ilmaisumahdollisuuksia, joihin sisältyvät usein algoritmi ja hän itse. Ohjelma luo monia elokuvia, jotka se jakelee nettiin. Nämä esimerkit kierrätetään sitten monella eri tavalla, kuten iT unesissa olevalla katseluohjelmalla ja Voronoi-diagrammeilla luotuina perhosina.

Robert Hodginin luomus vuodelta 2007: <http://flight404.com/>

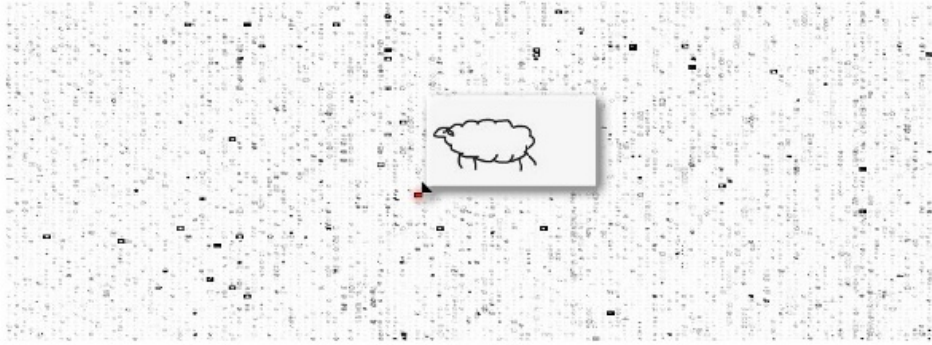
THE SHEEP MARKET

THE SHEEP MARKET

10-000 sheep created
by online workers,
Here...



4445 / 10000



Tämän taideprojektin osana nuoret ja vanhat käyttäjät kutsuttiin piirtämään lammasta. Lampaista tulee jotenkin mieleen Antoine de Saint-Exuperyn teos Pikku prinssi. Nämä piirrokset on jaettu suureen tietokantaan, joka on saatavilla verkossa. Processing voi lukea kaikki 10 000 piirrosta ja visualisoida kynän reitin.

Aaron Koblinin luomus vuodelta 2008:

<http://www.thesheepmarket.com>

Tämän luvun kuvien tekijänoikeudet kuuluvat teosten tekijälle.

3. PROCESSINGIN ASENTAMINEN

Processing on kirjoitettu Java-ohjelmointikielellä, se toimii periaatteessa missä tahansa käyttöjärjestelmässä, joka tukee Java-ohjelmointikieltä. Parhaiten tuetut alustat ovat kuitenkin Windows (Windows XP ja Windows 7), Linux ja Mac OS X.

Processingin asentaminen on aika helppoa ja riippuu käyttöjärjestelmästäsi. Joka tapauksessa voit mennä lataussivulle: <http://processing.org/download> ja napsauttaa käyttöjärjestelmäsi nimeä.

Processingin käyttöliittymä avautuu:

↓ Linux

↓ Mac OS X

↓ Windows

↓ Windows (Without Java)*

WINDOWS

Windowsissa ei kannata valita vaihtoehtoa "Without Java", koska silloin joudut asentamaan Javan erikseen.

Napsauttamalla linkkiä Windows lataat tiedoston "processing-xxx.zip". Kun lataus on valmis, pura arkisto ja laita purettu "Processing"-kansio C:\Program Files\ -hakemistoon.

Tämän jälkeen voit mennä kansioon C:\Program Files\Processing ja ajaa tiedoston "processing.exe".

PERUSTEET

4. PROCESSINGIN PERUSTEET

4. PROCESSINGIN PERUSTEET

Processing tarjoaa sekä kehitysympäristön että kattavan ja helppokäyttöisen funktiokirjaston, jotka on lisätty Java-ohjelmointikieleen. Ohjelmointiympäristöä käytetään ohjelmien kirjoittamiseen, kääntämiseen, julkaisemiseen ja korjaamiseen. Se on yksinkertainen ja sisältää ohjelmoinnin vaatimat ominaisuudet, mutta on silti helppokäyttöinen.

Processing perustuu Javaan. Java on ohjelmoitaessa käytettävä syntaksi. Processing huolehtii yleensä monimutkaisista operaatioista, kuten ikkunoiden, äänen, videon ja kolmiulotteisuuden hallinnasta. Se tarjoaa laajan joukon ennalta määritettyjä metodeja, jotka yksinkertaistavat grafiikkaohjelmien tekemistä, ja tekevät siitä helppokäyttöisen ihmisille, jotka tahtovat luoda grafiikkaa ilman monimutkaisia ohjelmointikäsitteitä ja matematiikkaa.

Tämä luku esittelee Processingin käyttöliittymän perusteet ja minimimäärän Javan syntaksia, jotta pääset alkuun.

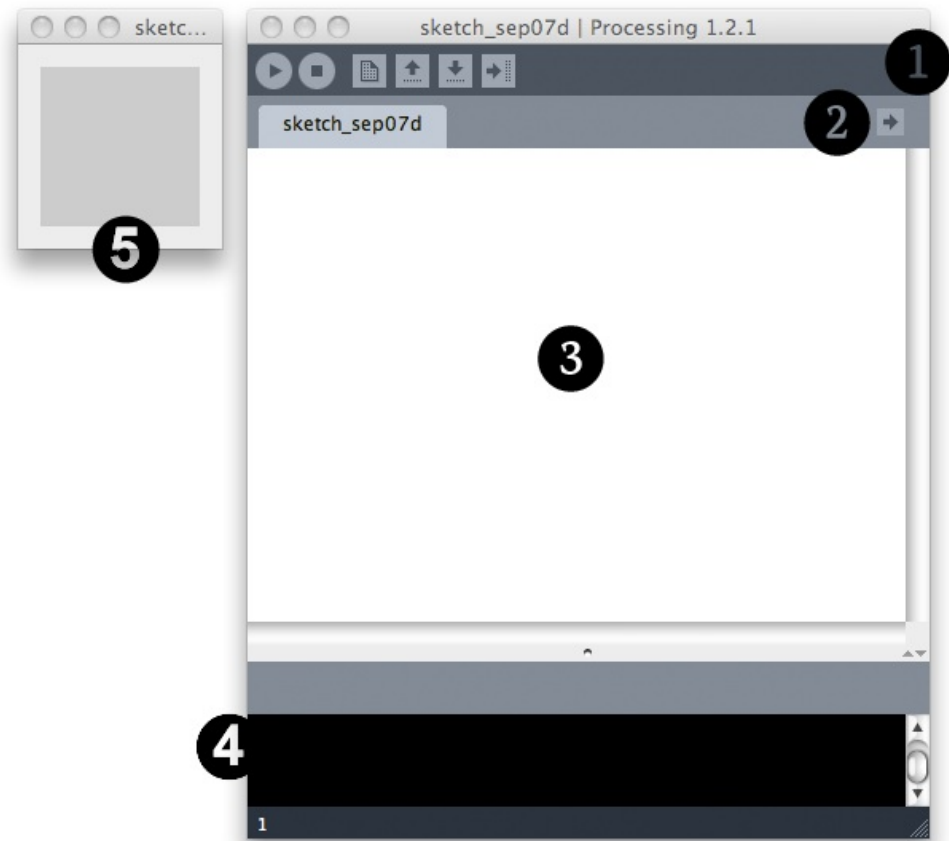
Nykyisin Processingin käyttöliittymä on olemassa vain englanninkielisenä. Sen käyttö on kuitenkin helppoa ja tulevaisuudessa se voi olla tarjolla lokalisoituna.

KÄYTTÖLIITTYMÄ

Processingin käyttöliittymä koostuu kahdesta erillisestä ikkunasta: pääikkunasta, jossa luot projektisi, ja esikatseluikkunassa, jossa ohjelman tulokset näkyvät.

Käyttöliittymässä on seuraavat osat:

1. Toimintopalkki
2. Välilehtipalkki
3. Editointialue
4. Ulostuloalue
5. Katseluikkuna
6. Valikkopalkki (ei näy ruutukaappauksessa)



Toimintopalkki



Suorita ohjelma.



Lopeta ohjelman suoritus.



Luo uusi ohjelma.



Avaa ohjelman.



Tallentaa ohjelman.

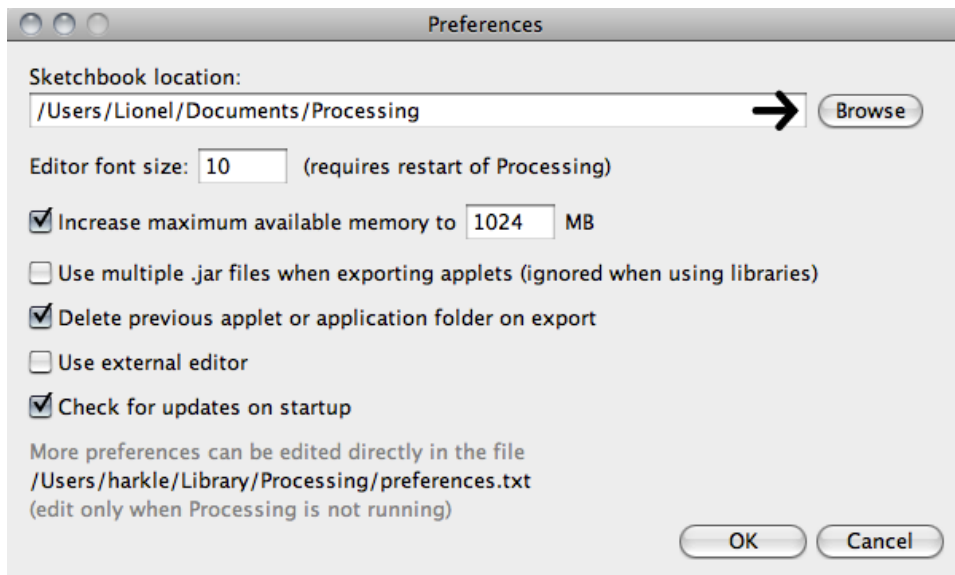


Vie ohjelman verkkoon.

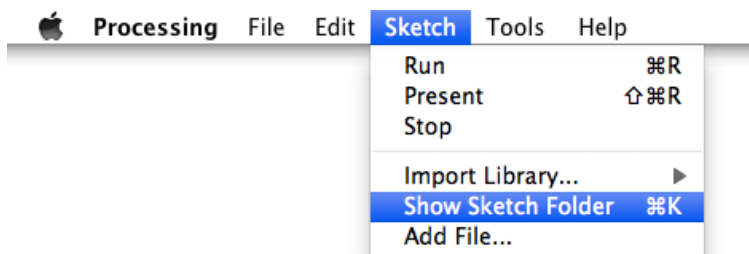
TYÖHAKEMISTO

Tämä on hakemisto, johon ohjelmat tallennetaan. Oletusarvoisesti tämä hakemisto on nimeltään Processing ja sijaitsee hakemistossa **Documents** (Mac) tai **My Documents** (Windows). GNU/Linuxissa tämä on omassa kotihakemistossasi.

Muuttaaksesi hakemistoa, mene valikkoon **Files > Preferences**. Ilmestyvässä ikkunassa voit napsauttaa **Browse** valitaksesi haluamasi hakemiston.



Saadaksesi selville työhakemistosi, valitse valikosta **Sketch > Show Sketch Folder**. Tämä vaihtoehto on myös saatavilla näppäimistökomennolla **Ctrl-k** Windowsissa ja Linuxissa tai **cmd-k** Macissa:



OHJELMOINTIKIELEN PERUSTEET

Processing käyttää Javaa luodakseen luonnoksesi. Tätä kieltä koneesi lukee luodakseen luonnoksesi, ja sillä on joukko syntaksisääntöjä, joita on noudatettava, tai ohjelma ei toimi. On tunnettava myös joitain peruskäsitteitä.

Processing ottaa huomioon kirjainten koon, joten isot ja pienet kirjaimet ovat eri asia: vapaa ja Vapaa ovat eri sanoja!

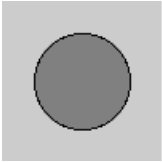
PUOLIPISTE

Jokaisen komennon jälkeen (piirrä ympyrä, tee laskelma, jne.) täytyy kirjoittaa puolipiste ";", joka kertoo ohjelmalle komennon päättyvän. Seuraavassa esimerkissä käytämme merkkejä "/" lisätäksemme kommentin, jota ei oteta huomioon ohjelmaa suoritettaessa.

```
//Luo ellipsi
ellipse(10,10, 10, 10);
//Julista muuttuja
int numero = 10 + 23;
```

METODIKUTSUT

Processing tarjoaa joukon ennaltamääriteltyjä metodeita: piirrä suorakulmio, määrittele väri, laske neliöjuuri jne. Jokaisella näistä metodeista on erityinen nimi. Kutsuaksesi niitä voit kirjoittaa metodin nimen ottaen huomioon isot ja pienet kirjaimet. Nimen jälkeen tulee usein sulut, joihin asetetaan tiettyjä arvoja, kuten väri, sijainti, koko jne. Seuraavassa esimerkissä on harmaa ympyrä.



```
fill(128);
ellipse(50, 50, 60, 60);
```

ULOSTULON NÄYTTÖ

Ulostuloalue näyttää tekstiä testausta ja korjausta varten. Nähdäksesi jotain voit käyttää metodia `println()`;

```
println("Hei maailma!");
```

```
Hei maailma!
```

```
println(1000);
```

```
1000
```

Laskut

Processing voi tehdä matemaattisia laskuja. Tätä ympäristöä käyttäessäsi joudut laskemaan arvoja. Yhteenlasku, vähennyslasku, jakolasku ja kertolasku voidaan yhdistää. Voit käyttää sulkuja määrittelemään toimenpiteiden järjestyksen. Muista pisteet! Processing käyttää desimaaleihin pistettä eikä pilkkua. Tässä on esimerkkejä:

```
println(10 + 5);
println(10 + 5 * 3); // 5*3 (tulos 15) ja lisätään 10
println((10 + 5) * 3); // 10+5 (tulos 15) ja 15 kertaa 3
println(10.4 + 9.2);
```

Tämä tuottaa seuraavan tuloksen:

```
15  
25  
45  
19.599998
```

Joitain aritmeettisia operaatioita voidaan lyhentää. Esimerkiksi `i++` antaa saman tuloksen kuin `i = i + 1`.

Nyt tiedät perusteet Processingista, voit jo kirjoittaa koodia, jonka avulla voit toteuttaa projekteja.

SUUNNITTELU

5. PIIRROSTILA

6. MUODOT

7. VÄRIT

8. KUVAT

9. TEKSTIT

5. PIIRROSTILA

Piirrostilassa näytetään koodin tulokset. Tämä ikkuna näyttää luomuksesi kaksi- tai kolmiulotteisena.

Tämän tilan luo komento `size()`, joka ottaa kaksi argumenttia: `size(leveys, korkeus);`

Kirjoita esimerkiksi seuraava komento Processingin ohjelmaikkunaan:



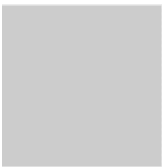
Seuraava ikkuna luodaan napsauttaessasi **suorita**-nappia:



Kokeile ulottuvuuksien muuttamista nähdäksesi tulokset.

Oletusarvoisesti:

```
size();
```

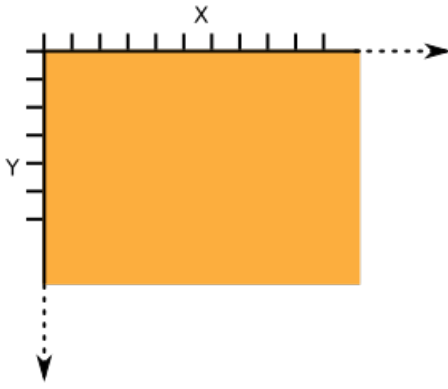


...luo ikkunan, jonka koko on 100 pikseliä kertaa 100 pikseliä.

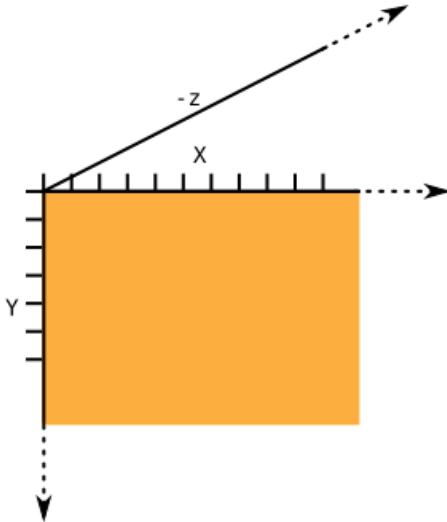
Onnittelut, olet luonut ensimmäisen ikkunas!

KOORDINAATTIAVARUUS

Kun työskentelet kahdessa ulottuvuudessa, akselien x ja y koordinaatit vastaavat leveyttä (horisontaalinen akseli) ja korkeutta (vertikaalinen akseli). Sopimuksen mukaan vasen yläkulma vastaa arvoja $x = 0$ ja $y = 0$. Akselin x arvot nousevat oikealle ja akselin y arvot lisääntyvät alaspäin, toisin kuin kartesiaanisessa tasossa. Nämä x ja y -arvot voivat teoriassa jatkua loputtomiin, vaikka käytännössä ikkunas koko rajoittaa näkyvän alueen maksimikoko. Tässä tilassa voimme piirtää.



Ja kun työskentelemme kolmiulotteisessa avaruudessa, kahden koordinaattiakselin lisäksi meillä on kolmas koordinaattiakseli z , joka esittää syvyyttä:



Tässä tapauksessa käytämme komentoa `size` yhdessä kolmannen parametrin kanssa osoittaaksemme, että työskentelemme kolmiulotteisessa avaruudessa `3D size(100, 100, P3D);`

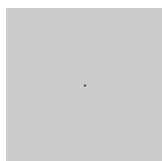
6. MUODOT

Processing tarjoaa monia muotoja. Tässä ovat tärkeimmät.

PISTE

Aloitamme piirtämisen pisteellä. Yksi piste vastaa pikseliä, joka on ikkunassa sijainnissa x leveyssuunnassa ja sijainnissa y korkeussuunnassa. Koordinaatit annetaan muodossa (x, y) . Tässä esimerkissä piste on hyvin pieni. Se on keskipisteessä.

```
point(50, 50);
```



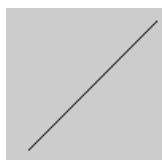
Huomaa, että kehys on 100x100, joten piste on keskellä. Jos piirrämme kehyksen ulkopuolelle, et näe pistettä.

VIIVA

Viiva (AB) koostuu määritelmällisesti loputtomasta määrästä pisteitä alkupisteen A ja loppupisteen B välillä. Rakentaaksemme sen keskitymme vain pisteiden A ja B koordinaatteihin x ja y . Jos esimerkiksi oletusikkunassa piste A on ikkunan vasemmassa alanurkassa, ja piste B on oikeassa ylänurkassa, voimme piirtää tämän viivan komennolla

```
line(xA, yA, xB, yB) :
```

```
line(15, 90, 95, 10);
```

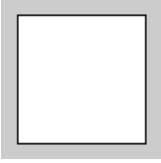


SUORAKULMIO

Suorakulmio piirretään neljällä arvolla komennolla `rect(x-yläkulma, y-yläkulma, leveys, korkeus)`. Ensimmäinen x ja y -arvojen pari merkitsee suorakulmion vasenta yläkulmaa, kuten pisteen tapauksessa. Sen sijaan seuraavat kaksi arvoa eivät merkitse vasenta alakulmaa, vaan tämän suorakulmion leveyttä (x -akseli, horisontaalinen) ja korkeutta (y -akselilla, vertikaalinen).

Esimerkki:

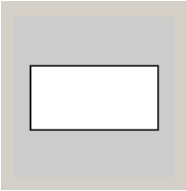
```
rect(10, 10, 80, 80);
```

Koska kaksi viimeistä arvoa (leveys ja korkeus) ovat identtiset, saamme neliön. Muuttele arvoja ja katso tuloksia.

Käytä metodia `CENTER` saadaksesi kaksi ensimmäistä arvoa vastaamaan suorakulmion keskipistettä:

```
rectMode(CENTER);  
rect(50, 50, 80, 40);
```

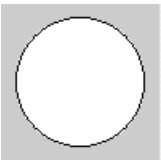


ELLIPSI

Kuten suorakulmio, ellipsi luodaan tilassa `CENTER` (oletusarvoinen tila) tai tilassa `CORNER`. Niinpä seuraava komento luo neliön, jonka keskipisteen koordinaatit ovat kaksi ensimmäistä arvoa sulkujen sisällä. Kolmas arvo vastaa horisontaalisen akselin läpimittaa ja vastaa vertikaalisen akselin läpimittaa: huomaa, että jos kolmas ja neljäs arvo ovat identtisiä, saat ympyrän, muuten saat ellipsin:

```
ellipse(50, 50, 80, 80);
```

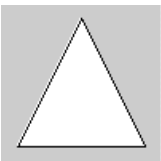
Kokeile kolmannen ja neljännen arvon muuttamista ja tarkkaile tuloksia.



KOLMIO

Kolmio on kolmen pisteen muodostama kuvio. Kolmio luodaan komennolla `triangle(x1,y1,x2,y2,x3,y3)`, jolla määritellään kolmion kolme kärkipistettä:

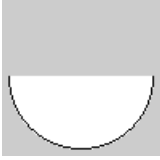
```
triangle(10, 90, 50, 10, 90, 90);
```



KAARI

Ympyrän kaari voidaan tehdä komennolla `arc(x, y, leveys, korkeus, alku, loppu)`, jossa pari `x, y` määrittelee ympyrän keskipisteen, toinen ja kolmas pari määrittää kulman loppupisteen:

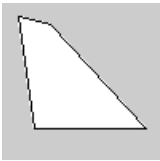
```
arc(50, 50, 90, 90, 0, PI);
```



NELITAHOKAS

Nelitahokas luodaan määrittelemällä neljä paria `x` ja `y` -koordinaatteja komennolla `quad(x1,y1,x2,y2,x3,y3,x4,y4)` myötäpäivään:

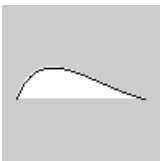
```
quad(10, 10, 30, 15, 90, 80, 20, 80);
```



KÄYRÄ

Käyrä luodaan komennolla `curve(x1, y1, x2, y2, x3, y3, x4, y4)`, jossa `x1` ja `y1` määrittelevät käyrän ensimmäisen läpikulkupisteen, `x4` ja `y4` toisen läpikulkupisteen, `x2` ja `y2` alkupisteen ja `x3`, `y3` loppupisteen:

```
curve(0, 300, 10, 60, 90, 60, 200, 100);
```



BÉZIER-KÄYRÄ

Toisin kuin komento `curve()`, Bézier-käyrä luodaan komennolla `bezier(x1,y1,x2,y2,x3,y3,x4,y4)`

```
bezier(10, 10, 70, 30, 30, 70, 90, 90);
```



TASOITETTU KÄYRÄ

Komento `curveVertex()` luo joukon x ja y -koordinaatteja kahden läpikulkupisteen välille komennolla `curveVertex(ensimmäinen läpikulkupiste, xN, yN, xN, yN, xN, yN, viimeinen läpikulkupiste)` :

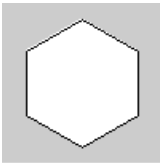
```
beginShape();
curveVertex(0, 100);
curveVertex(10, 90);
curveVertex(25, 70);
curveVertex(50, 10);
curveVertex(75, 70);
curveVertex(90, 90);
curveVertex(100, 100);
endShape();
```



VAPAAAT MUODOT

Kokonainen joukko vapaita muotoja voidaan piirtää tasolle pisteiden sarjana, komennoilla `beginShape()`, `vertex(x,y)`, ..., `endShape()`. Jokainen piste luodaan koordinaateilla x ja y. Funktio `CLOSE` komennossa `endShape(CLOSE)` osoittaa, että muoto suljetaan, ja että jälkimmäinen yhdistetään, kuten kuusikulmion esimerkissä:

```
beginShape();
vertex(50, 10);
vertex(85, 30);
vertex(85, 70);
vertex(50, 90);
vertex(15, 70);
vertex(15, 30);
endShape(CLOSE);
```



ÄÄRIVIIVAT

Olemme havainneet, että tähän mennessä kaikilla mainituilla esimerkeillä on ääri viivat ja täytetty pinta. Jos tahdot tehdä näkymättömän ääri viivan, käytä komentoa `noStroke()`:

```
noStroke();
quad(10, 10, 30, 15, 90, 80, 20, 80);
```

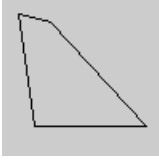


TÄYTE

Vastaavasti voimme jättää pinnan täyttämättä käyttämällä komentoa

`noFill()` :

```
noFill();  
quad(10, 10, 30, 15, 90, 80, 20, 80);
```



Tähän mennessä ikkunan tausta on ollut harmaa, muotojen reunat ovat olleet mustat, ja täyttö on ollut valkoinen. Seuraavassa luvussa opit muuttamaan nämä värit.

7. VÄRIT

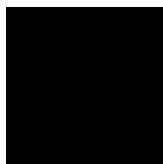
Piirrä kuva ruudulle ja muuta pikselien väriä. Pikselit ovat pieniä alueita, yleensä neliömäisiä, joilla on väri. Jokaisella värillä on kolme kanavaa, jotka ovat punainen, vihreä ja sininen. 100% arvo jokaisella kanavalla on valkoinen. 0% arvo jokaisella kanavalla on musta. Se on valoa, ei maalia. Niinpä jokaisen kanavan arvon kasvaessa sen väri vaalenee.

Esimerkiksi 100% punainen, 80% vihreä, 20% sininen antaa värin oranssi. Metodi `fill()` antaa meidän määritellä värin seuraaville piirrettäville muodoille. Jokainen värikanava annetaan asteikolla 0-255. Niinpä 80% arvosta 255 on 204 ja 20% arvosta 255 on 51.

TAUSTAVÄRIT

Voit muuttaa taustaväriä metodilla `background()`. Huomaa: komento `background()` pyyhkii ruudun edellisen komennon jälkeen!

```
background(0, 0, 0);
```



TÄYTTÖVÄRI

Joka kerta kun piirrät muodon, se tehdään täyttövärillä, joka valitaan tässä vaiheessa. Se tehdään kutsumalla metodia `fill()`.



```
noStroke();  
fill(255, 204, 51);  
rect(25, 25, 50, 50);
```

Processing tarjoaa erilaisia formaatteja värin ilmaisemiseen. Jos teet verkko-ohjelmointia, tunnet luultavasti heksadesimaaliformaatin. Kirjoitamme siis:

```
fill(#ffcc33);  
rect(25, 25, 50, 50);
```

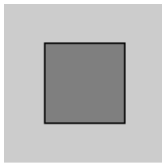
Voit määritellä värin alfakanava-arvon. Se esittää värin läpinäkyvyyden. Tehdäksemme tämän menetelmälle täytyy tarjota neljä arvoa. Neljäs arvo on alfa-arvo.

```
noStroke();  
fill(255, 204, 51); // oranssi  
rect(25, 25, 50, 50);
```

```
fill(255, 255, 255, 127); // valkoinen osittain läpinäkyvä
rect(35, 35, 50, 50);
```



Jos tahdot valita värin, joka on harmaan sävy, voimme antaa yhden parametrin komennolle. Se on harmaan arvo väliltä 0-255.



```
fill(127);
rect(25, 25, 50, 50);
```

Voit ottaa muotojen täytön pois päältä kutsumalla metodia `noFill()`.

ÄÄRIVIIVAN VÄRI

Muuttaaksesi ääri viivan väriä, voit kutsua metodia `stroke()`, jonka parametreinä on halutun värin kanavat. Kutsu kommentoaa `noStroke()` poistaaksesi ääri viivan värit.



```
size(200, 200);
smooth();
background(255); // piirrämme valkoisen taustan

stroke(#000000); // ääri viiva on musta
fill(#FFFF00); // täyte on keltainen
strokeWeight(5);

translate(width / 2, height / 2);

ellipse(0, 0, 120, 120);

stroke(255, 0, 0); // viiva on punainen
line(20, -40, 20, 0);

stroke(0, 0, 255); // viiva on sininen
line(-20, -40, -20, 0);

noFill(); // seuraavaa muotoa ei täytetä
stroke(#000000); // viivasta tulee musta
bezier(-45, 0, -45, 60, 45, 60, 45, 0);
```

VÄRIMUUTOSTEN LAAJUUS

Oletusarvoisesti mikä tahansa tyylimuutos (täytön väri tai esineen muoto tai paksuus) vaikuttaa kaikkeen, jonka maalaat. Rajoittaaksesi näiden muutosten laajuutta voit laittaa muutoksesi komentojen `pushStyle ()` ja `popStyle ()`:n väliin.



```
size(100, 100);
background(255);

stroke(#000000);
fill(#FFFF00);
strokeWeight(1);
rect(10, 10, 10, 10);

pushStyle(); // Avaa tyylin "sulkeet"

stroke(#FF0000);
fill(#00FF00);
strokeWeight(5);
rect(40, 40, 20, 20);

popStyle(); // Sulkee tyylin "sulkeet"

rect(80, 80, 10, 10);
```

Harjoitus:

Poista komennot `pushStyle()` ja `popStyle()` ja tarkkaile muutoksia.

VÄRIAVARUUS

Värien valinnassa punainen, vihreä ja sininen on yksi mahdollisista väriavaruuksista. Processing tarjoaa myös HSB-tilan. Tämä merkitsee "hue, saturation, brightness", eli väri, kylläisyys, kirkkaus. Valitsemme arvon 0-100 jokaiselle kanavalle. Väri on numero, joka valitsee värin sijainnin kromaattisella skaalalla, joka on sateenkaari. Punainen on vasemmalla, sen jälkeen tulevat oranssi, keltainen, vihreä, sininen ja purppura.

Metodi `colorMode()` on käyttökelpoinen muutettaessa numeroskaalaa, jolla määritellään värit ja muutetaan väriavaruutta. Esimerkiksi komento `colorMode(RGB, 1.0)` muuttaa skaalan, jota käytetään määrittämään jokaista värikanavaa, joten se menee nollasta ykköseen.

Tässä muutamme väritilaksi HSB:n tehdäksemme liukuvärin, joka muistuttaa sateenkaaren värejä.



```
noStroke();
size(400, 128);
```

```
// Väri määritellään numeroilla 0-400
colorMode(HSB, 400, 100, 100);

// Se on 400 toistoa...
for (int i = 0; i < 400; i++) {
  fill(i, 128, 128);
  rect(i, 0, 1, 128);
}
```


8. KUVAT

Processingin "kuva" ei ole pohjimmiltaan mitään muuta kuin kokoelma pikseleitä, jotka on koottu suorakulmion sisään. Piirtääksemme kuvan ruudulle laitamme jokaiselle suorakulmion pikselille värin, jonka jälkeen annamme sijainnin (x, y), johon tahdomme piirtää tämän pikselikokoelman. On myös mahdollista muuttaa kuvan kokoa (leveys, korkeus), vaikka nämä ulottuvuudet eivät olisi yhteensopivia kuvan alkuperäisen koon kanssa.

ETSI KUVA

Piirtääksesi kuvan Processingissa, voit etsiä kuvan ja tuoda sen piirrookseen. Voit ottaa kuvan ja laittaa sen tietokoneellesi, ottaa kuvan kamerallasi, tai etsiä kuvia, jotka ovat koneellasi. Tässä harjoituksessa kuvan alkuperällä ei ole väliä. Aluksi neuvomme kokeilemaan melko pientä kuvaa, jonka koko on 400 x 300 pikseliä.

Tässä aloitamme hieman kutistetulla kuvalla Ermenonvillen saarelta. Se löytyy Wikimedia Commonsista seuraavalta osoitteelta:

<http://fr.wikipedia.org/wiki/Fichier:Erm6.JPG>



KUVAFORMAATIT

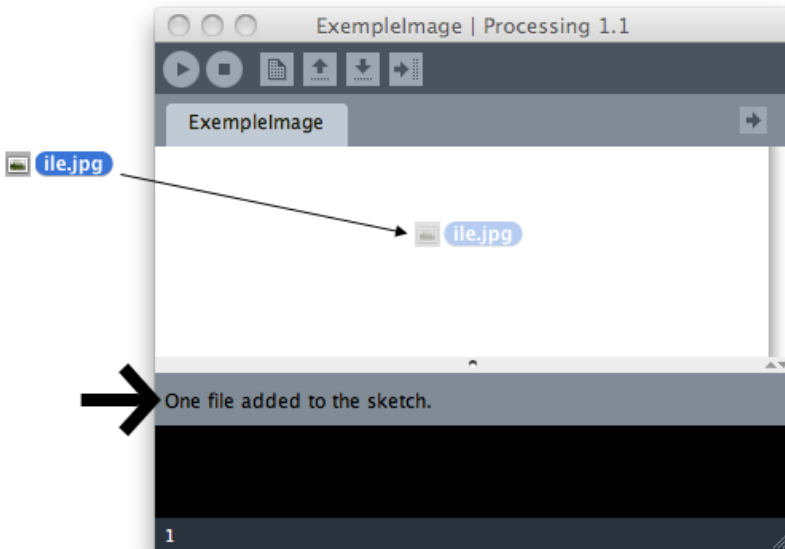
Kolme kuvaformaattia hyväksytään Processingissa: PNG, JPEG tai GIF. Verkkosivuja suunnitelleiden ihmisten pitäisi tunnistaa nämä kolme kuvaformaattia, koska ne ovat suosituimpia WWW:ssä. Jokainen formaatti käyttää omia kuvanpakkauksen menetelmiään, joilla on etunsa ja haittansa:

1. JPEG -kuvilla esitetään usein pakattuja valokuvia. JPEG ei salli läpinäkyviä alueita.
2. GIF -kuva on usein käytössä käyttöliittymän napeissa ja muissa elementeissä. Siinä voi olla läpinäkyviä alueita. Se voi olla animoitu, mutta Processing ei kiinnitä huomiota animaatioihin.
3. PNG -kuvia käytetään molempiin edellämainittuihin käyttötarkoituksiin (kuvat ja käyttöliittymä) ja siinä voi olla läpinäkyviä alueita, jotka eivät ole binääriä, ja jossa on asteittaisina vaihtoehtoina läpinäkyvä, osittain läpinäkyvä ja täysin läpinäkymätön.

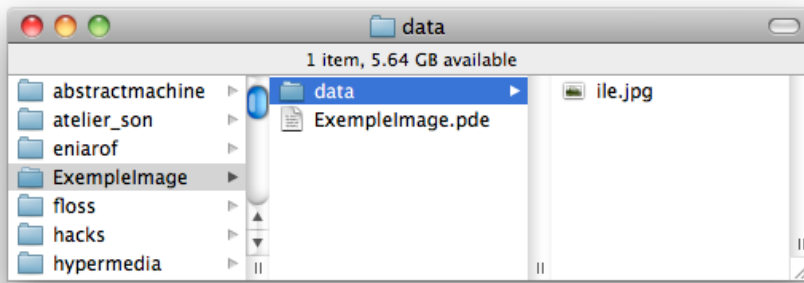
VETÄMINEN

Tuomme nyt kuvatiedoston Processing-ympäristöön. Saadaksemme tämän aikaan suosittelemme tallentamaan ensimmäisen luonnoksesi mieluiten Processing-tiedostoon, jonka pitäisi olla oletusarvoisesti My Documents tai Documents -hakemistossasi.

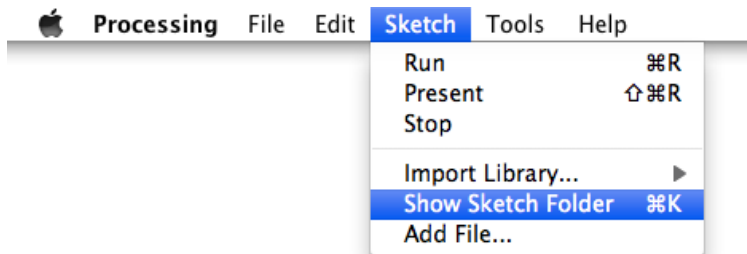
Etsi nyt kuvatiedostosi ja vedä se suoraan Processing-ikkunaan:



Kun siirrämme tiedostoja tällä tavalla, Processing osoittaa, että kuva lisättiin. Todellisuudessa Processing vain kopioi kuvan tiedostoon hakemistossa **Data**, jonka pitäisi näkyä nyt ohjelmasi vieressä.



Tässä tapauksessa hakemisto **Data**, johon itse asiassa laitamme kaikki kuvat - ja jopa muun median, kuten kirjasimet ja äänet - joita tahdomme käyttää kuvassamme. Jos tahdot päästä tähän hakemistoon nopeasti, esimerkiksi laittamalla sinne lisää kuvatiedostoja, valitse vain hakemistosta **Sketch** vaihtoehto **Show Sketch Folder**. Tähän vaihtoehtoon pääset myös valitsemalla pikavalinnan **Ctrl-k** Windowsissa ja Linuxissa tai **Cmd-k** Macissa:



LATAA TIEDOSTO

Nyt kun olemme asettaneet kuvan **Data**-hakemistoomme, voimme käyttää niitä ohjelmassamme.



```
size(500,400);  
PImage saari = loadImage("ile.jpg");  
image(saari,50,10);
```

Ensin olemme antaneet piirroksellemme alueen, joka on isokokoisempi kuin kuvamme, jotta voimme paremmin ymmärtää, kuinka se voidaan sijoittaa viimeiseen luonnokseen.

On kolme tapaa katsoa kuvaa Processingissa:

1. Luo muuttuja, joka sisältää kuvan datan (pikselit).
2. Tuo tiedoston pikselit muuttujaamme.
3. Piirrä tämän muuttujan pikselit suunnittelutilassamme.

Ensin Processingissa täytyy luoda muuttuja. Mutta mikä on muuttuja? Tässä tapauksessa se on ohjelman sisäinen nimi, joka sisältää kaikki pikselit tiedostosta ile.jpg. Kun kirjoitamme sanan "saari", ohjelmamme ymmärtää, että tämä on pikselikokoelma, joka tekee kuvan. Tämä on täsmälleen funktion `loadImage("tiedostonimi")` tarkoitus: se etsii nämä pikselit tiedostosta ja vie ne muuttujaan, jonka nimi on "saari".

Olet ehkä huomannut myös oudon `PImage`-sanan edellisen koodin alussa. Tämä osoittaa, millaisen Processing-muuttujan se avaa. Muuttujalle varataan sopiva määrä tilaa. Kaava on `{muuttujan tyyppi}{muuttujan nimi}={muuttujan arvo}`.

Jos olisi esimerkiksi tarpeen tuoda koiran kuva Processingiin, voisimme käyttää komentoa `PikkuKoiran milou = loadDog("milou.dog");` Kirjoitamme ensin olion tyyppin ja sen jälkeen sen arvon.

Tässä arvo annetaan funktiolla `loadImage()`, joka hakee kuvatiedoston pikselit ja tuo ne muuttujaan, jonka nimi on `saari`.

Lisätietoa muuttujista ja muuttujatyypeistä on luvussa, joka käsittelee tätä aihetta.

Lopulta, kun olemme täyttäneet muuttujamme, vedämme sen sijaintiin `{x,y}` kuvassamme. Jos tahdomme piirtää kuvan alkuperäisessä koossaan, käytämme lyhyttä versiota komennosta `image`, joka vaatii vain kolme muuttujaa: `{PImage}, {x}, {y}`.

```
image(saari,50,10)
```

TUO KUVA VERKOSTA

Voimme myös tuoda kuvan suoraan World Wide Webistä, osoittaen kuvan osoitteen tiedostonimen sijasta.

```
size(400,400);
```

```
PImage webcam;  
webcam = loadImage("http://www.gutenberg.org/files/3913/3913-  
h/images/rousseau.jpg");  
image(webcam,10,20,width,height);
```

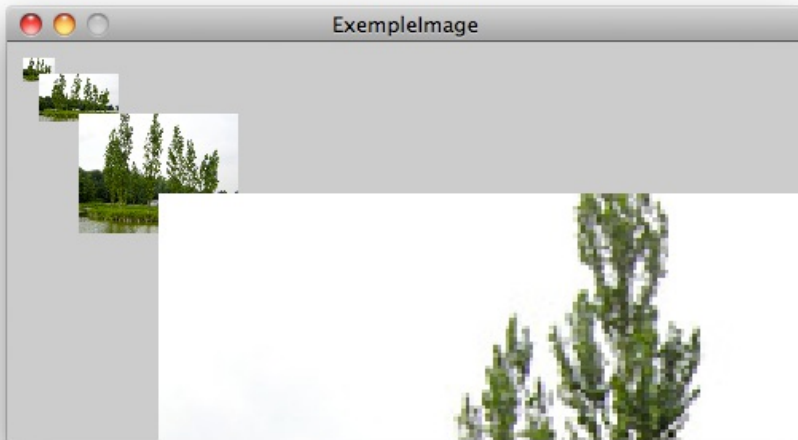
Jos ohjelmamme ei ole animoitu, kuten tässä tapauksessa, joudumme vain odottamaan, kun kuva latautuu Processingiin.

Varmista, että ohjelmasi ei lataa uusia kuvia käynnistyttyään, sillä muuten se pysähtyy aina ladatessaan. On olemassa muita tapoja tuoda kuvia tässä tapauksessa, mutta nämä tekniikat ovat liian kehittyneitä tähän kirjaan. Lisätietoa voit löytää Processing-foorumilta hakusanalla "thread".

MUUTA KOKOA

Lisäämällä kaksi parametriä voimme muuttaa kuvan kokoa. Tämä koko voi olla suurempi tai pienempi kuin alkuperäinen kuva, teoriassa rajaa ei ole olemassa. Kuitenkin kuvan alkuperäistä kokoa suurennettaessa Processing joutuu laajentamaan kuvan pikseleitä, mikä antaa pikselöityneen vaikutelman. Tämä tehoste voi usein olla käyttökelpoinen.

Muittaaksesi kuvan kokoa, voit lisätä kaksi parametriä kuvaasi, `{leveys, korkeus}`, mikä antaa meille 5 parametriä: `{kuvamuuttuja, x, y, leveys, korkeus}`.

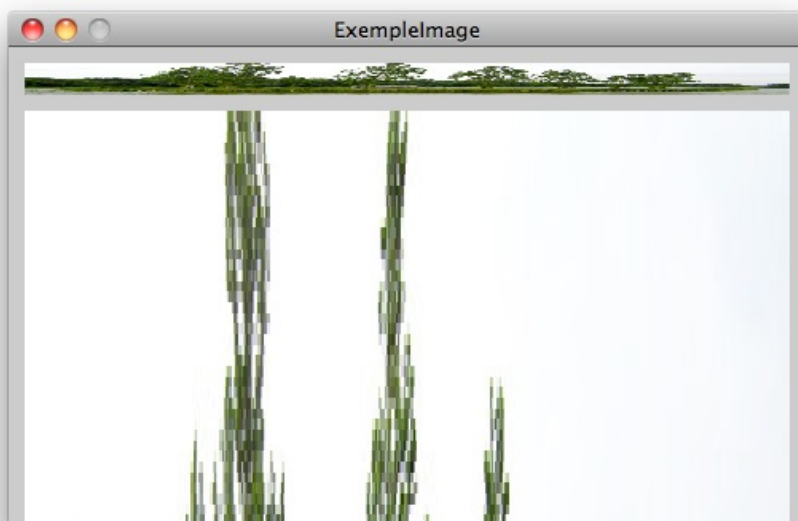


```
size(500,250);

PImage saari;
saari = loadImage("ile.jpg");

image(saari,10,10,20,15);
image(saari,20,20,50,30);
image(saari,45,45,100,75);
image(saari,95,95,1000,750);
```

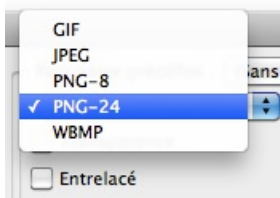
Huomaa, että olemme tuoneet vain kerran saaren kuvan muuttujaan, sitä voidaan käyttää nyt koko loppuohjelmassa. Huomaa myös, että olemme kunnioittaneet kuvan ulottuvuuksia $\{x,y\}$ kokoa muuttaessamme, mutta ne voisivat ottaa minkä tahansa arvon, joka venyttäisi kuvaa toisella näistä akseleista.



ALFA

Usein täytyy tuoda kuvia, jotka eivät ole neliön tai suorakulmion muotoisia, kuten esimerkiksi pienten kaksiulotteisten videopelien tapauksessa. Emme näe valkoista neliötä Pacmanin tai Marion ympärillä. Valitettavasti pikseleihin perustuvat kuvat täytyy nykyisin tuoda neliömuodossa. Tämän esteen kiertämiseksi jotkin tiedostomuodot käyttävät alfakanavan konseptia. Tämä asetetaan kuvan muodostavan punaisen, sinisen ja vihreän päälle. Tämä kerros osoittaa jokaisen pikselin läpinäkyvyyden asteen, PNG-kuvissa voi myös olla osittain läpinäkyviä pikseleitä.

Jos tahdot tallentaa kuvasi alfakanavan kanssa, sinulla on kolme vaihtoehtoa, jotka voit valita viedessäsi kuvaa valitsemallasi kuvankäsittelyohjelmalla:



Jokaisella ohjelmalla on tapa viedä kuvia. Sinun täytyy tietää, että esimerkiksi GIF, PNG-8 ja PNG-24 kuvien välillä on eroja, kuten alfakanavan käsittely. Näistä kolmesta turvallisim ja eniten vaihtoehtoja antava on PNG-24.

Tässä on kuva runoilija Lucretia Maria Davidsonista, jonka voit ladata osoitteesta

http://upload.wikimedia.org/wikipedia/commons/1/17/LucretiaMariaDavidson_transparent.png



Tämä kuva sopii täydellisesti tämän oppaan sivun pohjaan, koska siinä on ainoastaan alfakanava, joka kuvaa läpinäkyviä ja läpinäkymättömiä osia. Huomaa, että esimerkiksi Lucretian ympärillä olevat kuvat ovat läpinäkyviä, joten hänen otsansa ja niskansa ovat läpinäkymättömiä. Processing-kuvassamme näemme, että ne tosiaan ovat läpinäkymättömiä:



```
size(400,300);

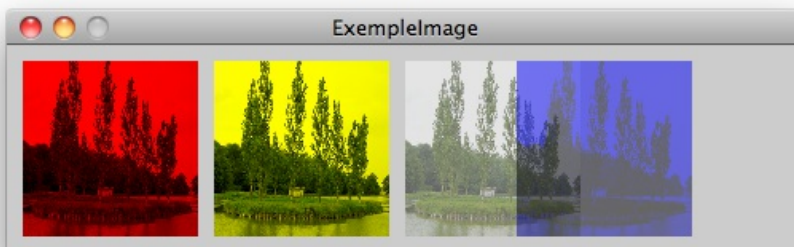
PImage saari = loadImage("ile.jpg");

PImage lucretia;
lucretia = loadImage("LucretiaMariaDavidson_transparent.png");

image(saari,0,0);
image(lucretia,20,20);
```

KUVIEN VÄRITTÄMINEN

Voit myös värittää kuvia. Toisin sanottuna voimme muuttaa pikselien väriä, joko kaikkien tai yksittäisten pikselien. Yksinkertaisin näistä menetelmistä tarkoittaa kaikkien pikselien värittämistä. Tämä väritysmenetelmä näyttää neliöiden värittämiseltä, mutta se vaatii uuden komennon, `tint()`, jolla se erotetaan neliön pikselien suorasta värittämisestä. Tässä tapauksessa `tint()` muuttaa jokaisen pikselin väriä, kun se piirretään piirustusavaruuteen.



```
size(500,130);
```



```
PImage saari;  
saari = loadImage("ile.jpg");  
  
tint(255,0,0);  
image(saari, 10,10, 110,110);  
  
tint(255,255,0);  
image(saari, 130,10, 110,110);  
  
tint(255,255,255,127);  
image(saari, 250,10, 110,110);  
  
tint(0,0,255,127);  
image(saari, 320,10, 110,110);
```

Kuten komennot `fill()` ja `stroke()`, komento `tint()` voit ottaa yhden, kaksi, kolme tai neljä arvoa, mikä riippuu siitä, mitä olemme tekemässä. Kun määrittelet kolme arvoa, voit esimerkiksi lisätä/vähentää punaisen, vihreän tai sinisen kerroksen intensiteettiä kuvassamme. Määrittämällä neljä arvoa voimme lisätä/vähentää kuvan läpinäkyvyyttä/läpinäkymättömyyttä.

9. TEKSTIT

Kirjoitamme nyt tekstiä kuvaamme. Kuvaan kirjoittaminen ei muistuta kirjoittamista perinteisessä merkityksessä, kuten esimerkiksi tekstinkäsittelyohjelmassa. Processingin kirjoittaminen muistuttaa enemmän graffitin kaltaista kirjaimien maalaamista. Ne päätyvät kaikki toistensa päälle. Siinä keskitytään enemmän kirjaimien ja sanojen ulkoasuun.

HEI, PROCESSING!

Kuten perusmuotojen tapauksessa, vain yksi komento tarvitaan piirrettäessä tekstiä kuvaan. Avaa uusi ikkuna Processingissa, kirjoita seuraava rivi, ja toista se:

```
text("Hei!", 10, 20);
```

Näet pienen ikkunan, joka on oletusarvoista kokoa 100x100 pikseliä, ja jossa lukee sana "Hei!" kirjoitettuna valkoisella:



Kuten nimestä voisi olettaa, komento `text()` piirtää tekstiä kuvaamme. Se vaatii kolme parametriä: {viesti, jonka tahdomme kirjoittaa},{koordinaatti x},{koordinaatti y}.

Tämä teksti voidaan myös värittää, aivan kuin mikä tahansa geometrinen muoto, vaihtamalla täyttöväriä:

```
fill(0);  
text("Heippa", 10, 20);
```

```
fill(255);  
text("koko", 10, 40);
```

```
fill(255,0,0);  
text("maailma!", 10, 60);
```

Tämän ohjelman pitäisi antaa seuraava tulos, jossa jokainen neljästä väristä on kirjoitettu omalla värillään:



Määritä vain väri, piirrä sen jälkeen teksti tuolla värillä. Voit tietenkin valita vain yhden värin ja kirjoittaa monta viestiä samalla värillä, tai piirtää värien yhdistelmällä, kuten tässä esimerkissä.

KASAAMINEN

Kuten tämän luvun johdannossa on mainittu, sanoja käytetään kuin piirroksessa, ei samalla tavalla kuin tekstinkäsittelyohjelmassa. Emme voi valita tai muokata tekstiä, elleimme kokonaan poista sitä ja kirjoita sen päälle.

Tässä on esimerkki periaatteesta:

```
fill(0);  
text("sana voi", 10, 20);  
text("olla piilossa", 10, 40);
```

```
fill(204);  
rect(28,25,50,20);
```

```
fill(0);  
text("toinen", 10, 60);
```

```
fill(255, 255, 255);  
text("toinen", 11, 61);
```

Olemme kirjoittaneet neljä viestiä, joskus samalla värillä, joskus toisella värillä. Huomaa, että sana "piilossa" on juuri piilotettu täyttövärillä, jolla on sama arvo kuin taustavärillä.



Kuten missä tahansa suunnittelussa, toimenpiteiden järjestys on tärkeä määriteltäessä lopullinen kuva. Tämä periaate pätee myös Processing-ohjelman ulkopuolella: "pukeudu, mene ulos, mene töihin" ei anna samaa tulosta kuin "mene ulos, mene töihin, pukeudu."

INTERAKTIIVISUUS

10. VIDEO

11. VIDEON TUOMINEN

12. MIKROFONIN SISÄÄNTULO

13. HIIREN TAPAHTUMAT

14. NÄPPÄIMISTÖN TAPAHTUMAT

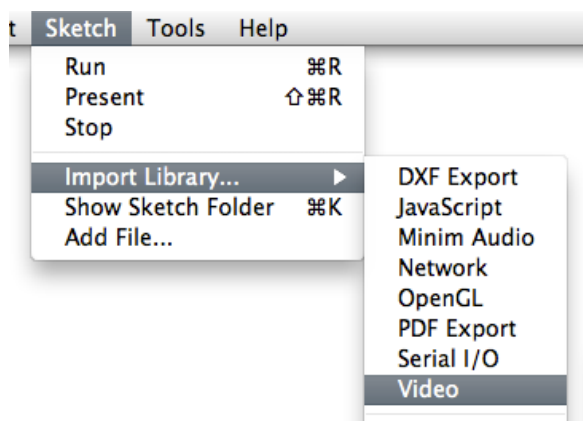
15. TULOSTAMINEN

10. VIDEO

Processing antaa sinun viedä animoidun tai interaktiivisen ohjelman QuickTime -formaattissa olevana videoanimaationa. Luodaksemme videoita voimme käyttää ennalta määriteltyä MovieMaker-objektia kirjastosta org.processing.video. Linux-käyttäjien tulee huomata, että tätä ominaisuutta ei ole käytettävissä.

MOVIEMAKERIN TARKOITUS

MovieMaker-objekti on Processingin ennalta määritelty luokka, joka luo videotiedoston, ja johon liitetään kuvia, kun ohjelma suoritetaan. Käyttääksesi sitä ohjelmassasi, tuo videokirjasto Processingiin:



Julistamme ohjelmamme alussa muuttujan, joka varastoi video-objektimme. Ohjelman täytyy luoda objekti `setup()` -osassa antamalla sille joukko parametrejä:

- Viittaus ohjelmaan
- Videon leveys
- Videon korkeus
- Ulostulotiedoston nimi

```
import processing.video.*;

MovieMaker mm; // Luo muuttuja objektiamme varten

void setup() {
  size(size(300,300);

  /*
   * Luodaan objekti MovieMaker.
   * Video on samaa kokoa kuin meidän kuvamme.
  */
  mm = new MovieMaker(this, width, height, "Video.mov");
}
```

LISÄÄ KUVIA

Jokainen metodin `draw()` kutsu joutuu kertomaan `MovieMaker`-objektille, että nyt ruudulla oleva kuva pitää tallentaa videotiedostoon.

```
void draw() {
    ellipse(mouseX,mouseY,20,20); //Ellipsin luominen
    mm.addFrame(); //Lisätään kuva videoon
}
```

VIDEON VIIMEISTELY

Tässä vaiheessa videon nauhoitus loppuu, kun lähdet pois ohjelmasta. Voit käskä ohjelman `MovieMaker` lopettaa nauhoittaminen milloin tahansa. Lisäämme toiminnon, joka lopettaa videon nauhoittamisen, kun painat jotain nappia.

```
void keyPressed() {
    mm.finish();
}
```

MUOKKAA REKISTERÖINTIÄ

Oletusarvoisesti `MovieMaker`-objekti tekee pakkaamatonta videota 15 ruutua sekunnissa. Interaktiivisille ohjelmille tämän raportointitavan pitäisi olla liian raskas. Voit määritellä pakkausformaatin ja muuttaa ruutujen määrän luodessasi objektin `MovieMaker`.

Parametrit ovat seuraavat:

- Viittaus ohjelmaan
- Videon leveys
- Videon korkeus
- Ulostulotiedoston nimi
- Ruutujen määrä sekunnissa
- Pakkauskoodi
- Laatu

```
mm = new MovieMaker(this, width, height, "video.mov", 30,
MovieMaker.H263, MovieMaker.HIGH);
```

(Luettavuuden vuoksi koodi on jaettu kahdelle riville.)

Näillä parametreilla videon nauhoittaminen on jouheampaa.

Tässä on joitain mielenkiintoisia videokoodikkeja:

- `MovieMaker.H264`: Hidas enkoodaus ja dekoodaus. Käyttää hyvin vähän tilaa.
- `MovieMaker.MOTION_JPEG_B`: Hyvin nopea enkoodaus ja dekoodaus. Käyttää vähän tilaa.
- `MovieMaker.ANIMATION`: Tukee alphanavaa. Käyttää paljon levytilaa, koska se on pakattu tappiottomasti.

11. VIDEON TUOMINEN

On mahdollista kaapata kuvia videokamerasta. Tätä kuvaa voit katsoa tai editoida.

TUETUT KAMERATYYPIT

Kamerat ovat helppokäyttöisiä USB-kameroita, joita kutsutaan myös verkkokameroiksi. On myös mahdollista käyttää kameraa, joka on kytketty Firewire-porttiin 1394. Tähän lasketaan DC1394 -kamerat ja DV -kamerat.

Processing-kirjaston käyttämä videokirjasto on QuickTime, jonka on tehnyt Apple. Kaikki QuickTimen tukemat kamerat toimivat. Apple tarjoaa tämän kirjaston vain Mac OS X ja Windows -käyttöjärjestelmiin. GNU/Linuxissa on parempi käyttää GSVideo-kirjastoa. Se käyttää GStreamer-kirjastoa erinomaisen hyvin.

VIDEON KAAPPAAMINEN KIRJASTOLLA

Tässä kerrotaan videokirjaston käytöstä. Kuten yllä on ilmaistu, kirjasto toimii vain käyttöjärjestelmissä Mac OS X ja Windows. Ensin meidän täytyy tuoda videokirjasto: **Sketch> Import Library > ... > Video.**

```
import processing.video.*;
```

Tämän jälkeen julistamme muuttujan, joka tallentaa objektin Camera. Valitsemme tätä käyttävän kameran. Kameran täytyy olla kytkettynä koneeseen, jotta voimme käyttää tätä kirjastoa. Jos koodi antaa virheviestin kun luomme kameraa, kokeile muita kameranumeroita.

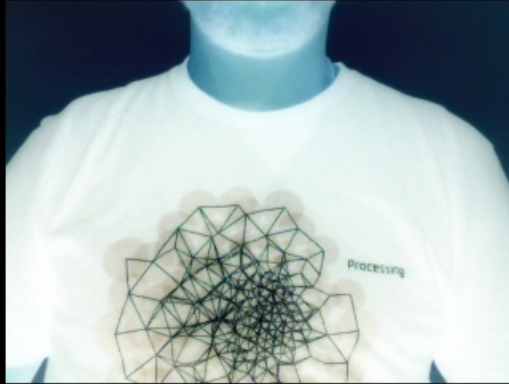
```
Capture camera;
```

```
void setup() {  
  size(640, 480);  
  background(0);  
  
  String[] devices = Capture.list();  
  println(devices);  
  
  camera = new Capture(this, 320, 240, devices[0]);  
}
```

Näytämme draw() -metodilla viimeisen kamerasta saadun kuvan ainoastaan kuvan ollessa uusi. Kuvat saadaan videokameralta tietyllä taajuudella, esimerkiksi 30 kertaa sekunnissa, mikä ei välttämättä vastaa ohjelmamme nopeutta.

```
void draw() {  
  if (camera.available()) {  
    camera.read();  
  
    camera.filter(INVERT);  
    image(camera, 160, 100);  
  }  
}
```


Voit käyttää videokaappaukseen kuvatyyppejä PiMag. Voimme täten lukea kuvan pikselit, muokata ne, näyttää ruutua muutaman kerran, ja monia muita asioita.



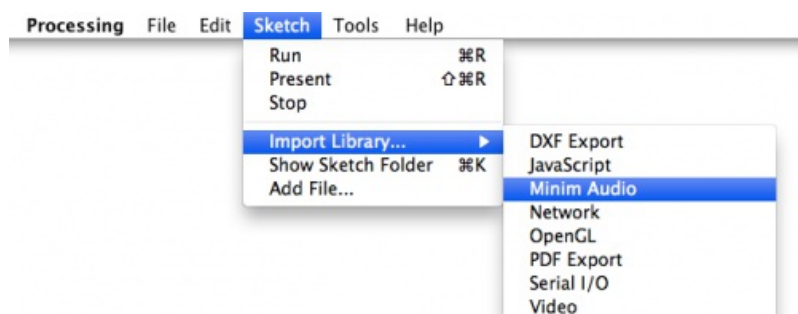
12. MIKROFONIN SISÄÄNTULO

Äänilähteet ovat helppo tapa lisätä interaktiivisuutta näppäimistön ja hiiren lisäksi. Ne tarjoavat käyttäjälle alkuperäisen lähteen, jota voi käyttää interaktiivisuuteen, ja joka tarjoaa erilaisen suhteen tietokoneeseen ja ohjelmaan. Käyttäjä voi olla fyysisemmin mukana ja käyttää ohjelmaasi epätavallisella tavalla. Mikrofonista tuleva ääni voidaan soittaa ja äänenvoimakkuutta ja muita muuttujia voidaan analysoida.

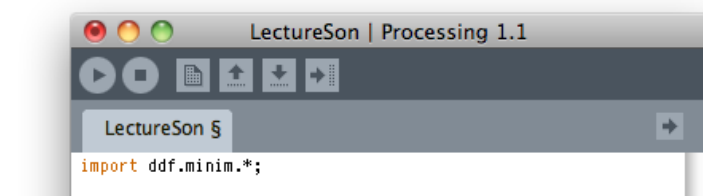
Processing voi käyttää kirjastoa Minim sound.

MINIM

Jossain vaiheessa Processingin jakeluversioon liitettiin Minim-äänikirjasto, joka soittaa ääntä ja kaappaa mikrofonista tulevat äänet. Tämä kirjasto on jo asennettu Processingiisi. Tarkasta sen olemassaolo hakemistosta **Sketch menu > Import Library > ... > Minim audio**.



Riippuen alustasta ja ohjelmaversiosta Processing lisää tätä toimintoa varten ohjelman alkuun enemmän tai vähemmän koodirivejä. Tässä tapauksessa tarvitsemme ainakin seuraavan komennon:



```
import ddf.minim.*;
```

Tämä on komento, jolla voit tuoda kaikki Minim-kirjaston ominaisuudet ohjelmaasi. Tämän tuontilauseen `ddf.minim.*` avulla voimme soittaa ohjelmassamme ääntä.

MINIMIN ASETUKSET

Kun laukaisemme ohjelman Run-napilla, äänikirjasto Minim käynnistyy. Tämä esimerkki voi vaatia pääsyä äänikortille käyttöjärjestelmässä, jotta ääniaaltoja voidaan luoda. Tämä merkitsee, että ohjelman lopussa Minim pitää ottaa pois päältä.

Ei haittaa, vaikka et vielä ymmärrä kaikkea sanomaamme. Tämä ei ole liian vakavaa. Tiedä vain, että seuraava koodi tarvitaan jokaisen Minimä käyttävän ohjelman alkuun:

```
import ddf.minim.*;

Minim minim;

void setup() {
  minim = new Minim(this);
}
```

Huomaa myös, että tämä koodi tarvitaan jokaisen Minimä käyttävän ohjelman loppuun:

```
void stop() {
  minim.stop();
  super.stop();
}
```

LISÄÄ ÄÄNILÄHDE

Lisätäksesi mikrofoniin sisääntulon meidän täytyy luoda ääniobjekti, joka pääsee mikrofoniin tuottamaan ääneen.

```
import ddf.minim.*;

Minim minim;
AudioInput in;

void setup() {
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, 512);
}
```

Sen täytyy myös lopettaa äänen kaappaus ja Minim-kirjasto ohjelman lopussa. Me käytämme metodia stop ().

```
void stop() {
  in.close();
  minim.stop();
  super.stop();
}
```

KATSO ÄÄNTÄ

Aiomme muuttaa ohjelmamme taustaväriä. Kun ääni on voimakkaampi, taustaväri (background) muuttuu valkoisemmaksi. Käytämme äänen sisääntuloa (in). Sillä on kolme aliobjektia: left, right ja mix. Stereomikrofoniin tapauksessa ne merkitsevät vasenta kanavaa, oikeaa kanavaa ja niiden välimuotoa. Näillä kanavilla voimme käyttää metodia level(), joka palauttaa äänen.

```
void draw() {
  background(in.mix.level()*2550);
}
```



PIENI PELI

Meidän pitää nyt näyttää, kuinka voimme luoda hyvin yksinkertaisen pelin, jossa ohjaukseen käytetään mikrofonia. Peli koostuu pallosta, joka on ruudun oikealla puolella. Se menee hitaasti mutta varmasti vasemmalle. Pelaajan ääni painaa palloa takaisin oikealle. Tavoitteena on päästä ääriviivan yli.

Aluksi muutamme ruudun kokoa, laitamme smoothin päälle, ja määrittelemme ääriviivat valkoisiksi.

```
void setup() {  
  size(600, 100);  
  smooth();  
  stroke(255);  
  
  minim = new Minim(this);  
  in    = minim.getLineIn(Minim.STEREO, 512);  
}
```

Voimme luoda muuttujan, joka tallentaa pallon sijainnin. Se julistetaan ohjelman alussa, jolloin se on saatavilla kaikkialla. Laitamme sille arvon 0.

```
float ballX = 0;  
  
void setup() {  
  ...  
}
```

Metodissa draw() määrittelemme mustan taustan ja piirrämme pallon, joka reagoi ääneen. Jokaisessa draw() -metodin kutsussa lisäämme mikrofonin äänitason pallon x-koordinaattiin. Se alkaa liikkua vähitellen, kun teemme ääntä.

```
void draw() {  
  background(0);  
  
  ballX = ballX + in.mix.level()*20;  
  
  ellipse(25 + ballX, height - 25, 50, 50);  
}
```



Estääksesi palloa lähtemästä ruudulta lisäämme kaksi ehtoa, jotka korjaavat tilanteen, jos muuttuja ballX on pienempi kuin 0 tai suurempi kuin ruudun leveys.

```
void draw() {  
  background(0);  
  
  ballX = ballX + in.mix.level()*20;
```

```

    if (ballX < 0) {
        ballX = 0;
    }

    if (ballX > width-25) {
        ballX = width-25;
    }

    ellipse(25+ballX, height-25, 50, 50);
}

```

Lisäämme ehdon, jonka mukaan pallon väri muuttuu, jos se menee rajan yli.

```

void draw() {
    background(0);

    ballX = ballX + in.mix.level()*20;

    if (ballX < 0) {
        ballX = 0;
    }

    if (ballX > width-25) {
        ballX = width-25;
    }

    if (ballX > 500) {
        fill(255, 0, 0);
    } else {
        fill(255);
    }

    line(500, 0, 500, 100);
    ellipse(25+ballX, height-25, 50, 50);
}

```

Jotta pelistä tehtäisiin monimutkaisempi, teemme ehdon, jolla pallo menee takaisin pysyvästi. Jokaisella komennon draw() kutsulla pienennämme hievan sen sijaintia x-akselilla.

```

void draw() {
    background(0);

    ballX = ballX - 0.5;    ballX = ballX + in.mix.level()*20;

    if (ballX < 0) {
        ballX = 0;
    }

    if (ballX > width - 25) {
        ballX = width - 25;
    }

    if (ballX > 500) {
        fill(255, 0, 0);
    } else {
        fill(255);
    }

    line(500, 0, 500, 100);
    ellipse(25 + ballX, height - 25, 50, 50);
}

```

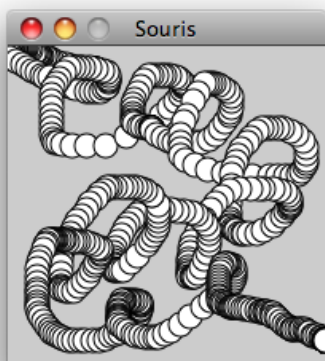
Nyt voit kuvitella muita tapoja käyttää ääntä projekteissasi. Voit analysoida äänen taajuutta, tehdä synteesin, luoda ääniefektejä mittausta varten, ja monta muuta asiaa.

13. HIIREN TAPAHTUMAT

Tässä luvussa näemme ohjelman vuorovaikutuksen hiiren kanssa. Processing kerää hiireltä tietoja, kuten hiiren sijainnin ja napsautukset. Luomme pienen ohjelman, joka perustuu ympyröihin, jolla esittelemme satunnaiset liikkeet.

HIIREN SIJAINTI

Hiiren koordinaatit ikkunassa nähdään kahdella muuttujalla: `mouseX` ja `mouseY`. Niillä voimme tietää hiiren sijainnin suhteessa piirrosikkunamme, alkaen ikkunan vasemmasta yläkulmasta. Seuraavassa esimerkissä luomme ohjelman, joka piirtää ympyrän hiiren sijaintiin.



```
void setup() {  
  size(200,200);  
  smooth();  
}  
  
void draw() {  
  ellipse(mouseX,mouseY,15,15);  
}
```

Muutamme edellisen esimerkin muuttamalla dynaamisesti ympyrän läpimittaa. Se valitaan satunnaisesti metodilla `random()`, joka luo satunnaisen numeron tietyllä arvovälillä. Kun Processing tekee komennon `execute()`, muuttuja `r` täytetään satunnaisella numerolla väliltä 3-30: 12, 25, 23, 11, 22, 4, 10, 11, 25... Ympyrän läpimitan muutos johtaa ympyrän välkkymiseen hiiren kursorin ympärillä.



```
void setup() {
  size(300,300);
  smooth();
}

void draw() {
  float r = random(3,30);
  ellipse(mouseX,mouseY,r,r);
}
```

Kun hiiri lähtee ikkunasta, hiiren sijaintia ei enää välitetä ohjelmallemme. Ympyrät piirretään ja ne kerääntyvät hiiren viimeiseen sijaintiin.

NAPSAUTUKSET

Voimme vastaanottaa hiiren napsautuksia käyttäen metodeita `mousePressed()` ja `mouseReleased()`.

Molemmat metodit sallivat meidän tietää koska hiiren nappia painetaan ja koska se on vapaana. Edellisen esimerkin mukaan voimme muuttaa ympyrän täyttöväriä sen mukaan painaako käyttäjä hiiren nappia. Valitsemme harmaan sävyn satunnaisesti komennolla `fill()`, kun hiiren nappia painetaan.



```
float ton_de_gris = 255;
```

```

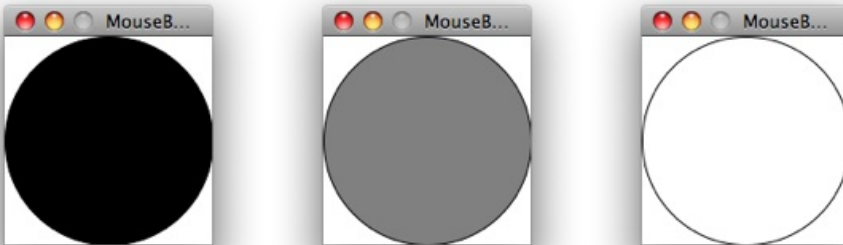
void setup() {
  size(300, 300);
  smooth();
}

void draw() {
  float r = random(10, 80);
  fill(grey);
  ellipse(mouseX, mouseY, r, r);
}

void mousePressed() {
  grey = random(255);
}

```

Processingin avulla voimme tietää mitä hiiren nappia painettiin. Tähän käytämme muuttujaa `MouseButton`, joka tunnistaa painetaanko vasenta, oikeaa tai keskimmäistä hiiren nappia (`LEFT`, `RIGHT`, `CENTER`). Käytä tätä muuttujaa piirtääksesi ympyrän, joka vaihtaa väriä riippuen siitä mitä hiiren nappia painetaan.



```

void setup() {
  size(200,200);
  smooth();
}

void draw() {
  background(255);
  ellipse(100,100,200,200);
}

void mousePressed() {
  if (mouseButton == LEFT) fill(0);
  if (mouseButton == RIGHT) fill(255);
  if (mouseButton == CENTER) fill(128);
}

```

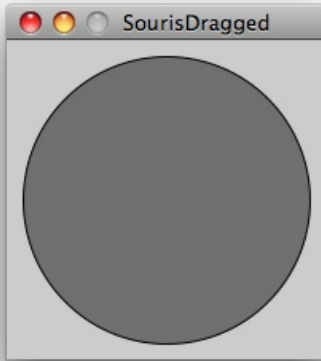
MUUT TAPAHTUMAT

Processing kaappaa kaksi muuta hiiren tapahtumaa täsmälleen sen sijainnissa ikkunan yläpuolella.

`mouseMoved()` havaitsee hiiren liikkeen kun se liikkuu piirrosikkunan yläpuolella. Jos hiiren osoitin poistuu ikkunan alueelta tai lopettaa liikkeen tämä metodi ei toimi.

`mouseDragged()` kutsutaan, kun käyttäjä napsauttaa nappia ja liikuttaa hiirtä ikkunan päällä. Sen ominaisuuksiin kuuluu hiiren vetämisen hallinta. `mouseDragged()` toimii, vaikka hiiren osoitin lähtisi ulos ikkunasta.

Seuraavassa esimerkissä käytämme näitä metodeja. Kun `mouseMoved()` kutsutaan, muutetaan ympyrän täyttöväriä, ja kun `mouseDragged` kutsutaan, muutetaan ympyrän kokoa.



```
int r = 100;
int c = 100;

void setup() {
  size(255, 255);
  smooth();
}

void draw() {
  background(255);
  fill(c);
  ellipse(width/2, height/2, r, r);
}

void mouseMoved() {
  c = mouseY;
}

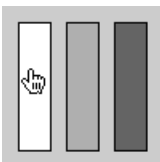
void mouseDragged() {
  r = mouseX;
}
```

KOHDISTIN

Joskus on hyvä kätkeä hiiren kohdistin. Tässä on käsky kohdistimen piilottamiseksi:

```
noCursor();
```

On myös mahdollista muuttaa kursorin muotoa osoittamaan eri tapahtumia käyttäjälle. Tähän käytetään komentoa `cursor()` parametreilla `ARROW`, `CROSS`, `HAND`, `MOVE`, `TEXT`, `WAIT`.



```
void setup() {
  fill(255);
```

```

    rect(10, 10, 20, 80);
    fill(175);
    rect(40, 10, 20, 80);
    fill(100);
    rect(70, 10, 20, 80);
}

void draw()
// Katsomme voitko lennättää yhtä kolmesta neliöstä // Ja
muutamme kursoria vastaavasti
if (mouseX > 10 && mouseX < 30 && mouseY > 10 && mouseY < 90) {
    cursor(HAND); // Näyttää käden
}
else
if (mouseX > 40 && mouseX < 60 && mouseY > 10 && mouseY < 90) {
    cursor(ARROW); // Näyttää nuolen
}
else
if (mouseX > 70 && mouseX < 90 && mouseY > 10 && mouseY < 90) {
    cursor(WAIT); // Näyttää tiimalasin
}
else {
    cursor(CROSS); // Näyttää ristin jos emme tee mitään
}
}

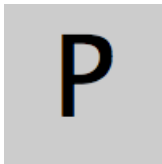
```

14. NÄPPÄIMISTÖN TAPAHTUMAT

Näppäimistön tapahtumat kaappaamalla ohjelma voi vastata käyttäjän toimenpiteisiin. Näytämme tässä kappaleessa kuinka haemme tiedon, joka liittyy näppäimistöön, ja me luomme yksinkertaisen kirjoituskoneen, jolla voimme kirjoitella tekstiä ruudulle.

Viimeksi painetun näppäimen arvo tallennetaan muuttujaan, joka julistetaan Processingissa. Tämä muuttuja voi tallentaa vain yhden kirjaimen, jota symbolisoi esimerkiksi 'a', 'b', 'c'... Tässä tapauksessa suuri kirjain on eri kirjain kuin pieni kirjain.

Seuraavassa esimerkissä näytämme ikkunassa merkin, joka vastaa painettua näppäintä. Tähän käytämme metodia text().

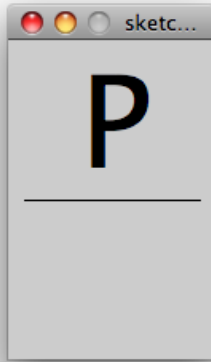


```
void draw() {  
  background(204);  
  fill(0);  
  textSize(70);  
  text(key, 30, 70);  
}
```

Voimme kaapata ohjelmaamme, kun käyttäjä on painanut tai vapauttanut näppäimen näppäimistöllä, metodeilla keyPressed() ja keyReleased(). Nämä kaksi metodia Processing kutsuu automaattisesti, kun näppäimistön tila muuttuu.

Seuraavassa esimerkissä keyPressed() ja KeyReleased() muuttavat muuttujan y arvoa, jolla kirjain sijoitetaan ruudulle.

Listen
Read phonetically



```
int y = 0;

void setup() {
  size(130,200);
  textSize(80);
  stroke(0);
  fill(0);
}

void draw() {
  background(204);
  line(10,100,120,100);
  text(key,35,y);
}

void keyPressed(){
  y = 180;
}

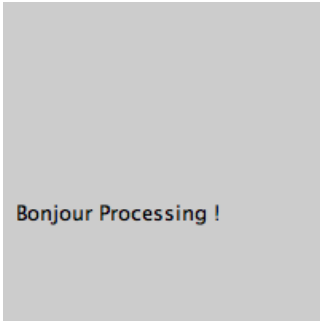
void keyReleased(){
  y = 80;
}
```

NÄPPÄINKOMENNOT

Erikoisnäppäimet kuten nuolet (YLÖS, ALAS, VASEMMALLE, OIKEALLE) tai ALT, CONTROL, SHIFT tallennetaan muuttujaan `keyCode`. Testi `if (key==CODED)` kertoo onko painettu näppäin erikoisnäppäin vai eikö. Ohjelmassa meidän täytyy erottaa suuret ja pienet kirjaimet riippuen näppäimestä, jota tahdot testata.

Seuraavassa esimerkissä luomme erityisen kirjoituskoneen, jossa on mahdollista liikuttaa tekstiä nuolilla.

Aina kun näppäintä on painettu, se tallennetaan merkkijonoon, joka näytetään `draw()` -komennolla käyttäen metodia `text()`.



Bonjour Processing !

```
String s = "";
int x = 50;
int y = 50;

void setup() {
  size(200,200);
}

void draw() {
  background(255);
  fill(0);
  text(s, x, y);
}

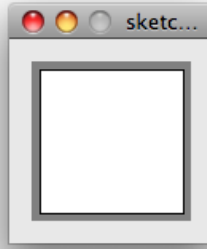
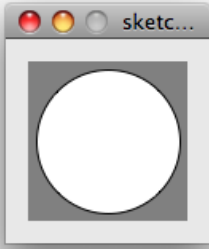
void keyPressed() {
  if (key == CODED){
    if (keyCode == LEFT)   x = x - 1;
    if (keyCode == RIGHT)  x = x + 1;
    if (keyCode == UP)     y = y - 1;
    if (keyCode == DOWN)   y = y + 1;
  }
  else {
    s = s + key;
  }
}
```

Huomaa aaltosulkujen käyttö jokaisen if-komennon jälkeen. Aaltosulut voidaan jättää pois, jos syntaksi toimii vain yhden komennon kohdalla. Käytämme myös String-tyyppistä muuttujaa, joka tallentaa sarjan merkkejä, ja joka on erittäin hyvä esimerkiksi lauseiden tallentamiseen.

Processing määrittelee myös muuttujan keyPressed.

Varoitus! Vaikka syntaksi on identtinen, sitä ei pidä sekoittaa metodiin keyPressed(), jolla on sama nimi. Tämä muuttuja kertoo meille aina, jos näppäintä painetaan, tai jos sitä ei voida käyttää metodissa draw().

Seuraavassa esimerkissä piirrämme ympyrän, jos painettu näppäin on 'c', neliön jos painettu näppäin on 'r'.



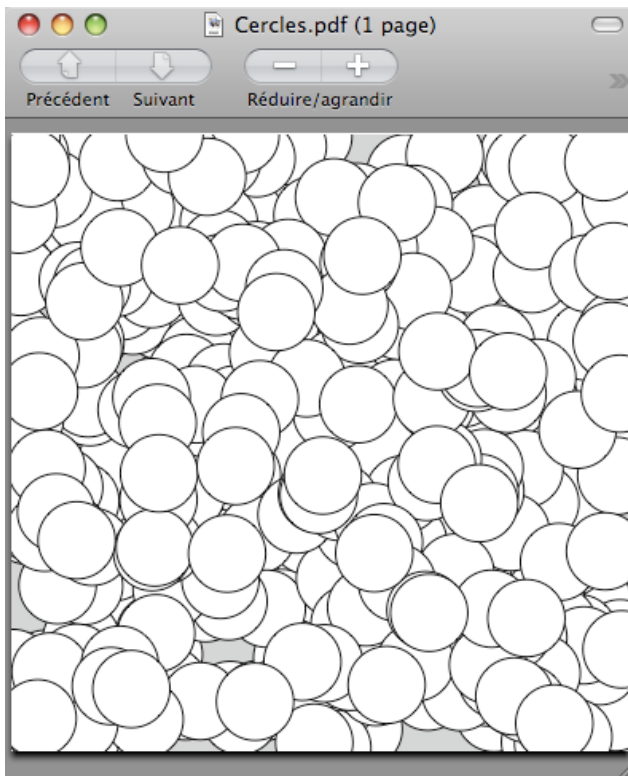
```
void draw()
{
  rectMode(CENTER);
  background(128);
  if (keyPressed == true)
  {
    if (key == 'c') {
      ellipse(50,50,90,90);
    }
    else if (key == 'r') {
      rect(50,50,90,90);
    }
  }
}
```

15. TULOSTAMINEN

Tähän asti olemme työskennelleet ruudulla luoden kuvia ja animaatioita. Nyt keskustelemme tulostamisesta: teemme PDF-tiedoston, jonka tulostamme paperille.

SUORA TILA

Tässä esimerkissä luomme 500 satunnaista ympyrää ja piirrämme ne PDF-dokumenttiin, joka tallennetaan työkansioomme.

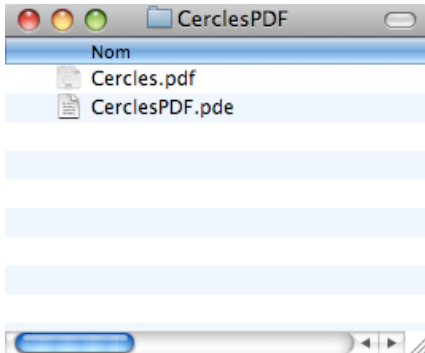


```
import processing.pdf.*;
size(400, 400, PDF, "Pallot.pdf");
for (int i = 0; i < 500; i++)
  ellipse(random(width), random(height), 50, 50);
exit();
```

Tässä esimerkissä aloitamme tuomalla kaikki luokat kirjastosta `processing.pdf`. Käytämme metodia `size()`, joka ottaa neljä parametriä. Huomaat varmaan tässä avainsanan, joka kertoo Processingille, että piirrämme dokumentin "`Pallot.pdf`", joka on metodin neljäs parametri.

Tällä metodilla emme tarvitse ikkunoita, koska voimme piirtää suoraan dokumenttiin emmekä ruudulle. Tämä ohjelma päättyy komennolla `exit()`, joka käskää Processingin lopettamaan ohjelmaa itselleen, kunhan olemme painaneet käyttöliittymän **Stop**-nappia.

Luotu dokumentti on ohjelmasi hakemistossa. Voit avata sen kaksoisnapsauttamalla.

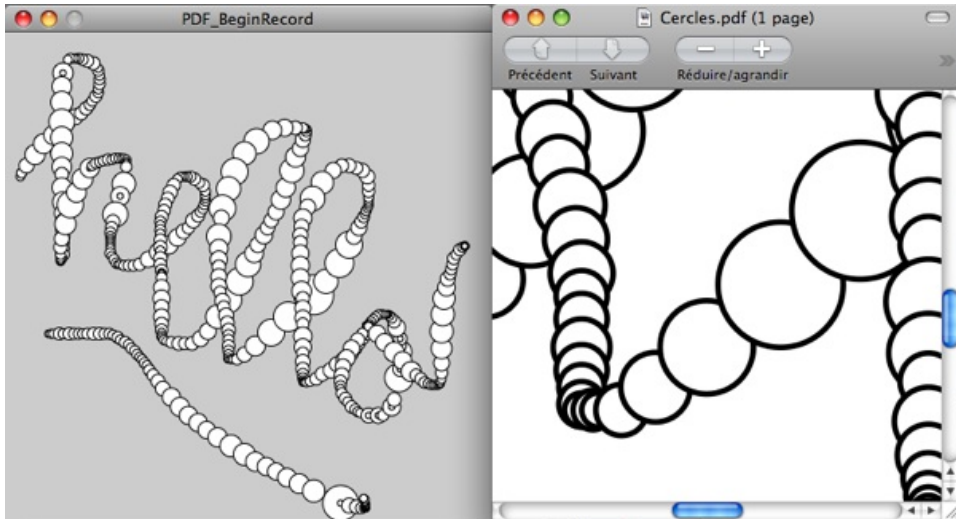


NAUHOITUSTILA

Voimme nyt nähdä, kuinka voit piirtää sekä ruudulle että tiedostoon. PDF-tiedostoon piirretään komennoilla `beginRecord()` ja `endRecord()`. Molemmat menetelmät antavat meidän aloittaa ja lopettaa grafiikkakomentojen piirtäminen PDF-tiedostoon.

Kuten aiemminkin, tuloksena oleva tiedosto on tallennettu ohjelman kansioon.

Seuraavassa esimerkissä luomme graafisen työkalun, jonka siveltimen koko riippuu hiiren nopeudesta. Ohjelman käynnistyessä metodi `setup()` aloittaa nauhoittamisen komennosta `beginRecord()`, joka ottaa ensimmäisenä parametrinä viennin tyyppin ja tiedostonimen toisena parametrinä. Kun painat **VÄLITYÖNTIÄ**, nauhoitus loppuu ja ohjelma pysähtyy. Kuten aiemminkin, tiedosto tallennetaan ohjelman kansioon.



```
import processing.pdf.*;
boolean drag = false;

void setup()
{
  size(400, 400);
  smooth();
  beginRecord(PDF, "Pallot.pdf");
}

void draw() {
  if (drag)
  {
    float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;
    ellipse(mouseX, mouseY, r, r);
  }
}

void mouseDragged() {
  drag = true;
}

void mouseReleased() {
  drag = false;
}

void keyPressed() {
  endRecord();
  exit();
}
```

Seuraava rivi:

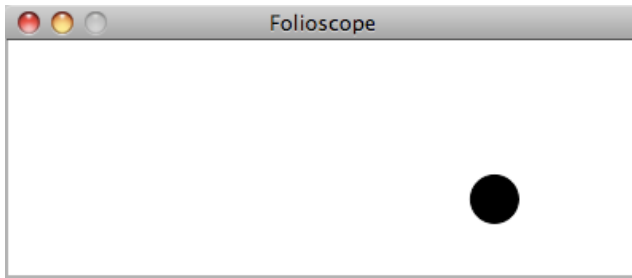
```
float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;
```

... voi laskea hiiren nopeuden käyttäen metodia `dist` ja muuttujia `pmouseX`, `mouseX`, `mouseY` ja `pmouseY`. Voimme lisätä 5, jotta meillä ei ole läpimitan 0 ympyröitä ruudulla.

Voimme käyttää muuttujaa `drag` ja piirtää käyttäjän tekemät ympyrät komennolla `draw()`.

KIRJAGENERAATTORI

Sen sijaan että loisimme yhden dokumentin, luomme monisivuisen kirjan. Luomme dynaamisesti käännettävät sivut (<http://fr.wikipedia.org/wiki/Folioscope>). Jokainen sivu näyttää kuvan animaation. Me käytämme metodia `nextPage()`, joka luo tyhjän sivun joka kerta kun metodia kutsutaan nauhoituksen aikana.



```
import processing.pdf.*;

PGraphicsPDF pdf;
float x = 0;
float y = 75;

void setup() {
  size(400, 150);
  smooth();
  pdf = (PGraphicsPDF) beginRecord(PDF, "Luoti.pdf");
}

void draw() {
  background(255);
  fill(0);
  ellipse(x, y, 30, 30);

  x = x + random(5);
  y = y + random(-5, 5);

  if (x - 30 > width) {
    endRecord();
    exit();
  }
  pdf.nextPage();
}
```

Olemme esitelleet muuttujan, joka varastoi pdf-dokumentin, jonka luomme. Kun olemme lopettaneet kuvan piirtämisen `draw()` -komennolla, siirrymme seuraavaan kuvaan metodilla `pdf.nextPage()`. Ohjelma pysäytetään ja voimme avata hakemistosta tiedoston `Luoti.pdf`.

PIKSELI VASTAAN VEKTORI

Kun piirrämme esimerkiksi neliön ruudulle, täytämme ruudun alueen pikselit tietyllä värillä. Jos tallennamme kuvan, voimme tallentaa jokaisen yksittäisen pikselin tietyssä kuvaformaattissa (JPEG, GIF, TIFF, jne.). Jos tahdomme laajentaa kuvan, tietokone ei kykene luomaan uusia pikseleitä vanhojen pikselien väliin ja lisäämään yksityiskohtia, koska se ei tallenna tätä tietoa tiedostoon.

Jos tallennamme saman kuvan vektoriformaatissa, Processing ei enää tallenna jokaista pikseliä erikseen, vaan pikemminkin jokaisen piirretyn muodon ominaisuudet: sen muodon, sijainnin ja värin. Joten kun laajennamme kuvaa, tietokone piirtää tarkasti kuvan ominaisuudet, joita ei olisi voitu laajentaa pikselöidystä kuvasta.

TUO PDF-OMINAISUUDET

Edellisessä esimerkissä käytimme kirjastoa `processing.pdf.*`, joka voi tuoda PDF-dokumentin luomiseen tarkoitetun kirjaston. Kirjasto on joukko ominaisuuksia, jotka laajentavat Processingin mahdollisuuksia aina käyttäessämme PDF-tiedostoja ohjelmassamme, jolloin käytämme **Import**-komentoa.



Jos yrität joskus käyttää PDF-viennin ominaisuuksia tuomatta kirjastoa, törmäät virheilmoituksiin, jotka ilmestyvät konsoli-ohjelmaan. Koodin kirjoittamisen lisäksi voit myös käyttää valikkoa **Sketch > Import Library > PDF Export**.

VIENTI

16. VIEMINEN

16. VIEMINEN

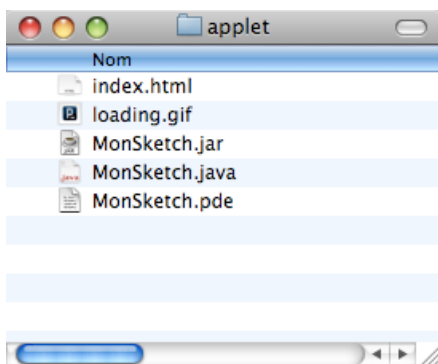
Toistaiseksi olemme työskennelleet ainoastaan Processing-ympäristössä. Ajamme koodirivejä ja suoritamme ohjelmat käyttöliittymän **Run**-nappulasta. Näemme, että Processingilla on kaksi tapaa viedä ohjelmat jaettavaksi. Yksi on animaatio, joka voidaan näyttää verkkoselaimessa, toinen luo ohjelman, joka voidaan ajaa Windowsissa, Macissa tai Linuxissa. Joka tapauksessa julkaisijan täytyy ajaa ohjelmasi.

VIEMINEN VERKKOON

Kun ohjelmasi toimii ja olet tyytyväinen, voit luoda sovelman, joka laitetaan verkkosivulle ja on selaimesi luettavissa. Tehdäksesi tämän voit painaa **Export**-nappia.



Tämä toimenpide avaa automaattisesti sovelman kansion. Siinä on monta tiedostoa.



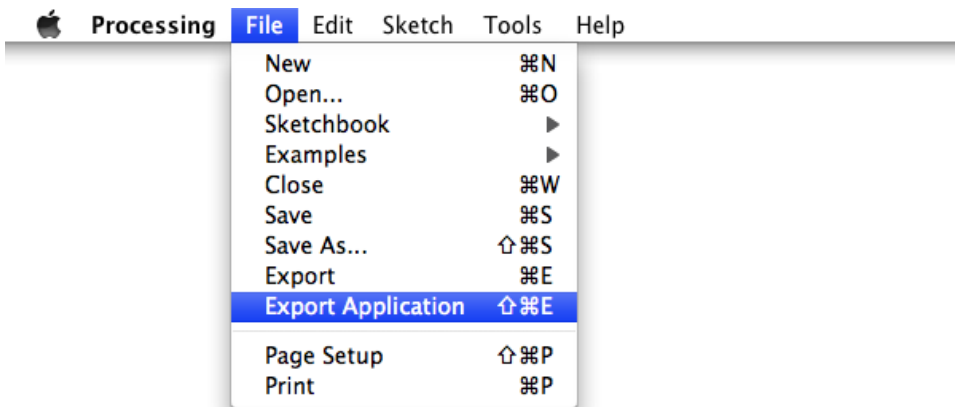
- **Index.html** on verkkosivu, joka on tehty Processingilla. Kaksoisnapsautuksella selaimesi laukaisee ja näyttää animaation, jonka olet juuri vienyt.
- **Loading.gif** on ikoni, jota käytetään ladattaessa sovelmaa verkkosivulle.
- **(ohjelman nimi).jar**. Jar on sovelma. Tämä on ohjelmasi versio, joka on muutettu Javaksi ja tallennettu kaiken tiedon kanssa.
- **(ohjelman nimi).java**. Java on ohjelmasi Java-versio.
- **(ohjelman nimi).pde**. Pde on lähdekoodi, jonka kirjoitit editoriin.

Varoitus! Jokainen Processingin vienti ylikirjoittaa tiedostoja. Jos teet muutoksia tiedostoon, joudut nimeämään tiedoston (ohjelman nimi).html.

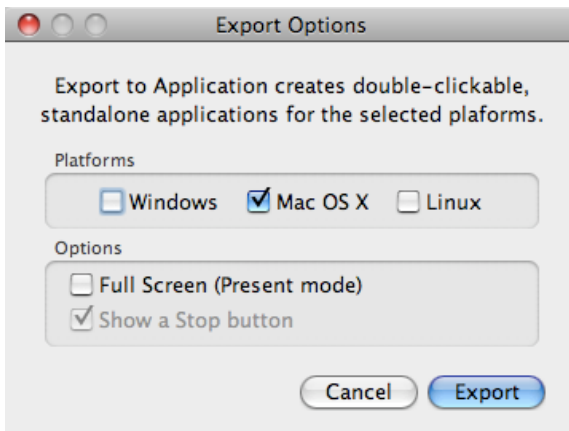
Jos sinulla on tilaa verkossa ja tunnet FTP:n käytön, voit ladata sovelman (nimeämällä sen uudestaan) verkkotilaasi ja katsoa ohjelmiasi verkossa.

OHJELMIEN VIEMINEN

Ohjelmiasi voidaan myös viedä Windowsiin, Maciin tai GNU/Linuxiin. Nyt voit kaksoisnapsauttaa laukaistaksesi ohjelman. Tehdäksesi tämän, napsauta valikkoa **File > Export Application**.

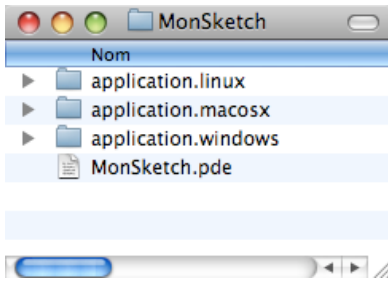


Seuraavaksi aukeaa **Export Options** -ikkuna.



- Kohdassa **Platforms** voit valita alustan ohjelmalle: Windows, Mac OS X tai Linux.
- **Full Screen (Present method)** avaa ohjelman koko ruudun kokoisena, kun kaksoisnapsautat sitä.
- **Show a Stop button** laittaa kokoruututilassa näkyville nappulan, joka lopettaa ohjelman ajamisen.

Seuraavaksi ohjelma ilmestyy näkyville jokaisen käyttöjärjestelmän mukaisissa kansioissa. Ohjelma on kansion sisällä, se on valmis ajettavaksi koneeltasi.



ANIMAATIO

17. ÄÄNENTOISTO

18. DRAW-METODI

19. OLIIDEN ANIMOINTI

20. USEAMMAN OLION ANIMOINTI

21. AIKAJANA

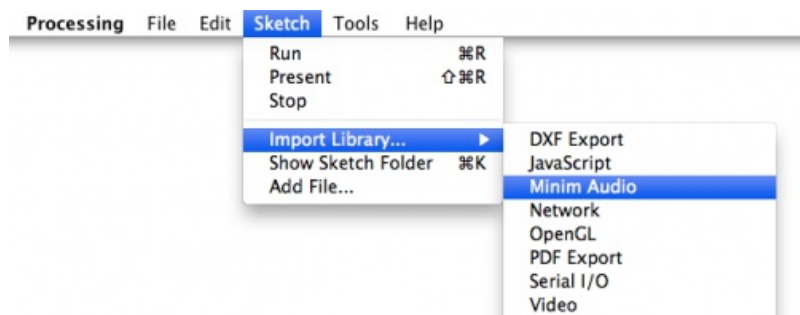
17. ÄÄNENTOISTO

Lähtökohtaisesti Processing on kuvan luomiseen tarkoitettu ympäristö. Sitä ei suunniteltu soittamaan ääntä ja vielä vähemmän kehittämään ääniaaltoja. Tätä varten on esimerkiksi Pure Data. Monet ulkoiset kirjastot on luotu juurikin laajentamaan äänenkäsittelymahdollisuuksia, ja voit liittää milloin tahansa nämä ulkoiset kirjastot ohjelmiisi: syntetisaattoreita, sampleita, audioliittymiä, MIDI-sekvenssereitä jne. Mutta tämä ei kuitenkaan ole Processingin perustarkoitus, joka on erilaisten visuaalisten muotojen luominen.

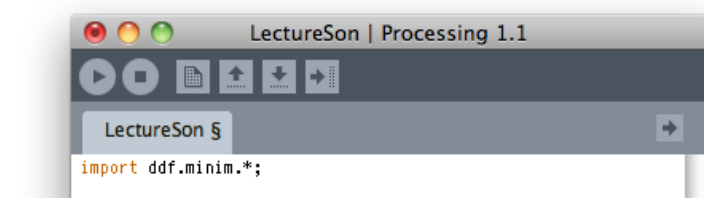
Kuitenkin ääni on olennainen osa interaktiivisuutta, ja on vaikea kuvitella tehokasta interaktiivisuustyökalua ilman ääntä.

MINIM

Processingin jakeluversioon päätettiin liittää Minim -äänikirjasto, joka soittaa ääntä ja osaa tallentaa mikrofonia tulevan äänen. Tämä kirjasto on jo asennettu koneellesi. Tarkastaaksesi sen asennuksen ja liittääksesi sen ohjelmaasi mene hakemistoon **Sketch > Import Library ... > Minim audio**.



Riippuen alustasta ja versiosta, Processing lisää tähän toimintoon enemmän tai vähemmän koodirivejä ohjelmasi alkuun. Tässä tapauksessa tarvitaan ainakin seuraava komento:



```
import ddf.minim.*;
```

Tämä komento tuo kaikki Minim-kirjaston toiminnot ohjelmamme käytettäväksi. Tämän ddf.minim.* -kirjaston avulla soitamme äänetä ohjelmassamme.

MINIM-ASETUKSET

Kun ajamme ohjelmamme **Run**-napin avulla, meidän täytyy aktivoida äänikone Minim. Tämän esimerkin pitäisi pyytää pääsyä äänikorttiin käyttöjärjestelmän kautta, jotta ohjelma voi soittaa ääniä. Tämä merkitsee, että ohjelman lopussa voimme myös sulkea Minim-kirjaston.

Ei haittaa, vaikka et vielä ymmärtäisi kaikkea sanomaamme. Olennaista on se, että jokaisen Minimä käyttävän ohjelman alussa tulee olla seuraava koodi:

```
import ddf.minim.*;

Minim minim;

void setup() {
  minim = new Minim(this);
}
```

Ohjelman loppuun tulee myös kirjoittaa muutama rivi:

```
void stop() {
  minim.stop();
  super.stop();
}
```

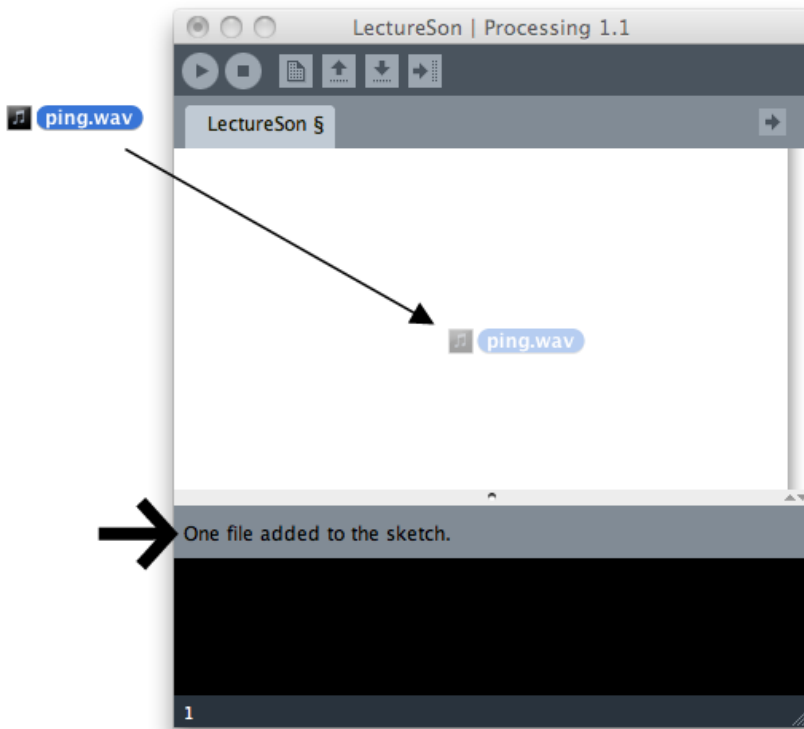
Mikä sana "this" on ohjelmassamme? Sana "this" tarkoittaa tätä ohjelmaa. Minim täytyy tietää tämän ohjelman osoite, jotta se voi kommunikoida ohjelman ja äänikortin kanssa. Tätä ei tarvitse ymmärtää vielä. Nämä ovat vain taikasanoja, jotka täytyy sanoa jokaisen ääntä käyttävän ohjelman alussa ja lopussa.

ÄÄNEN TUOMINEN

Tuomme kuvan ohjelmaamme aivan kuin kuvan tai kirjasintyyppin. Kannattaa tallentaa ohjelma ennen tätä askelta.

Ääni voi tulla monesta eri lähteestä: ne voidaan ladata netistä, tehdä Audacityllä, ja on jopa ääniä, jotka voivat olla osa käyttöjärjestelmäsi.

Etsi nyt äänitiedosto ja vedä se suoraan Processing-ikkunaan:



Tämä toiminto laittaa äänitiedoston hakemistoon "data" ohjelmassasi. Jos tahdot nähdä tämän tiedoston, paina **ctrl-k** (Windows / Linux) tai **cmd-k** (Mac), muuten voit valita valikosta **Sketch > Show Sketch Folder**.

ÄÄNITIEDOSTOT

Voit käyttää kolmea ääniformaattia Minimian kanssa: WAV, AIFF ja MP3. Kaksi ensimmäistä formaattia voidaan viedä oletusarvoisesti ilmaisella ohjelmalla, kuten Audacityllä. Kolmas formaatti vaatii lisäosan Audacityssä, mutta se on melko helppo asentaa.

Kuten kuvien tapauksessa, eri formaateilla on etunsa ja haittansa.

1. WAV ja AIFF -tiedostot ovat melkein identtisiä ja molemmat formaatit ovat pakkaamattomia. Niitä Minim voi lukea ilman suurempia hankaluuksia, koska niitä ei tarvitse purkaa. Niitä käytetään usein lyhyisiin ääniin, kuten ääniefekteihin. Molemmat formaatit ovat paljon isompia kuin vastaavan pituiset MP3-tiedostot.
2. MP3-formaatti on pakattu formaatti. Se on usein paljon lyhempi kuin WAV ja AIFF. MP3-toisto vaatii enemmän tietokojenkäsittelytehoja koneelta, koska tiedosto täytyy purkaa, kun se soitetään.

KELLO

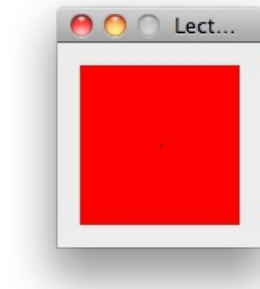
Tässä on yksinkertainen ohjelma, joka näyttää ellipsin tietokoneesi kellon sekunneilla. Käytämme tätä ohjelmaa pohjana luodaksemme kelloäänän, joka soi joka minuutti. Koska minuutin odottelu on pitkä aika, esitämme sekuntien kulumisen ellipsillä, joka kasvaa kuvan keskellä.



```
void draw() {  
    background(255);  
    ellipse(50,50,second(),second());  
}
```

Tehdäksemme kellomme käytämme yksinkertaisesti järjestelmän kellon sekunteja asettaaksemme ellipsin parametrit (korkeus, leveys). Jos sekuntiarvo on 42, meillä on 42 x 42 kokoinen ellipsi. Jos sekunnit ovat 2, meidän ellipsimme on 2 x 2 pikseliä.

Voisimme lisätä uuden ehdon, jolla osoitamme minuuttien muutoksen aina kun sekunnit ovat nollassa. Tällöin ruutu on punainen.



```
void draw() {  
  
    if (second() == 0) {  
        background(255,0,0);  
    } else {  
        background(255);  
    }  
  
    ellipse(50,50,second(),second());  
}
```

Ehto `if (second() == 0) {}` vertailee jälkimmäistä arvoon 0. Jos molemmat vastaavat toisiaan, tausta on punainen, muuten tausta on valkoinen.

KELLON ÄÄNI

Meillä on nyt ohjelmassa ääni, joka toistetaan aina kellon sekuntien ollessa kohdassa 0.

Aloittaaksemme piirrämme kellon `draw()` -metodissa. Kaikki ääneen tarvittavat metodit ovat tässä osassa.

```
import ddf.minim.*;

Minim minim;
AudioSnippet ping;

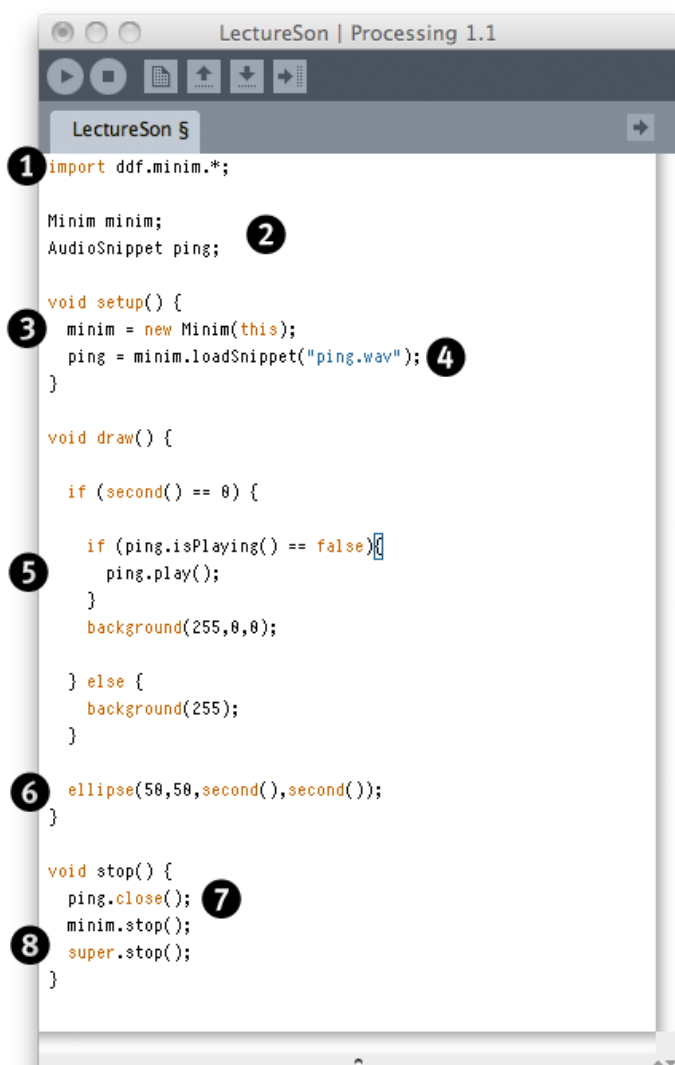
void setup() {
  minim = new Minim(this);
  ping = minim.loadSnippet("ping.wav");
}

void draw() {
  if (second() == 0) {
    if (ping.isPlaying() == false){
      ping.play();
    }

    background(255,0,0);
  } else {
    background(255);
  }

  ellipse(50,50,second(),second());
}

void stop() {
  ping.close();
  minim.stop();
  super.stop();
}
```



Tämä ohjelma vaatii monta askelta toimiakseen:

1. Tuodaan Minim ohjelmaan.
2. Luodaan kaksi muuttujaa, joista toinen sisältää Minim-koneen ja toinen äänidataa.
3. Käynnistetään kone Minim in allettua.
4. Kopioidaan data muuttujaan äänitiedostosta.
5. Ehdollisesti soimitaan ääni ja näytetään punainen ruutu, kun sekunnit ovat nollassa (tarkastetaan ettei ääni ole jo soimassa).
6. Piirretään ellipsi kellon sekuntien pohjalta.
7. Lopetetaan ääni ohjelman lopussa.
8. Lopetetaan Minim ohjelman lopussa.

Olemme jo selittäneet Minim-koneen käynnistämisen ja lopettamisen. Tässä tulee kiinnittää huomio ohjelman osiin 2, 4 ja 5, jotka kuvaavat tuonnin ja toiston.

Ensin luomme muuttujan tyyppiä AudioSnippet. AudioSnippet -tyyppi on muuttuja, joka on määritelty Minim koodissa. Muistutamme, että jokaisen muuttujan tapauksessa kaava on tyyppiä (tyyppi)(muuttujan nimi) = (muuttujan arvot). Jos on esimerkiksi mahdollista tuoda Processingiin söpö pieni lintu, riittää kirjoittaa PikkuLintu loadLintu = ("lintu.bird"). Kirjoitamme ensin olion tyypin, sen nimen, ja lopulta sen arvon. Tässä arvo annetaan funktiolle loadSnippet(), joka hakee wave-äänitiedoston arvot ja tuo ne muuttujaan nimeltä "ping".

Lisätietoa muuttujista ja erilaisista muuttujatyypeistä saat aiheelle omistetusta luvusta.

Kun muuttujaan on ladattu ääni, voit vain toistaa sen. AudioSnippetin soittamiseksi kutsutaan komentoa play(). Tässä voidaan myös kysyä onko ääni jo toistettavana, jolloin komento isplaying kertoo meille toistuuko ääni jo (true) vai eikö toistu (false). Jos tämä komento unohdetaan, komento laukaistaan monta kertaa metodin second() ollessa nollassa.

AUDIOSNIPPET, AUDIOSAMPLE, AUDIOPLAYER

Minimin dokumentaation mukaan Processingissa on monta tapaa soittaa ääniä. Periaatteessa valintasi perustuu äänen rooliin ohjelmassasi. Minim dokumentaation mukaan AudioSnippetä käytetään lyhyen äänen soittoon, AudioSamplea nopeaan toistuvan äänen lukemiseen, esimerkiksi rumpukoneen äänen, ja AudioPlayer on käytössä pidempien äänien (usein MP3) tapauksessa. Nämä pidemmät äänet toistetaan suoraan kovalevyiltä, jotta vältetään ohjelman sisäisen muistin käyttöä. Tässä on esimerkki, joka käyttää kaikkia kolmea toistotapaa.

```
import ddf.minim.*;

Minim minim;
AudioSample ping;
AudioSnippet pop;
AudioPlayer song;

void setup() {
  minim = new Minim(this);
  ping = minim.loadSample("ping.aiff");
  pop = minim.loadSnippet("pop.aiff");
  song = minim.loadFile("song.mp3");
}

void draw() {
}

void keyPressed() {
  if (key == 'a') ping.trigger();
  if (key == 'b') {
    pop.rewind();
  }
}
```

```

        pop.play();
    }
    if (key == 'c') song.play();
}

void stop() {
    ping.close();
    pop.close();
    song.close();
    minim.stop();
    super.stop();
}

```

Huomaa, että `AudioPlayer` ja `AudioSample` käyttävät molemmat `play()`-metodia, koska niiden lukema on aina ainutlaatuinen (yksi ääni kerrallaan). `AudioSample` vaatii monen samanaikaisen äänen lukemista ja käyttää komentoa `trigger` äänen laukaisemiseen. Englanniksi sana "trigger" tarkoittaa liipaisinta.

DOKUMENTAATIO

Minimistä löytyy lisää tietoa sivulta <http://code.compartmental.net/tools/Min/> ja vaikeampaa teknistä dokumentaatiota sivulta <http://code.compartmental.net/minim/javadoc/ddf/minim/package-tree.html>.

18. DRAW-METODI

Toistaiseksi olemme luoneet lineaarisia ohjelmia: ohjelma alkaa, se piirtää piirroksemme ja loppuu komentojen loputtua.

Processing ei kuitenkaan ole pelkkä piirrosympäristö, se on myös interaktiivinen ympäristö. Interaktiivisuus vaatii aikaa, sillä ympäristö muuttuu ajan kuluessa ja eri seikkojen vaikuttaessa siihen. Tähän tarvitaan loputonta silmukkaa, jota ohjelma kutsuu jatkuvasti päivittääkseen grafiikkaa. Processingissa tämä loputon silmukka on komento `draw()`. Siihen liittyy usein metodi `setup()`, joka valmistelee kuvan, esimerkiksi asettamalla sen koon heti alussa.

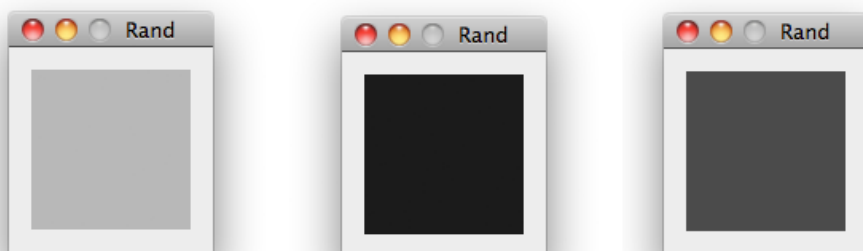
Näiden kahden metodin puute tekee Processingista epäaktiivisen. Jos koodissa ei ole metodia `draw()`, ohjelma loppuu koodin loppuessa.

DRAW()

Aloita uusi tyhjä Processing-ohjelma, kirjoita seuraavat rivit skriptiikkunaan, ja paina **play** -nappia:

```
void draw() {  
  background( random(255) );  
}
```

OK, olet mahdollistanut animaation Processingissa. Sinun pitäisi nähdä ikkuna, joka välähtää 30 kertaa sekunnissa satunnaisessa harmaan sävyssä jossain mustan ja valkoisen välillä. Tämä on metodi `random(255)`, joka antaa satunnaisia arvoja väliltä 0-255. Tämä arvo haetaan sen jälkeen taustalle komennolla `background()`. Koska kaikki tapahtuu toistuvasti, päivittyy kuva kuin elokuvassa.



Oletusarvoisesti komennot, jotka ovat kahden draw-komennon aaltosulun välissä, suoritetaan 30 kertaa sekunnissa. Processing avaa tämän metodin 30 kertaa sekunnissa ja katsoo mitä sisällä on. Voimme laittaa tämän metodin sisään niin monta komentoa kuin tahdomme. Nämä komennot Processing toistaa loputtomasti.

Varo kirjoitusvirheitä tai Processing lopettaa toiston - mikä on animaatiosi loppu. Jos olet kuitenkin kirjoittanut komennon draw() ilman virheitä, se kutsuu silmukkaa 30 kertaa sekunnissa, kunnes käyttäjä lopettaa ohjelman, sähkötkä katkeavat, tai maailma loppuu. Niinpä voimme luoda animaatioita tällä koodin luomisen menetelmällä.

SUORITUSTAAJUUS

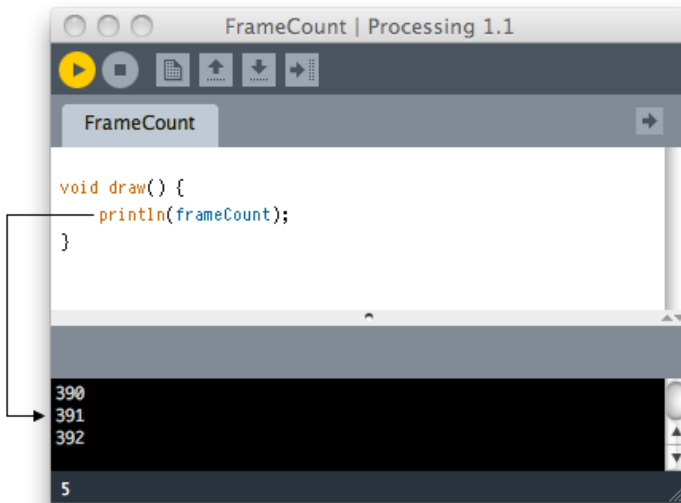
On mahdollista määritellä eri taajuus käyttämällä metodia frameRate().

Jos muutat edellistä esimerkkiä seuraavasti, huomaat nopeuden hidastuneen kolmasosaan aiemmasta.

```
void draw() {  
    frameRate(10);  
    background( random(255) );  
}
```

KUINKA MONTA KERTAA DRAW () KUTSUTTIIN

Processing voi myös laskea kuinka monta kertaa metodi draw () on kutsuttu ohjelman käynnistämisestä alkaen muuttujalla frameCount.



SETUP ()

Usein joudumme myös kirjoittamaan ohjeet vain ohjelman alkuun. Voimme esimerkiksi määritellä kuvan koon vain kerran, ja tätä määritelmää ei sijoiteta komennon draw() alkuun, koska draw() suoritetaan jatkuvasti. Processing antaa meille oletusarvoisen kuvan, jonka koko on 100x100 pikseliä, ja voimme muuttaa sen asetuksia.

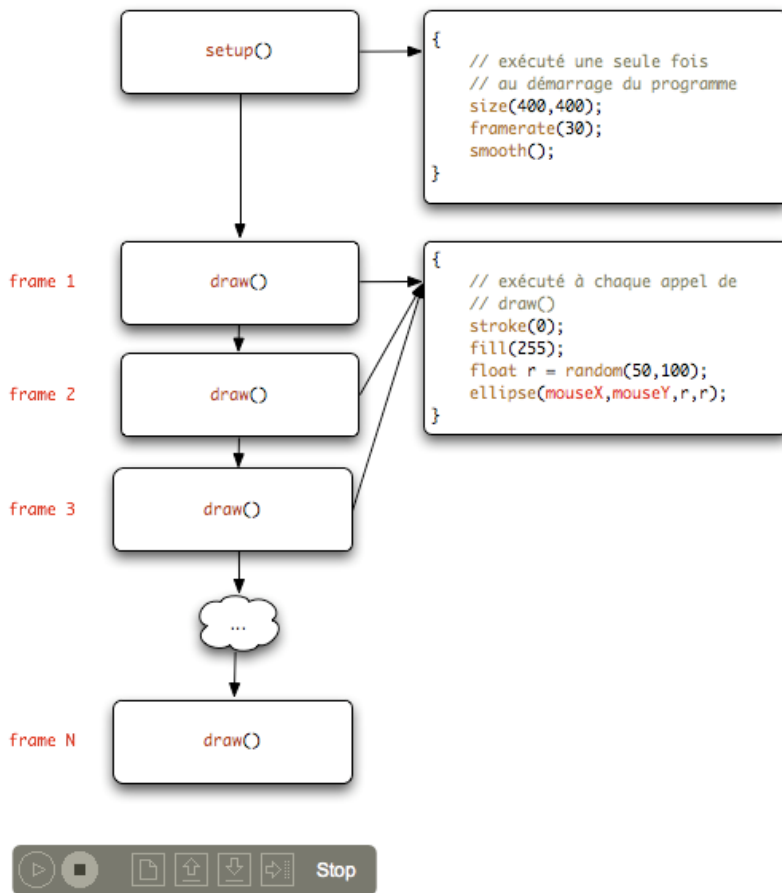
Näistä syistä on luotu täydentävä komento: metodi setup().

```
void setup() {  
}
```

Tämän metodin loivien yskittäisten merkkien hienoudet eivät ole tärkeitä tässä vaiheessa. Sinun pitää vain tietää, että alkuun pitää kirjoittaa void setup(), jota seuraa aaltosulku, käyttämäsi komennot, ja lopulta toinen aaltosulku.

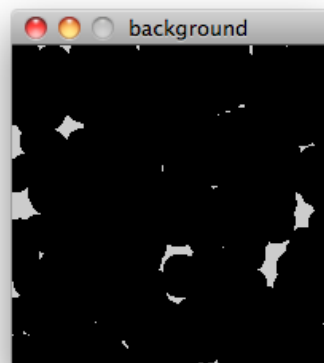
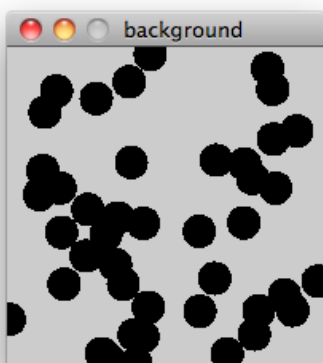
```
void setup() {  
  size(500, 500);  
}  
  
void draw() {  
  background(random(255));  
}
```

Tämän jälkeen Processing katsoo metodin sisälle joka kerralla, kun ohjelma suoritetaan. Kun nämä komennot on suoritettu, Processing kutsuu metodia draw(), joka toistetaan uudestaan ja uudestaan.



BACKGROUND()

Tässä on ohjelma, joka täyttää ruudun ellipseillä.



```
void setup() {
```

```

    size(200, 200);
    fill(0);
}

void draw() {
    ellipse(random(200), random(200), 20, 20);
}

```

Kuten ruutukaappauksesta näkyy, tämä elokuva täyttää kuvamme kokonaan mustalla.

Lisää vain ohjelmaan puhdas taustakuva, käyttäen metodia `background()`. Se ottaa yhdestä kolmeen väriarvoa, kuten metodit `fill()` tai `stroke()`.

Tässä on ohjelman muunnos, jossa näemme nyt yhden animoidun muodon:

```

void setup() {
    size(200, 200);
    fill(0);
}

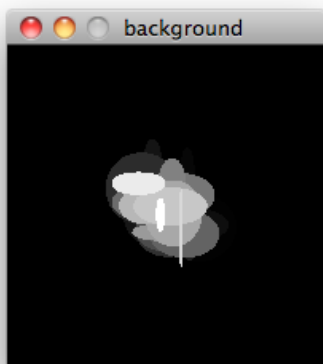
void draw() {
    background(255);    ellipse( 100, 100, random(100), random(100));
}

```

Todellisuudessa tyhjennämme ruudun 30 kertaa sekunnissa, jonka jälkeen se piirtää ellipsin taustalle.

LISÄÄ SULATUS

Tätä tempua käytetään usein Processingissa. Kuvan tausta pyyhitään, mutta jäljelle jätetään osittain läpinäkyvä neliö, johon piirretään komennolla.



```

void setup() {
    size(200,200);
    background(0);
    noStroke();
}

void draw() {
    fill(0, 0, 0, 20);
    rect(0, 0, 200, 200);
    fill(255);
    ellipse(100 + random(-20,20), 100 + random(-20,20), random(50),

```

```
random(50));  
}
```

19. OLIOIDEN ANIMOINTI

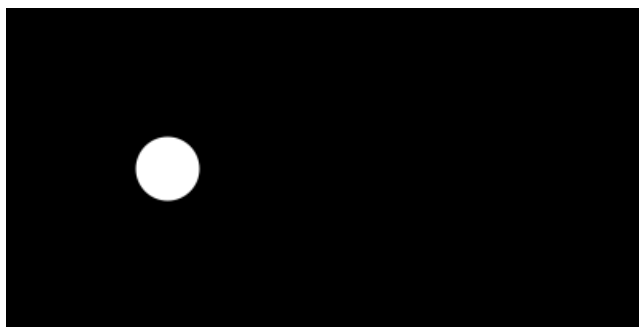
Olioiden animointi on näyttämö, jolle voimme ajatella laajempia käyttötarkoituksia. Olion animointi merkitsee toimintojen lisäämistä malliin: liike, pomppoilu, törmäysten testaus jne. Tämä luku näyttää esimerkkejä luvussa Oliot luodun pallon käsittelystä ja sen pomppoiluttamisesta ruudun neljän reunan välillä.

Lopputuloks on kuin seuraavassa kuvassa. Laitamme viivästyseffektin näyttääksemme pallon liikeradan.



KOODIPERUSTA

Alkupisteeksi otamme pallon koodin luvusta Kohteet. Se antaa meidän nähdä pallon ruudulla. Tämän proseduurin kuluessa lisäämme koodinpätkiä perusesimerkkiin. Uudet osat näytetään paksulla ja niissä on kommentit //LISÄÄ tai // LISÄYS ALKAA ja // LISÄYS LOPPUU. Voit myös kopioida koko koodiblokin ja korvata kyseessä olevan luokan tai metodin.



```
//Yhden pallon instanssin julistaminen
Pallo myPallo = new Pallo(100, 100, color(255));

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko
}

void draw() {
  background(0); //Mustan taustan luominen
  noStroke(); //Ei ääriiviivaa

  myPallo.display(); //Näytetään pallo
}
```

```

class Pallo {
    //Pallon ominaisuuksien julistus
    float x;
    float y;
    color col;

    //Pallon luominen
    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;
    }

    //Pallon suunnittelu
    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }
}

```

LIIKE

Pallon täytyy liikkuu x- ja y- akseleita pitkin. Luomme kaksi muuttujaa malliin ja olion toimintaa edustaa sen nopeus akseleita x ja y pitkin. Tämän jälkeen lisäämme olioon uuden metodin move(), jota kutsutaan ohjelman metodista draw(). Tämä metodi tulee kuitenkin, aina oliota näytettäessä, muuttamaan pallon sijaintia suhteessa sen nopeuteen. Se alustaa myös konstruktorin nopeuden kuvaavat muuttujat. Aluksi annamme niille pysyvän arvon.

```

class Pallo {
    //Pallon ominaisuuksien julistus
    float x;
    float y;
    float speedX; //LISÄÄ
    float speedY; //LISÄÄ
    color col;

    //Pallon rakentaminen
    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;

        speedX = 2; //LISÄÄ
        speedY = 2; //LISÄÄ
    }

    //Pallon suunnittelu
    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }

    //LISÄYS ALKAA
    void move() {
        x = x + speedX;
        y = y + speedY;
    }
    //LISÄYS LOPPUU
}

```

Se kutsuu metodia move() metodista draw(). Voimme lisätä kutsun komenttoon background pyyhkiäksemme ruudun jokaisen uuden kuvan jälkeen.

```

void draw() {
    background(0); //Mustan taustan luominen
    noStroke(); //Ei ääri viivoja

    //Pallon liikuttaminen ja näyttäminen

```



```

    myPallo.move(); //LISÄÄ
    myPallo.display();
}

```

TÖRMÄYKSET

Tällä hetkellä pallo jatkaa liikettään törmätessään ruudun seinään. Pallo pitäisi saada poukkoilemaan seinistä, joten lisäämme metodin `testaaTormays`, joka kääntää pallon liikesuunnan sen osuessa seinään.

```

class Pall {
    //Pallon ominaisuuksien julistaminen
    float x;
    float y;
    float speedX;
    float speedY;
    color col;

    //Pallon luominen
    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;

        speedX = 2;
        speedY = 2;
    }

    //Pallon suunnittelu
    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }

    void move() {
        x = x + speedX;
        y = y + speedY;
    }

    //LISÄYS ALKAA
    void testaaTormays() {
        //Pallo pomppaa törmätessään seinään
        if (x > width-20 || x < 20) {
            speedX = speedX * -1;
        }
        if (y > height-20 || y < 20) {
            speedY = speedY * -1;
        }
    }
    //LISÄYS LOPPUU
}

```

Nyt lisätään kutsu metodiin `testaaTormays` metodista `draw()`.

```

void draw() {
    //Korvataan komennolla background () näille kahdelle riville
    fill(0, 0, 0, 1); // Läpinäkyvä väri
    rect(0, 0, width, height);
    noStroke();

    //Pallon liikuttaminen ja näyttö
    myPallo.move();
    myPallo.testaaTormays(); //LISÄÄ
    myPallo.display();
}

```

LOPULLINEN KOODI

Tässä on lopullinen koodi kaikkien näiden muutosten jälkeen.

```

//Yhden olion pallo instanssin luominen
Pallo myPallo = new Pallo(100, 100, color(255));

```

```

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko
}
void draw() {
  fill(0, 0, 0, 1);
  rect(0, 0, width, height);

  noStroke();

  //Pallon näyttö ja liikuttaminen
  myPallo.move();
  myPallo.testaaTormays();
  myPallo.display();
}

class Pallo {
  //Pallon luokan ominaisuuksien julistaminen
  float x;
  float y;
  float speedX;
  float speedY;
  color col;

  //Pallon luominen
  Pallo (float newX, float newY, color newColor) {
    x      = newX;
    y      = newY;
    col    = newColor;

    speedX = 2;
    speedY = 2;
  }

  //Pallon suunnittelu
  void display() {
    fill(col);
    ellipse(x, y, 40, 40);
  }

  void move() {
    x = x + speedX;
    y = y + speedY;
  }

  void testaaTormays() {
    //Pallo pomppaa törmätessään muuriin
    if (x > width-20 || x < 20) {
      speedX = speedX * -1;
    }
    if (y > height-20 || y < 20) {
      speedY = speedY * -1;
    }
  }
}

```

20. USEAMMAN OLION ANIMOINTI

Olioista koostuvan ohjelman oliomäärä voi lisääntyä nopeasti. Olioihin johdattelevassa luvussa näimme, että voimme laittaa ruudulle kaksi palloa ja laittaa siirtovuoron metodiin `draw()`. Tämä voi olla hankala toimintatapa, kun kohteita tarvitaan yli kaksi.

Tässä kappaleessa jatkamme pomppoilevan pallon esimerkillä. Lisäämme enemmän palloja, käytämme listoja, jonka jälkeen lisäämme metodin käsittelemään pallojen törmäyksiä toisiinsa.



KOODIPOHJA

Aloituspisteenä on koodi luvusta "Olion animointi". Luvun koodi antaa meidän laittaa ruudulle pomppivan pallon. Proseduurin kuluessa lisäämme koodinpätkiä vanhaan esimerkkiin. Uudet koodinpätkät näytetään lihavoituna ja merkittyinä kommentteina // LISÄÄ tai // LISÄYKSEN ALKU ja // LISÄYKSEN LOPPU. Voit myös kopioida koko koodiblokin ja korvata kyseessä olevan luokan tai metodin.



```
//Julista ja luo olion instanssi Pallo
Pallo myPallo = new Pallo(100, 100, color(255));

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko
}

void draw() {
  fill(0, 0, 0, 1); // Läpinäkyvä väri
  rect(0, 0, width, height);

  noStroke();
```

```

    //Pallon liikuttaminen ja näyttäminen
    myPallo.move();
    myPallo.testCollision();
    myPallo.display();
}

class Pallo {
    //Pallon perusparametrien julistus
    float x;
    float y;
    float speedX;
    float speedY;
    color col;

    //Pallon rakennus
    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;

        speedX = 2;
        speedY = 2;
    }

    //Piirrä pallo
    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }

    void move() {
        x = x + speedX;
        y = y + speedY;
    }

    void testaaTormays() {
        //Jos pallo osuu seinään se pompahtaa
        if (x > width-20 || x < 20) {
            speedX = speedX * -1;
        }
        if (y > height-20 || y < 20) {
            speedY = speedY * -1;
        }
    }
}

```

LISTAA PALLOT

Voimme nyt käyttää listan konseptia pomppivaan palloon. Tämän avulla meillä voi olla monta palloa ruudulla, ilman että joudumme kopioimaan koodia!

Ensin julistamme listan palloja yhden pallon sijasta. Käytämme muuttujaa palloLaske laskemaan ohjelmassa olevien pallojen määrän.

Korvaamme seuraavan koodin

```
Pallo myPallo = new Pallo(100, 100, color(255));
```

tällä koodilla:

```

//Julista pallojen määrän sisältävä muuttuja
int palloLaske = 3;

//Julistetaan pallon instanssien määrän sisältävä muuttuja
Pallo[] pallot = new Pallo[palloLaske];

```

Kuten kokonaislukuja käsittelevässä esimerkissä, meidän täytyy vain julistaa mallipallon kopiot. Ne täytyy nyt luoda komennossa setup (). Luomme kolme palloa ruudun keskelle. Kaikki kolme ovat valkoisia.

```

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko

  //Lisäys alkaa
  //Tämä silmukka luo kolme palloa
  //Valkoista ruudun keskellä
  for (int i = 0; i < palloLaske; i++) {
    pallot[i] = new Pallo(width/2, height/2, color(255));
  }
  //Lisäys loppuu
}

```

Metodissa draw () luomme myös silmukan, joka käy läpi kaikki elementit siirtääkseen niitä, testaa niiden törmäykset ja näyttää ne. Korvaamme koodin

```

//Liikuta palloa ja näytä
myPallo.move();
myPallo.testaaTormays();
myPallo.display();

```

tällä koodilla:

```

//Tämä silmukka liikuttaa ja näyttää kolmea palloa
for (int i = 0; i < palloLaske; i++) {
  pallot[i].move();
  pallot[i].testaaTormays();
  pallot[i].display();
}

```

Lopullisena operaationa on muokata mallia pallost, jotta jokaisella pallolla on oma nopeutensa ja suuntansa. Tehdäksemme tämän käytämme funktiota random(). Korvaamme seuraavan koodin

```

//Pallon luominen
Pallo (float newX, float newY, color newColor) {
  x      = newX;
  y      = newY;
  col    = newColor;

  speedX = 2;
  speedY = 2;
}

```

tällä koodilla:

```

//Pallon luominen
Pallo (float newX, float newY, color newColor) {
  x      = newX;
  y      = newY;
  col    = newColor;
  speedX = 2 + random(-1,1);
  speedY = 2 + random(-1,1);
}

```

LOPULLINEN OHJELMA

Tässä on lopullinen ohjelma, jossa on eri arvot muuttujalle ballCount.

```

//Pallojen määrän sisältävän muuttujan julistaminen
int laskePallo = 3;

//Pallon instanssit sisältävän listan julistaminen
Pallo[] pallot = new Pallo[palloLaske];

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko

```

```

//Tämä silmukka luo kolme valkoista palloa
//Ikkunan keskelle
for (int i = 0; i < palloLaske; i++) {
    pallot[i] = new Pallo(width/2, height/2, color(255));
}

void draw() {
    fill(0, 0, 0, 1); // Väri on läpinäkyvä
    rect(0, 0, width, height);

    noStroke();

    //Tämä silmukka luo ja näyttää kolme palloa
    for (int i = 0; i < palloLaske; i++) {
        pallot[i].move();
        pallot[i].testaaTormays();
        pallot[i].display();
    }
}

class Pallo {
    //Pallon parametrien julistaminen
    float x;
    float y;
    float speedX;
    float speedY;
    color col;

    //Pallon luominen
    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;

        speedX = 2 + random(-1,1);
        speedY = 2 + random(-1,1);
    }

    //Pallon suunnittelu
    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }

    //Pallon liike
    void move() {
        x = x + speedX;
        y = y + speedY;
    }

    void testaaTormays() {
        //Pallo pompsahtaa muurista
        if (x > width-20 || x < 20) {
            speedX = speedX * -1;
        }
        if (y > height-20 || y < 20) {
            speedY = speedY * -1;
        }
    }
}

```

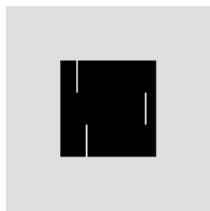
21. AIKAJANA

Animaatiota luotaessa tarvitaan liikettä. Liike merkitsee muutosta ajan kuluessa. Sijainti, väri tai joku muu asia voi muuttua. Tieto muutoksesta voidaan tallentaa muuttujiin.

Animaatioita luotaessa sinun tulee tietää, missä järjestyksessä asiat tapahtuvat aikajanalla. Voimme käyttää joko oikeaa aikaa tai laskea esimerkiksi yhdestä kymmeneen.

PALJONKO KELLO ON?

Voimme luoda kellon kutsumalla Processingissa metodeita `hour()`, `minutes()` ja `second()`. Käytämme ulostuloa muuttaaksemme kolmen ohuen neliön sijaintia horisontaalisella akselilla.



Tässä on koodi hyvin yksinkertaisen kellon luomiseen:

```
void setup() {
  size(60, 60);
  noStroke();
}

void draw() {
  background(0);

  // Tunnit ovat 0-23
  // Muunnamme skaalalta 0-60

  rect((hour() / 24.0) * 60, 0, 1, 20);
  rect(minute(), 20, 1, 20);
  rect(second(), 40, 1, 20);
}
```

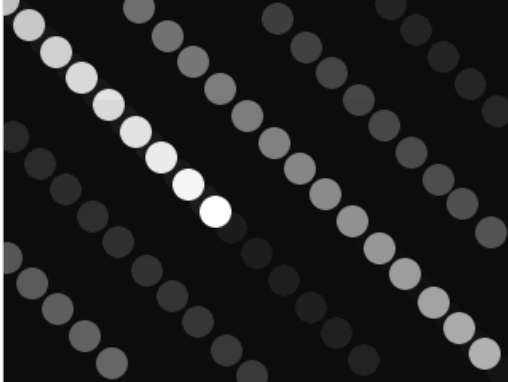
Tässä esimerkissä kuvat piirretään toistensa päälle. Metodin `background()` kutsuminen piirtää kuvat toistensa päälle. Komento `background()` täyttää piirroksen määritellyllä värillä, jotta kuva poistetaan.

MITTAA AJAN KULUMINEN

Saat ohjelman suorittamisen aloitushetkestä kuluneen ajan komennolla `millis()`. Se palauttaa ajan millisekunteina. Voimme luoda animaatioita käyttäen tätä arvoa. Sekunnissa on tuhat millisekuntia. Tämä antaa meille melko korkean tarkkuuden muotoja animoitaessa.

Jotta animaatiot toimivat, voit käyttää modulo-operaattoria (%), josta saat jakojäännöksen. Esimerkiksi komento `System.out.println(109% 10);` näyttää tuloksen 9, sillä 109 jaettuna kymmenellä antaa 10 ja 9 jakojäännökseksi jää 9. Vastaavasti kahden numeron x ja y ottaminen, ja jakaminen numeroilla x ja y, antaa jakojäännöksen, joka on vähemmän kuin y. Modulo antaa meidän laskea ja olla koskaan ylittämättä tiettyä numeroa.

Seuraavassa esimerkissä piirrämme ympyrän, joka kulkee hyvin nopeasti vinossa suunnassa. Tämä ympyrä jättää jälkeensä vanan, joka katoaa hyvin hitaasti.



Tässä on koodi, jota käytetään sen piirtämiseen.

```
void setup() {
  size(320, 240);
  noStroke();
  frameRate(60);
  smooth();
}

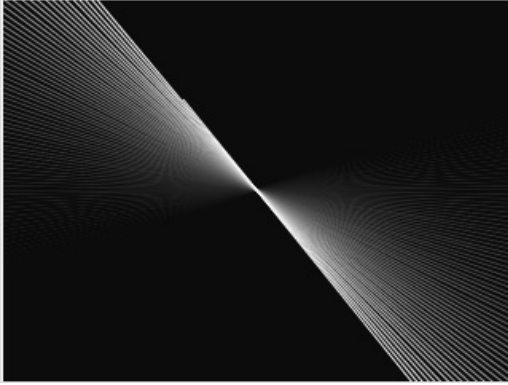
void draw() {
  fill(0, 0, 0, 10);
  rect(0, 0, width, height);
  fill(255);
  ellipse(millis() % width, millis() % height, 20, 20);
}
```

Sen sijaan että kutsuttaisiin komentoa `background()`, piirrämme neliön osittain läpinäkyvänä. Tämä tekee liikkeen sumennuksen. Kutsumme komentoa `smooth()`, koska rakastamme tasaisia kurveja.

On paras tehdä toiminnot ajan perusteella eikä laskea piirrettyjä kuvia. Tämä johtuu siitä, että renderöintinopeus riippuu koneesta ja sillä pyörivistä muista ohjelmista. Animaatioista tulee nykiviä, jos käytämme mittarina laskuria eikä aikaa. Katsomme kuitenkin animaatioiden tekemistä laskurilla, koska se on hyvin helppoa.

ANIMOI LASKURILLA

Näemme nyt, kuinka animaatio tehdään laskurilla. Piirrämme yhden viivan, joka liikkuu jatkuvasti.



Koodi on melko yksinkertaista. Avain on kuvien laskenta. Muutamme viivan pyörimistä laskemalla kuvan renderöintikertoja.

```
int laskin;

void setup() {
  size(320, 240);
  frameRate(60);
  fill(0, 0, 0, 10);
  stroke(255);
  smooth();
  laskin = 0;
}

void draw() {
  frame = frame + 1;

  rect(0, 0, width, height);

  translate(width / 2, height / 2);
  rotate(radians(frame));
  line(-height, -height, height, height);
}
```

Julistetaan laskuri.

```
int laskin;
```

Laskuri initialisoidaan komennossa setup ().

```
void setup() {
  size(320, 240);
  frameRate(60);
  fill(0, 0, 0, 10);
  stroke(255);
  smooth();
  laskin = 0; }
```

Jokaisessa draw () -metodikutsussa yksi lisätään muuttujaan frame.

```
frame = frame + 1;
```

Tämän jälkeen voit siirtää piirroskohdan piirrosavaruuden keskustaän ja pyörimisliike riippuu ruudusta.

```
translate(width / 2, height / 2);
rotate(radians(frame))
```

OHJELMOINTI

22. KOMMENTIT

23. OLIOT

24. METODIT

25. LISTAT

26. TOISTAMINEN

27. EHDOT

28. MUUTTUJAT

29. MUUTOKSET

30. VIIVATYYLIT

22. KOMMENTIT

Koodin dokumentointi on tärkeää, jotta muut voivat korjata ja laajentaa koodia. Se mahdollistaa myös tietyn muuttujan tai metodin käytön tutkimisen. Kun ohjelmaa kirjoitetaan, ovat kommentit sen dokumentointi. Joskus kommentit näyttävät turhilta, mutta jos tahdot käyttää koodia uudestaan, kommenttien käyttö tekee siitä paljon helpompaa.

Processing tarjoaa kaksi tapaa kommentoida koodia: kommentit riveillä ja monen rivin kommenttiblokkit.

YHDEN RIVIN KOMMENTIT

Vain yhden rivin kommenttien kirjoittamista varten voit laittaa merkit // kommentin alkuun. Kaikki näistä merkeistä oikealle oleva teksti lasketaan kommentiksi: sitä ei oteta huomioon ohjelmaa suoritettaessa. Esimerkkejä kommenteista riveillä:

```
void setup() {  
  // Määritellään ikkunan koko  
  size(400, 300);  
  smooth(); // Aktivoidaan ääri viivojen siistiminen  
}
```

KOMMENTTIBLOKIT

Jos selitys vaatii pidemmän kommentin, se voidaan kirjoittaa useammalle riville laittamalla ne merkkien "/*" kommentti "*/" väliin. Esimerkiksi:

```
/*  
Metodi alustaa ohjelman setup(),  
voimme määritellä ikkunan koon,  
asettaa ohjelman alkutilan jne.  
*/  
void setup() {  
  
}
```

Voit kommentoida rivin menemällä edit-valikkoon ja napsauttamalla "comment/uncomment" -nappia. Voimme tehdä saman poistaaksemme kommenttiblokin.

KOMMENTTIEN KÄYTTÖ

Ohjelman kommentointi ei merkitse jokaisen rivin merkityksen selittämistä! Yleensä kommenttiblokki selittää metodien käyttötarkoituksen ja -tavan. Tämä vaatii jättämään tavalliset Processingin komennot selittämättä: esimerkiksi setup(), draw() jne. voidaan jättää selittämättä. Yleisiä komentoja ei tarvitse kommentoida. Esimerkiksi size(), fill() ja ellipse() ovat ilmeisiä.

Kun ohjelman rivit tekevät yhteisen kommentin, kannattaa laittaa erillisten kommenttien sijasta yksi kommentti. Esimerkiksi:

```
x = x + 10; //Lisätään 10 koordinaattiin x  
y = y + 10; //Lisätään 10 koordinaattiin y
```

Voitaisiin kirjoittaa:

```
//Koordinaattien x ja y muokkaaminen  
x = x + 10;  
y = y + 10;
```

23. OLIOT

Olio-ohjelmoinnissa ohjelma rakennetaan elementeistä, jotka ovat olemassa ohjelmassa (luodit, seinät, hahmot...). Olio on malli, joka voidaan kopioida, ja jonka jokainen kopio on yksilöllinen, ne ovat sovelluksen rakennuspalikoita. Olio rakentuu ominaisuuksista ja metodeista. Jokainen olion instanssi omaa erilaiset ominaisuudet, mutta sillä on käytössään samat metodit.

Tässä luvussa näytämme objektien käsittelyn ja piirrämme kaksi palloa (yksinkertaista ympyrää) ruudulle. Tässä esimerkissä on pallon malli (olio) ja sen kopiot (instanssit), joista jokaisella on erilaiset ominaisuudet.

Pallon malli rakentuu ominaisuuksista (muuttujista) seuraavasti:

- **x** - sijainti x-akselilla.
- **y** - sijainti y-akselilla.
- **col** - värit.

Malli sisältää seuraavan pallon toiminnon (metodin).

- **display()** - pallo näkyviin.

OLION LUOMINEN

Kuinka voidaan piirtää joukko palloja ruudulle? Selkein selitys on palloa esittävien muuttujien kopiointi. Lisäksi tarvitaan ohjeet pallojen näyttämistä ja hallintaa varten. Koodin koko on suhteessa ruudulla oleviin kohteisiin. Kaksi ongelmaa ilmenee: jos tahtoo muuttaa tai monimutkaistaa kohdetta, koodi täytyy kopioida yhtä monta kertaa kuin kohteita on. Pian koodia on mahdoton pitää järjestyksessä ja se vaatii uutta rakennetta.

Välttääksemme tämän ongelman me muunnamme oliota pallo. Kun olento ohjelmassasi muuttuu liian monimutkaiseksi tai sen täytyy olla olemassa lukemattomina kopioina, sen täytyy olla olio.

LUO OLIO

Olion luominen tapahtuu kahdessa askeleessa:

1. Oliomallin määrittely
2. Yhden olion kopion (instanssin) luominen.

MALLI

Processing käyttää avainsanaa `class` määrittelemään olion. Sitä käytetään metodin määritelmänä: `class (objektinNimi)`. Kaikki toiminnot ja ominaisuudet kirjoitetaan sen sisälle. Yleisesti ottaen kirjoitamme olion määritelmän koodimme lopussa. Varsinkin jos olio on monimutkainen, kannattaa luoda uusi välilehti, joka on erillään muusta koodista.

```
class Pallo {  
}
```

INSTANSSI

Kun malli on määritelty, meidän täytyy luoda yksittäinen kopio tästä mallista. Processing käyttää avainsanoja luodakseen uuden instanssin oliosta: `new objektinNimi()`. Tämä olio pitäisi varastoida muuttujaan, jotta sitä voidaan manipuloida myöhemmin.

Listen

Read phonetically

```
Pallo myPallo = new Pallo();
```

Edellisessä esimerkissä julistamme muuttujan `myPallo`, joka on kopio oliosta `Pallo`. `myPallo` viittaa tähän olioon ja voi toimia myöhemmin ohjelmassa. Jotta tämä olio on käytettävissä koko ohjelmassa, laitamme sen julistuksen ohjelman alkuun.

OMINAISUUDET

Olioilla on ominaisuuksia, jotka määrittelevät ne ja tekevät niistä ainutlaatuisia. Nämä ovat muuttujia, jotka tehdään objektin alussa.

```
class Pallo {  
  //Pallon parametrien julistus  
  float x;  
  float y;  
  color col;  
}
```

Editoidaksesi ominaisuutta objektin luomisen jälkeen täytyy edetä seuraavaksi: `instanssinNimi.ominaisuudenNimi = arvo`. Esimerkiksi:

```
myPallo.x = 100;
```

KONSTRUKTORI

Konstruktori on metodi, jota kutsutaan oliota luotaessa. Sitä vastaa metodi `setup()`. Sillä on edelleenkin sama nimi kuin oliolla. Konstruktori ottaa joukon muuttujia ja laittaa ne olion parametriin:

```
class Pallo {  
  //Pallon parametrien julistus
```

```

float x;
float y;
color col;

Pallo (float newX, float newY, color newColor) {
    x      = newX;
    y      = newY;
    col    = newColor;
}
}

```

Kun luomme olion instanssin, annamme sille suoraan sen omat ominaisuudet parametreinä. Esimerkki alla luo valkoisen pallon, joka on sijainnissa 100, 100.

```
myPallo = new Pallo(100, 100, color(255));
```

Varoitus! Meidän täytyy vain luoda kopio olion mallista. Sitä ei vielä näytetä ruudulla.

TOIMINNOT

Olion toiminnot edustavat eri asioita, joita olio voi tehdä. Nämä metodit julistetaan olion sisällä. Olion instanssin kutsu toimintaan on seuraava: instanssinNimi.metodinNimi();

Pallollamme on vain yksi toiminto: olla näkyvillä. Käytämme komentoja fill() ja ellipse() piirtääksemme sen.

```

class Pallo{
    //Pallon parametrien julistus
    float x;
    float y;
    color col;

    Pallo (float newX, float newY, color newColor) {
        x      = newX;
        y      = newY;
        col    = newColor;
    }

    void display() {
        fill(col);
        ellipse(x, y, 40, 40);
    }
}

```

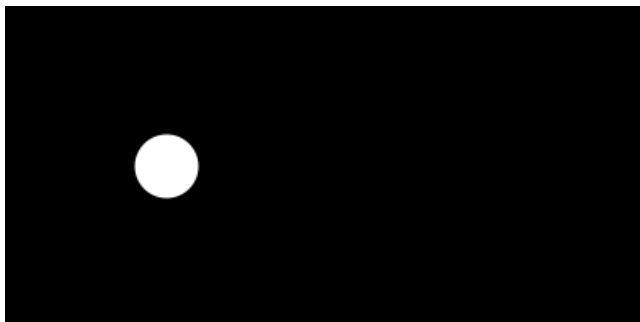
Katsoaksemme palloamme voimme kutsua metodia display() sen kopiosta ohjelman metodissa draw().

```

void draw() {
    myPallo.display();
}

```

LOPULLINEN OHJELMA



```
//Pallon instanssin ja parametrien julistus
Pallo myPallo = new Pallo(100, 100, color(255));

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko
}

void draw() {
  background(0); //Mustan taustan julistus
  noStroke(); //Ei ääri viivoja

  myPallo.display(); //Pallo näkyviin
}

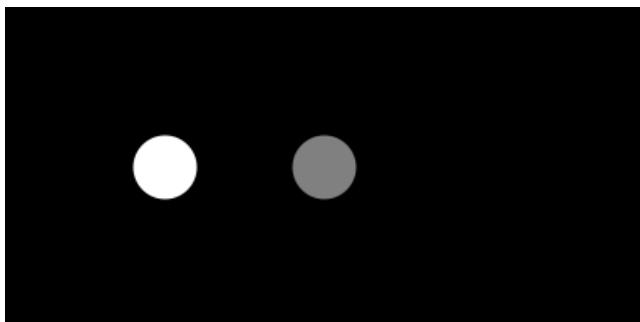
class Pallo {
  //Pallon parametrien julistus
  float x;
  float y;
  color col;

  //Pallon luominen
  Pallo (float newX, float newY, color newColor) {
    x      = newX;
    y      = newY;
    col    = newColor;
  }

  //Pallon näyttäminen
  void display() {
    fill(col);
    ellipse(x, y, 40, 40);
  }
}
```

MONTA PALLOA

Luo toinen pallo julistamalla se esimerkiksi nimellä myPallo2 ja laita se näkyville.



```
//Kahden pallo-instanssin julistus
Pallo myPallo1 = new Pallo(100, 100, color(255));
Pallo myPallo2 = new Pallo(200, 100, color(128));
```



```

void setup() {
  smooth();
  size(400, 200); //Ikkunan koko
}

void draw() {
  background(0); //Mustan taustan julistus
  noStroke(); //Ei ääri viivoja

  myPallo1.display(); //Pallo 1 näkyviin
  myPallo2.display(); //Pallo 2 näkyviin
}

class Pallo {
  //Pallo-luokan parametrien julistus
  float x;
  float y;
  color col;

  //Pallon konstruktori
  Pallo (float newX, float newY, color newColor) {
    x      = newX;
    y      = newY;
    col    = newColor;
  }

  //Pallo näkyviin
  void display() {
    fill(col);
    ellipse(x, y, 40, 40);
  }
}

```

24. METODIT

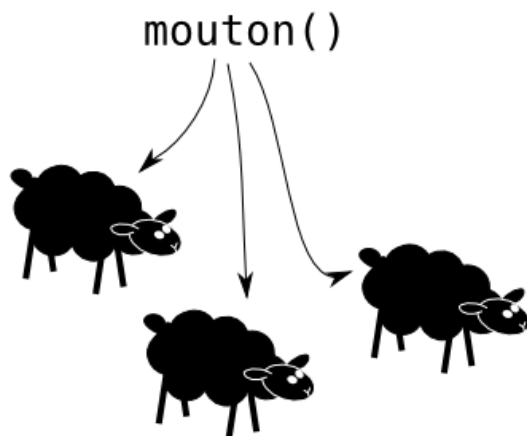
Yksi menetelmä on blokki, joka sisältää ohjeet, joita tahdot käyttää uudestaan. Metodeita on hyvä käyttää, sillä voimme kirjoittaa toiminnon vain kerran ja käyttää sitä uudestaan lukemattomia kertoja. Käyttäessämme metodia koodissamme voimme kutsua sitä uudestaan eri kohdissa ohjelmaa.

Tiedät jo monia metodeja, mutta vain käyttäjänä. Esimerkiksi `rect()`, `ellipse()`, `line()` ja `stroke()` ovat kaikki metodeja. Luomalla omat metodimme voimme lisätä Processingiin ominaisuuksia, joita ohjelman kehittäjät eivät ole vielä tehneet.

Tässä on puhtaan teoreettinen esimerkki lopputuloksesta.

```
tausta(255);  
kaunisMaisema();  
puu(sypressi, 0,300);  
kuu(400,100);  
ruoho(0,300,width,100);  
lammas(50,133);  
lammas(213,98);  
lammas(155,88);
```

Päämääränä on pakata ohjelmien monimutkaisuus avainsanoihin, joita voit kutsua niin usein kuin tahdot. Samalla voit käyttää Processingin omia metodeja. Tuloksena on yksinkertaisempi ja luettavampi koodi. Myös toistolta vältytään. Enemmän järjestystä ja vähemmän leikkaamista ja liimaamista.



AVAINSANAT

Kun luomme omat metodimme, meidän täytyy antaa jokaiselle niistä nimi. Kun metodi on nimetty, sitä voidaan käyttää ohjelmassa. Kutsu sitä vain sen nimellä.

Processing yhdistää joukon menetelmiä, joilla voimme korjata omat versiomme. Tämä pätee metodeille `draw()`, `setup()`, `mousePressed()`... Katso muista luvuista. Voimme myös luoda kustomoituja metodeja antamalla niille valitsemamme nimen. Tässä tapauksessa voit vain varoa käyttämästä nimeä, joka on jo otettu.

LEIKKAUS

Toistaiseksi olemme ohjelmoineet lineaarisesti Processingissa, olemme kirjoittaneet koodia ohjelman alusta alaspäin. Kun tahdomme luoda ohjelmamme metodit itse, joudumme leikkaamaan ohjelmamme kappaleiksi. Tämä antaa meidän osoittaa osat, jotka suoritetaan heti ohjelman alussa, ja ne osat, jotka kutsutaan myöhemmin.

Huono puoli on se, että emme voi kirjoittaa komentoja suoraan Processingiin laittamatta niitä metodin tai luokan sisään. Metodien kanssa toimiminen on kaikki tai ei mitään.

VOID SETUP ()

Processing tarjoaa paikan kirjoittaa koodia, joka suoritetaan ohjelmamme alussa. Tämä metodi on `setup()`:

```
void setup() {  
}
```

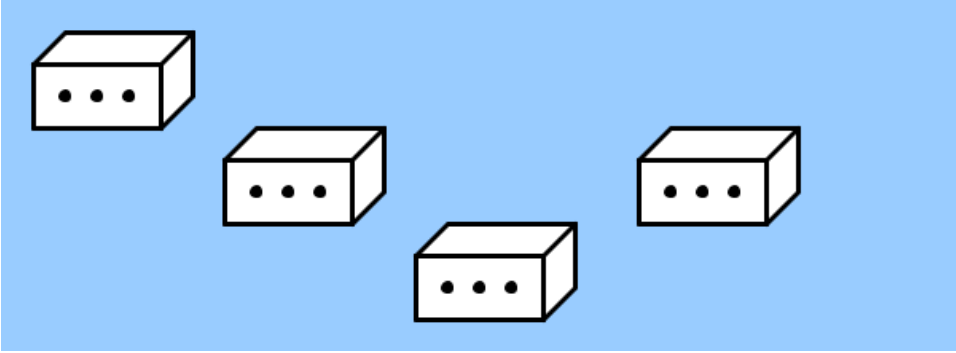
Kaikki ohjelman alussa suoritettava koodi pitää sijoittaa metodin `setup()` aaltosulkujen sisälle. Nyt ei kannata miettiä sanan `void` merkitystä, mutta sen täytyy olla ohjelman alussa. Tämän jälkeen sana `setup`, jota seuraa sulut, ja sitten aaltosulut.

Suurimman osan aikaa käytämme menetelmää `setup()` määrittelemään kuvamme koon. Tämä koko voidaan asettaa vain kerran - mikä yhteensattuma, sillä käynnistyskin tapahtuu vain kerran ohjelmassa.

```
void setup() {  
  size(500,500);  
}
```

LUO KUSTOMOITUJA METODEJA

Alla olevissa esimerkeissä luomme metodin `luoLammas()`, joka sisältää ohjeet lampaan piirtämiseen.



Todellisuudessa lammas on kätkeyty laatikkoon, joten näemme vain laatikon! Se piirtää myös reikiä, jotta lammas voisi hengittää. Kutsumme tätä metodia monta kertaa piirtääksemme enemmän lampaita.

Tässä on koodi piirtämiselle:

```
void setup() {  
  
  size(600, 220);  
  background(153,204,255);  
  smooth();  
  
  // Kutsutaan metodiamme lampaan piirtämiseksi  
  luoLammas();  
  translate(120, 60);  
  luoLammas();  
  translate(120, 60);  
  luoLammas();  
  translate(140, -60);  
  luoLammas();  
}
```

// Metodi lampaan piirtämiseksi.

```
void luoLamma() {  
  
  strokeWeight(3);  
  strokeJoin(ROUND);  
  stroke(0);  
  fill(255);  
  
  rect(20, 40, 80, 40);  
  beginShape();  
  vertex(20, 40);  
  vertex(40, 20);  
  vertex(120, 20);  
  vertex(120, 40);  
  endShape(CLOSE);  
  
  beginShape();  
  vertex(100, 40);  
  vertex(120, 20);  
  vertex(120, 60);  
  vertex(100, 80);  
  endShape(CLOSE);  
  
  fill(0);  
  ellipse(40, 60, 5, 5);  
  ellipse(60, 60, 5, 5);  
  ellipse(80, 60, 5, 5);  
}
```

Ohjelman alku määritellään metodissa setup(). Koska käytämme metodia piirtääksemme lampaita, loput ohjelmasta pitää sijoittaa johonkin tässä metodissa. Processingissa:

```
void setup() {  
}
```

Tämän jälkeen määrittelemme kuvamme koon ja sen taustaväriin.

```
size(600, 220);  
background(153,204,255);
```

Olet ehkä esittänyt kysymyksen komennon `smooth()` roolista ohjelmassamme. Tämä komento on valinnainen, mutta se antaa meille miellyttävimmät viivat tietokoneen ruudulla.

```
smooth();
```

Lopulta piirrämme lampaan käyttäen menetelmää, jonka olemme määritelleet myöhemmin.

```
luoLammas();
```

Kun Processing kohtaa sanan `luoLammas()` se tietää sanojen olevan metodina jossain ohjelmassamme. Jos tämä metodi on olemassa, se kulkee tämän metodin läpi ja tekee kaiken sen sisällä olevan.

Jos tätä metodia ei löydy - ja sitä ei ole Processingissa itsessään - ohjelmasi lopettaa virheviestiin.

Huomaa: voit kirjoittaa avainsanan `luoLammas ()` niin monta kertaa kuin tahdot. Tässä ohjelmassa `luoLammas()` on kirjoitettu 4 kertaa:

```
luoLammas();  
translate(120, 60);  
luoLammas();  
translate(120, 60);  
luoLammas();  
translate(140, -60);  
luoLammas();
```

Huomaa, että asetimme jokaisen metodin `luoLammas ()` väliin komennon `translate (x, y)`. Tämä komento antaa meidän piirtää saman lampaan neljä kertaa eri paikkaan. Luku Muutokset selittää komennon `translate()` toiminnan. Komentoa `translate ()` käytetään siirtämään piirrosten alkupiste.

```
void luoLammas() {  
  
  /* ... */  
}
```

Lopulta pääsemme metodiin `luoLammas()` itseensä. Tässä piirrämme tarvittut viivat ja muodot. Emme keskustele tästä, sillä käytetyt komennot selitetään muotoja käsittelevässä luvussa.

Huomaa `void`-avainsana ennen metodimme nimeä. Tämä merkitsee, että se ei palauta mitään. Emme siis saa dataa ulos metodistamme.

METODIN PALAUTUSARVO

Metodilla voi olla palautusarvo. Toistaiseksi emme ole määritelleet palautusarvoa. Metodilla `setup()` tai `draw()` ei ole palautusarvoa. Sana `void` kertoo näissä tapauksissa, että mitään ei pitäisi palauttaa metodille, joka kutsui ne.

Jos metodilta tahdotaan ulos joku tieto, se tulee palautusarvon muodossa. Jos tahdomme tietää paljonko kello on, käytämme metodia `second()`, `minutes()` tai `hour()` saadaksemme niiden arvon muodossa `integer (int)`. Jos nämä metodit eivät anna meille mitään (`void`) palautusarvona, ne eivät mahdollistaisi hyödyllisiä toimintoja.

Metodien täytyy palauttaa arvo kutsulle, joka on esitetty ennen avainsanaa, joka määrittelee palautetun arvon tyyppin. Metodi jonka tyyppi on `int` palauttaa arvotyyppin `int`, metodi jonka tyyppi on `float` palauttaa tyyppin `float` arvon, ja niin edelleen.

Tässä on esimerkki metodista, joka antaa sekuntien määrän kellon 00:00:00 jälkeen.

```
int sekuntejaEilisesta() {  
    return hour() * 3600 + minute() * 60 + second();  
}  
  
void draw() {  
    println( sekuntejaEilisesta() );  
}
```

Jos et tiedä metodia `draw()`, käytä silti tätä miniohjelmää ja katso konsolia Processing-ikkunan alaosaan. Voit nähdä, että metodi `draw()` kutsuu jatkuvasti komentoa `sekuntejaEilisesta ()` ja käyttää tämän metodin tuloksia näyttääkseen sekunteja.

METODIN PARAMETRIT

Metodi voi ottaa vastaan parametrejä. Yleensä niitä kutsutaan argumenteiksi. Jokaisella parametrillä täytyy olla tyyppi ja nimi, aivan kuin muuttujilla.

Metodin kutsumiseksi voit kirjoittaa sen nimen, jonka perään tulee sulut. Sulkujen sisällä ovat metodin parametrit.

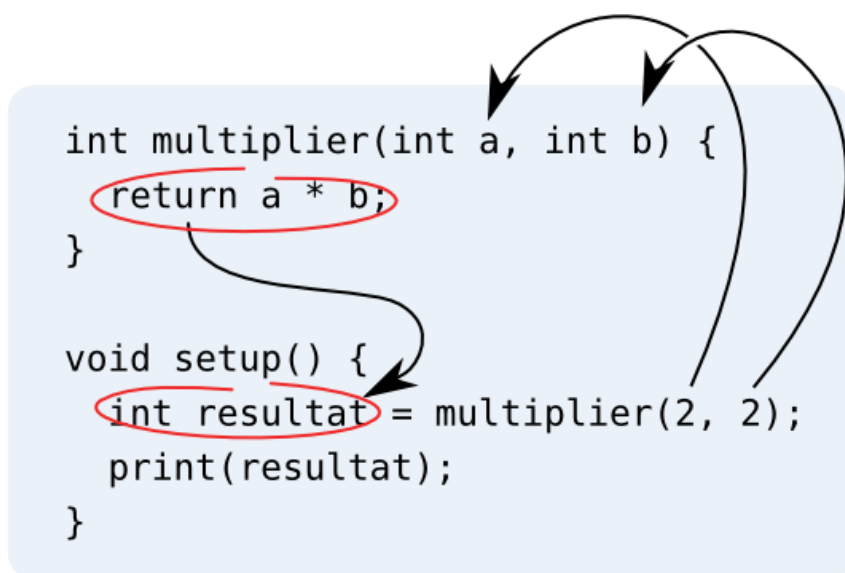
```
kerroin(2, 2);
```

Metodin sisällä Processing pääsee käyttämään niiden arvoja, kuten se tekee muuttujien kanssa.

Esimerkiksi tässä metodi kutsutaan argumenteilla 2 ja 2, arvot ovat $a = 2$ ja $b = 2$. Metodin palautusarvo on 4.

```
int kerroin:(int a, int b) {  
    return a * b;  
}
```

Huomaa, että argumenttien sijainti määrittää, mitkä arvot asetetaan mihinkin pisteeseen.



Esimerkissä loimme metodin, joka palauttaa kahden argumentin kertolaskun tuloksen. Tämä kerrotaan metodia ennen olevassa kommentissa. Tällainen kommentointi on hyvä tapa.

```
/*  
 * Palauta kahden argumentin kertolaskun tulos.  
 */  
int kerroin(int a, int b) {  
    return a * b;  
}  
  
void setup() {  
    int tulos = kerroin(2, 2);  
    print(tulos);  
}
```

4

MUUTTUJIEN LAAJUUS

Katsotaanpa yleistä virhettä, joka voi tapahtua, kun käytät ohjelmassa muuttujia ja metodeja.

Muuttujat - olkoot primitiivisiä tyyppejä tai objekteja - eivät välttämättä ole saatavilla koko ohjelmassasi. Se riippuu niiden julistuspäikasta. Metodin sisällä julistettu muuttuja on käytettävissä seuraavassa:

```
void setup() {  
    int x = 10;  
}  
  
void draw() {  
    /* Tästä ohjelmasta tulee virheviesti, koska  
     * muuttuja x on olemassa vain metodin setup() sisällä  
     */  
    x = x + 1;  
}
```

Jotta muuttujat ovat saatavilla koko ohjelmassasi, sinun pitää julistaa se ohjelman alussa:

```
int x;

void setup() {
  x = 10;
}

void draw() {
  x = x + 1;
}
```


25. LISTAT

Voimme laittaa monenlaisia asioita muuttujaan: numeron, murtoluvun, tekstin lauseen tai vaikka kokonaisen kuvan. Mutta vaikka muuttujat voivat teoriassa sisältää minkä tahansa tyyppin arvon, ne voivat sisältää vain yhden näistä arvoista kerrallaan. Joissain tapauksissa olisi mukavaa ryhmittää joukko asioita yhteen, jolloin ne olisivat ainakin jonkinlaisessa yksittäisessä oliossa. Tästä syystä on keksitty erityinen muuttujatyyppi, listat.

Listoja käytetään tallentamaan ennalta määrätty joukko tietoa tai olioita yhdessä muuttujassa. Sen sijaan että loisit 20 muuttujaa tallentamaan 20 erilaista arvoa, voit luoda yhden säilön näille 20 arvolla ja käsitellä niitä yksi kerrallaan.

LUO LISTA

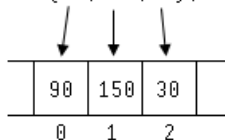
Jos käytämme peruselementtejä, kuten numeroita, on hyvin helppo tehdä lista:

```
int[] numerot = {90,150,30};
```

Aaltosulut tarkoittavat, että sisällä ei ole vain yksittäinen kokonaisluku, vaan kokonainen lista kokonaislukuja, joiden sisällä on useampia arvoja. Niinpä voimme täyttää tämän listan aaltosulkeiden sisällä olevilla arvoilla.

Tietokone luo tarpeeksi muistipaikkoja, jotta jokaiselle arvolla löytyy sopiva laatikko:

```
int[] numbers = {90,150,30};
```



Tästä syystä käytämme sanaa int, koska Processingin täytyy tietää jokaisen listan laatikon arvo. Jos ne olisivat kuvia, jokainen listan solu tarvitsisi paljon enemmän tilaa.

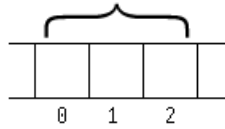
LUO TYHJÄ LISTA

Tämä suoran luomisen menetelmä ei toimi epätavallisempien elementtien, kuten äänien tai kuvien kanssa.

Yleensä lista julistetaan seuraavasti: tyyppi [] LISTA NIMI= new LISTAN TYYPPI [KOHTEIDEN MÄÄRÄ LISTASSA]. Esimerkki alla luo kolmen kokonaisluvun listan. Ole tarkkana, kun luomme listan. Tämä on sarja tyhjiä laatikoita: ne eivät sisällä arvoa.

```
int[] numerot = new int[3];
```

```
int[] numbers = new int[3];
```



KOKONAINEN LISTA

Arvojen sijoittaminen listaan toimii samalla tavalla kuin arvon laittaminen muuttujaan. Täytyy olla myös tarkennus, mihin sijaintiin listalla arvo lisätään.

```
numerot[0] = 90;  
numerot[1] = 150;  
numerot[2] = 30;
```

Nämä arvot numeroidaan automaattisesti arvosta 0 arvoon listan pituus miinus yksi.

LISTAN SISÄLLÖN KÄYTTÖ

Listan elementin käyttö on samanlaista kuin muuttujan käyttö. Määrittelet vain kyseisen elementin sijainnin listalla:

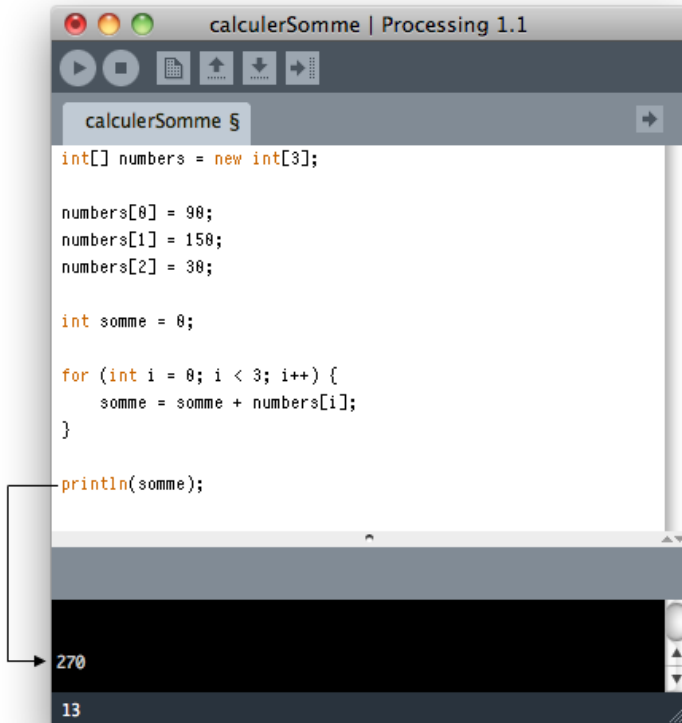
```
println( numerot[0] );
```

Huomaa, että tietokone aloittaa sijainnista 0, ei 1. Jos kutsumme arvoa numerot[2], Processing antaa meille kolmannen sijainnin arvon eikä toisen sijainnin arvoa. Processingin muisti toimii seuraavaksi: nollas ensin, sitten ensimmäinen, toinen jne. Nollas arvo on listan alussa.

Tässä esimerkissä käytämme silmukkaa laskeaksemme kaikkien aiemmin julistettujen elementtien summan:

```
int[] numerot = new int[3];  
  
numerot[0] = 90;  
numerot[1] = 150;  
numerot[2] = 30;  
  
int summa = 0;  
  
for (int i = 0; i < 3; i++) {  
    summa = summa + numerot[i];  
}  
  
println(summa);
```

Tämän ohjelman pitäisi näyttää seuraava ulostulo konsolissasi, Processing-ikkunan alaosassa:



Ensin loimme muuttujan, joka sisältää kaikkien arvojen summan. Se alkaa arvosta nolla.

Tämän jälkeen lisäämme silmukan, joka toistaa itsensä kolme kertaa, ja lisäämme joka kerralla seuraavan arvon summaan. Tässä näemme läheisen suhteen, joka on olemassa listan ja `for ()` -silmukan välillä. Arvo `i` voi kulkea listan arvojen läpi yksitellen, aloittaen arvosta nolla, siirtyen arvoon 1 ja sen jälkeen arvoon 2.

KUVAT SEURAAVAT TOISIAAN

Yksi tapa käyttää listoja on kuvien tuominen Processingiin. Oletetaanpa, että tahdomme tuoda viisi kuvaa Processingiin. Kun emme käytä listoja, voimme kirjoittaa jotain tällaista:

```
PImage kuva1;
kuva1 = loadImage("kuva_1.png");
image(kuva1,0,0);
```

```
PImage kuva2;
kuva2 = loadImage("kuva_2.png");
image(kuva2,50,0);
```

```
PImage kuva3;
kuva3 = loadImage("kuva_3.png");
image(kuva3,100,0);
```

```
PImage kuva4;
kuva4 = loadImage("kuva_4.png");
image(kuva4,150,0);
```

```
PImage kuva5;  
kuva5 = loadImage("kuva_5.png");  
image(kuva5,200,0);
```

Äärimmillään tämä esimerkki on hallittavissa, mutta kirjoitusvirheitä tulee helposti. Jopa kirjoittaessani näitä rivejä tähän kirjaan tein paljon kirjoitusvirheitä tai unohdin muuttaa edelliseltä riviltä kopioimani tekstin.

Parempi tapa kirjoittaa tämä esimerkki on listojen käyttö:

```
PImage[] kuvat = new PImage[5];  
  
kuvat[0] = loadImage("kuva_0.png");  
kuvat[1] = loadImage("kuva_1.png");  
kuvat[2] = loadImage("kuva_2.png");  
kuvat[3] = loadImage("kuva_3.png");  
kuvat[4] = loadImage("kuva_4.png");  
  
image( kuvat[0], 0, 0);  
image( kuvat[1], 50, 0);  
image( kuvat[2], 100, 0);  
image( kuvat[3], 150, 0);  
image( kuvat[4], 200, 0);
```

Listaa käyttäen voimme laittaa kaikki kuvamme yhteen muuttujaan, joka initialisoidaan vain kerran: `PImage [] = new PImage [5]`. Tämän jälkeen voimme täyttää jokaisen tiedoston jokaisen arvon sijaintien listaan.

Mutta näinkin kirjoittamiseen menee liikaa aikaa. Mitä teet, jos sinulla on 200 tai jopa 2000 kuvaa? Kirjoitatko todella kaiken 200 kertaa? Ja mitä teet, kun tahdot tuoda uudelleennimetyn tiedoston?

Tässä on paras tapa tuoda kuvasarja:

```
size(500,500);  
  
PImage[] kuvat = new PImage[20];  
  
for(int i=0; i<images.size(); i++) {  
  kuvat[i] = loadImage("kuva_" + i + ".png");  
  image( kuvat[i], random(width), random(height) );  
}
```

Käyttäen silmukkaa `for()` voimme nyt tuoda niin monta kuvaa kuin tahdomme.

Ensin luomme kuvalistan. Se on aluksi tyhjä lista:

```
PImage[] kuvat = new PImage[20];
```

Tämän jälkeen saamme listan pituuden kysymällä siltä montako kuvaa se sisältää komennolla `images.size()`. Tämän jälkeen voimme käyttää tätä komentoa antamaan `for()` -silmukan toistojen määrän.

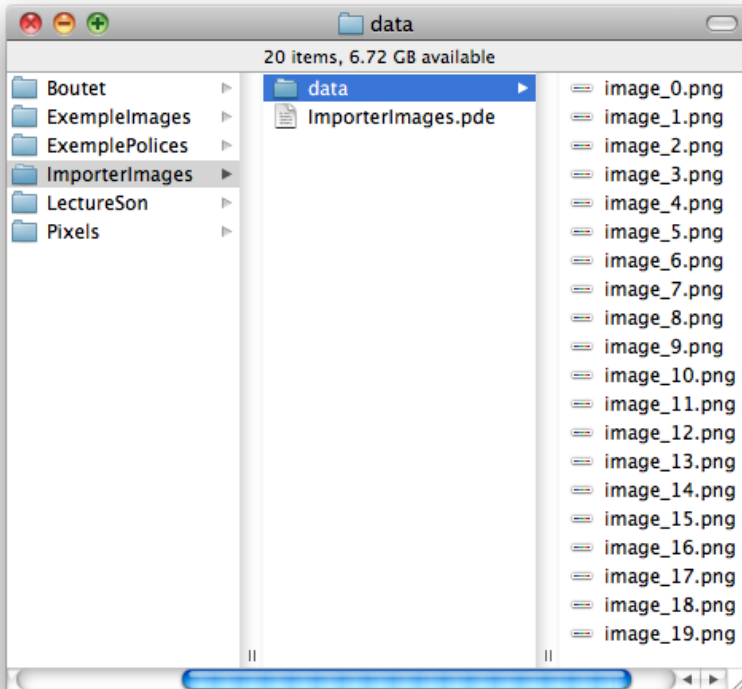
```
for(int i=0; i<images.size(); i++)
```

Voimme jopa automatisoida kuvien tuomisen käyttäen arvoa `i` kutsuaksemme kuvien tiedostonimiä. Merkillä "+" voimme yhdistää kaksi sanaa, mikä merkitsee, että kaksi sanaa tai merkkiä voidaan yhdistää yhteen viestiin.

```
kuvat[i] = loadImage("kuva_" + i + ".png");
```

Liittämällä sanat "kuva_", muuttujan i arvon ja tiedostopäätteen ".png", saamme viestin, joka sisältää tekstin "kuva_42.png". Näin voimme käyttää yhtä riviä koodia luomaan niin monta kuvaa kuin tahdomme. Jos muuttuja i sisältää arvon 9, tuodun tiedoston nimi on kuva_9.png. Jos muuttuja i sisältää arvon 101, tuodun tiedoston arvo on kuva_101.png.

Jotta tämä tekniikka toimisi, sinun täytyy vain laittaa tiedostot valmiiksi data-hakemistoosi. Tässä on datahakemisto, joka sisältää 20 kuvatiedostoa:



Kun jokainen näistä kuvista tulee ohjelmaamme, piirrämme sen johonkin luonnoksessamme.

```
image ( kuvat[i], random(width), random(height) );
```

Tämä lause piirtää kuvan (0,1,2,3,...) satunnaiseen paikkaan x ja y -akseleilla. Se käyttää leveyden ja korkeuden nykyistä arvoa sijaintina. Nämä kaksi arvoa vastaavat kuvan kokoa piirroksessa, mikä merkitsee, että jokainen kuva piirretään satunnaiseen paikkaan piirroksessamme.

PIKSELISARJA

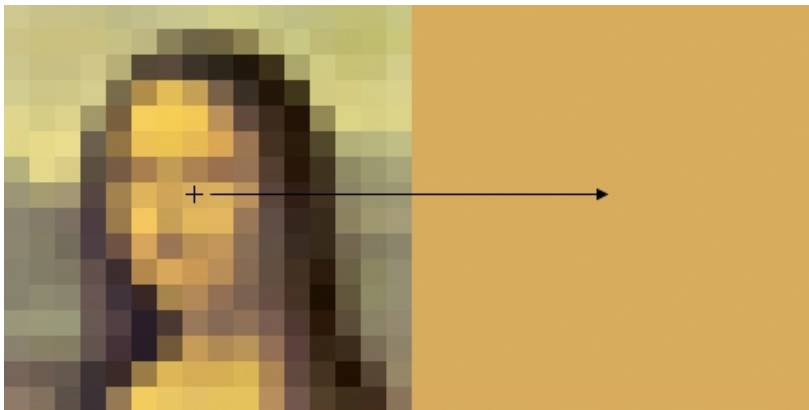
Jos luit kuvia käsittelevän luvun, näit että PiMag-muuttuja ei sisällä yhtä muuttujaa vaan monia, ellei kuvasi ole yhden pikselin korkuinen ja levyinen. Tästä harvinaisesta poikkeamasta huolimatta kuvasi sisältää sarjan muutoksia, jotka edustavat kaikkia kuvan muodostavia pikseleitä. Tätä sarjaa kutsutaan listaksi.

Voit päästä suoraan pikseleihin pyytämällä suoraa pääsyä alimuuttujaan nimeltä "pixels".

Kuvitelkaapa, että meidän täytyy aloittaa mystisen naisen kuva tiedostossa lhooq.png.



Jos tuomme tämän kuvan Processingiin, voimme ottaa tietyn pikselin kuvan menemällä kovalistan sisään. Tätä listaa kutsutaan funktiolla `pixels []`.



```
size(512,256);

PImage lhooq;
lhooq = loadImage("lhooq.png");
image(lhooq,0,0);

int x = 119;
int y = 119;

int index = x + (y * lhooq.width); color c =
lhooq.pixels[index];

noStroke();
fill(c);
rect(256,0,256,256);

stroke(0);
```

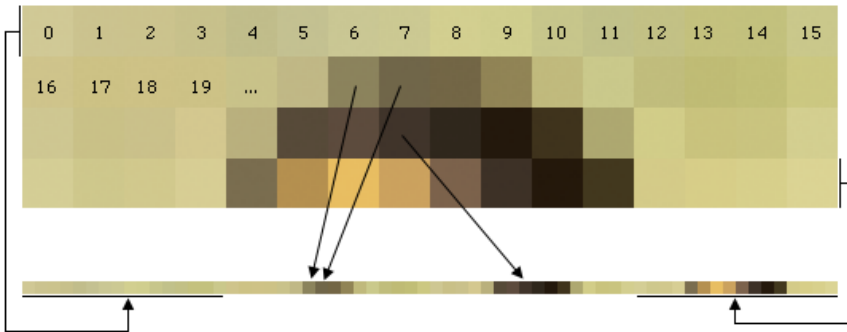
```
line(x-5,y,x+5,y);
line(x,y-5,x,y+5);
```

Kaksi tärkeintä riviä on lihavoitu:

```
int index = x + (y * lhooq.width); color c =
lhooq.pixels[index];
```

Huomautamme, että kuva ei ole mitään muuta kuin lista pikseleitä.

Tyyppin PiMag muuttujat tallentavat pikselien arvot pitkään listaan.

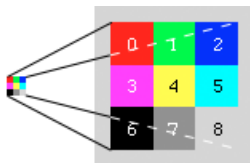


Näemme listan kahdella (x, y) ulottuvuudella, PiMag nähdään yhdestä suunnasta. Jos löydämme arvon sijainnissa $(2,1)$, todellisuudessa sen pitäisi osoittaa, että tahdomme laittaa sen sijaintiin 17 kuvassa. Todellisuudessa rivin pisteen ja vastaavan seuraavan rivin pisteen välillä on 16 pikseliä. Toisin sanottuna meillä on 16 pikseliä jokaista riviä kohden.

Tästä syystä pikselin kaava kirjoitetaan $(x + (\text{leveys} * y))$. Jokaista y -riviä kohden on paljon x -arvoja, joiden arvot täytyy saada.

KUVAN PIKSELIEN VÄRITTÄMINEN

Voit jopa luoda omat PiMag-muuttujasi ja antaa sille pikselien värit. Kuvapikseleitä käytetään nollasta alkaen, kuten mitä tahansa listaa.



Tässä esimerkissä piirrämme kolme kertaa kolmen pikselin ruudukon. Piirrämme suuremman kuvan venyttämällä sen 80×80 pikseliin.

```
PImage img = createImage(3, 3, ARGB);
img.loadPixels();
img.pixels[0] = color(255, 0, 0);
img.pixels[1] = color(0, 255, 0);
img.pixels[2] = color(0, 0, 255);
img.pixels[3] = color(255, 0, 255);
img.pixels[4] = color(255, 255, 0);
img.pixels[5] = color(0, 255, 255);
img.pixels[6] = color(0, 0, 0);
img.pixels[7] = color(127, 127, 127, 255);
```

```
img.pixels[8] = color(255, 255, 255, 0);  
img.updatePixels();  
image(img, 10, 10, 80, 80);
```

Huomaa, että muunnamme Processingissa kuvaan komennolla `loadPixels ()` ja lopetamme kuvan muokkaamisen komennolla `updatePixels ()`. Muuten emme näe listaan tekemiemme muutosten seurauksia.

Huomaa, että voit myös tehdä tällaisen maalauksen käyttäen toistoja.

26. TOISTAMINEN

Toistaminen voi ajaa joukon komentoja loputtomasti. Näin ei ole tarpeen kirjoittaa koodinpätkiä uudestaan.

Varoitus! Toistot eivät luo animaatioita oikeaan aikaan. Ne tehdään niin nopeasti kuin mahdollista. Kun tietokone lukee koodia ja löytää silmukan, se ajaa hakasulkujen sisällä olevan koodin niin monta kertaa kuin silmukka käsketään toistamaan.

Seuraavassa esimerkissä näemme kuusi horisontaalista mustaa viivaa. Ensimmäinen koodi sisältää kymmenen kertaa komennon `line()`, toinen koodi tehdään silmukkana. Molempien koodien tulos on sama.

```
line(0, 0, 100, 0);
line(0, 10, 100, 10);
line(0, 20, 100, 20);
line(0, 30, 100, 30);
line(0, 40, 100, 40);
line(0, 50, 100, 50);
line(0, 60, 100, 60);
line(0, 70, 100, 70);
line(0, 80, 100, 80);
line(0, 90, 100, 90);
```

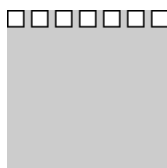
...tai yksinkertaisemmin:

```
for (int i = 0; i < 100; i = i + 10) {
    line(0, i, 100, i);
}
```

FOR-SILMUKKA

Tällainen silmukka toistaa joukon komentoja monta kertaa. Siihen kuuluu muuttuja, jonka arvoon lisätään yksi jokaisen silmukan toiston jälkeen. Silmukan sisäisen muuttujan nimenä käytetään usein kirjainta `i`. For-silmukan muuttuja tarvitsee alkunumeron, maksiminumeron ja lisäyksen. Esimerkiksi: `for (int i = AlkuNumero i < MaksimiNumero i = i + LISÄYS) {}`.

Alla oleva esimerkki näyttää valkoisia neliöitä vierekkäin. Jokainen on 10 pikseliä leveä. Niiden väli on 5 pikseliä. Laitamme ensimmäisen neliön kohtaan 0.0. Seuraava esitetään koordinaateissa 15.0, sitä seuraava koordinaatissa 30.0 ja niin edelleen. Silmukkamme lisää muuttujaan 15 pikseliä joka askeleella. Koska tahdomme täyttää koko piirrostilan, maksimiarvo on kuvan leveys (`width`).



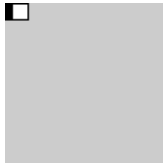
```
for (int i = 0; i < width; i = i + 15) {
    rect(i, 0, 10, 10);
}
```

MITTARIT

Tähän asti olemme käyttäneet silmukoita niin, että silmukan sisäinen muuttuja on suoraan käytettävissä. Edellisessä esimerkissä se antaa meille heti jokaisen x-akselin arvon, johon neliö piirretään.

Silmukoita voidaan myös käyttää laskureina. Niille annettiin minimi- ja maksimimuuttuja, johon lisätään vain 1 jokaisella silmukan suorituskerralla: tämä laskee käskyjen suorituskerrat, joten käytetään käsitettä laskuri.

Tämä menetelmä vaatii enemmän matematiikkaa silmukan sisällä olevan muuttujan käyttöön.



```
for (int i = 0; i < 5; i = i + 1) {  
    rect(i, 0, 10, 10);  
}
```

Edellisessä esimerkissä näemme neliöiden olevan päällekkäin. Koska muuttujalla i on arvot 0-5 ja koska me käytämme muuttujaa i sijoittamaan ne avaruuteen, ne sijoitetaan kohtiin 0.0, 1.0, 2.0, 3.0 ja 4.0. Vain viimeinen näistä on kokonaan näkyvillä. Saadaksemme saman tuloksen kuin edellisessä esimerkissä voimme kertoa muuttujan. Tässä tapauksessa kerrotaan muuttuja arvolla 15, jolloin neliöiden väliin jää 5 pikseliä. Ne ovat tasaisesti 15 pikselin välein. Voimme silti käyttää laskurin arvoa muihin operaatioihin.

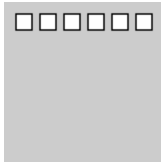


```
for (int i = 0; i < 5; i = i + 1) {  
    rect(i * 15, 0, 10, 10);  
}
```

SISÄKKÄISET SILMUKAT

Silmukat voidaan laittaa toistensa sisään. Tämä tekniikka antaa meidän siirtyä nopeasti kahteen visualisaatioon ja näkemään kolme ulottuvuutta. Kun silmukat ovat toistensa sisällä, meidän täytyy pitää huolta nimestä, joka annetaan jokaisen silmukan muuttujalle. Jos jokaisen muuttujan nimi on i, ohjelma yhdistää silmukat. Jokaisen silmukan jokaisella muuttujalla pitää olla kunnon nimi. Esimerkiksi: i, j, k, jne. Tai jos ne liittyvät ulottuvuuksiin: x, y ja z.

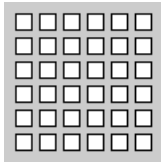
Silmukan alku



```
translate(7, 7);

for (int x = 0; x < 6; x = x + 1) {
  rect(x * 15, 0, 10, 10);
}
```

Kaksi silmukkaa



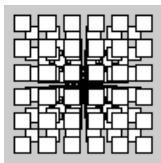
```
translate(7, 7);

// Ensimmäinen silmukka
for (int y = 0; y < 6; y = y + 1) {

  // Toinen silmukka
  for (int x = 0; x < 6; x = x + 1) {
    rect(x * 15, y * 15, 10, 10);
  }
}
```

Kolme silmukkaa

Tässä esimerkissä tutustumme kolmiulotteiseen avaruuteen. Laittaaksemme syvyysulottuvuuden neliöihimme, voimme käyttää metodia `translate ()`.



```
size(100, 100, P3D);

translate(7, 7);

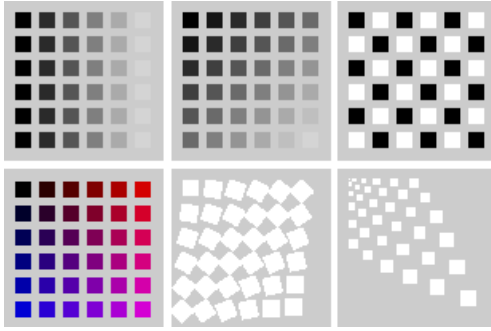
//Ensimmäinen silmukka
for (int z = 0; z < 6; z = z + 1) {
  translate(0, 0, z * -15); //Kohde akselilla z

  //Toinen silmukka
  for (int y = 0; y < 6; y = y + 1) {

    //Kolmas silmukka
    for (int x = 0; x < 6; x = x + 1) {
      rect(x * 15, y * 15, 10, 10);
    }
  }
}
```

Muunnelmat

Tässä on joukko muunnelmia, joissa käytetään metodeja `fill ()`, `scale ()` ja `rotate ()`. Tässä voit nähdä muutokset silmukoiden sisällä.

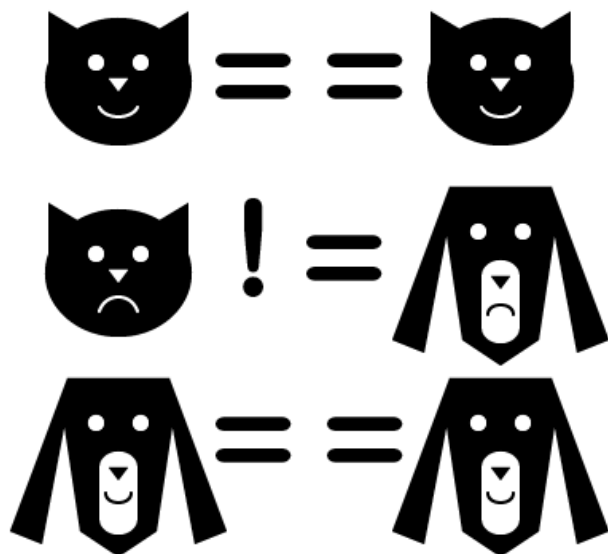


27. EHDOT

Ehtojen avulla tietokone voi tehdä päätöksiä. Se voi muuttaa ohjelmaa riippuen asettamistasi ehdoista. Jos tahdot esimerkiksi muuttaa ohjelmasi ulkoasua ajan kuluessa, voit laittaa mustan taustan ehdoksi kellonajan 10-18 ja valkoisen taustan ehdoksi muut kellonajat. Tämä kysely "paljonko kello on" on ehto.

VERTAILU

Perusehto on vertailu. Ennen kuin voit käyttää vertailua, meidän täytyy tehdä Processingissa kysymys. Tämä kysymys on melkein aina vertailu.



Jos kyselyn tulos on "tosi", Processing ajaa komennon. Jos kyselyn tulos on "epätosi", se suorittaa toisen komennon. Englanniksi tosi on "true" ja epätosi on "false".

Ehdon syntaksi on `if(TESTI) {}`. TESTI on operaatio, joka vertailee kahta arvoa, ja pääättelee onko tulos tosi vai epätosi. Jos tulos on tosi, Processing tekee aaltosulkujen välillä olevat komennot. Komento else voi käsitellä myös tilanteen, jossa ehto ei ole tosi eikä epätosi. Sen toiminnot ovat myös sulkujen sisällä. Voit laittaa näiden sulkujen sisälle niin monta komentoa kuin tahdot.

Samanarvoisuus

Käytämme kaavaa `if(arvo1==arvo2)` tarkastamaan kahden muuttujan samanarvoisuuden. Seuraava esimerkki merkitsee "Sano että on keskipäivä, jos metodi `hour()` antaa arvon 12."

```
if (hour() == 12) {  
  println("On keskipäivä!");  
}
```

```
} else {  
    println("Ei ole keskipäivä!");  
}
```

Tulos on muulloin kuin kello 12:00 ja 12:59 välillä:

```
Ei ole keskipäivä!
```

PIENEMPI KUIN JA SUUREMPI KUIN

Voimme tarkastaa että arvo on pienempi tai suurempi kuin toinen arvo käyttäen operaattoreita < ja >. Seuraavassa esimerkissä kirjoitamme ulostuloon onko aamu vai eikö.

```
if (hour() < 12) {  
    println("On aamu!");  
} else {  
    println("Ei ole aamu!");  
}
```

Ohjelman tulos kello 12:59 jälkeen:

```
Ei ole aamu!
```

PÄÄTÖKSEN YHDISTELMÄ

Komennot if ja else voidaan yhdistää useampien tapausten käsittelemiseksi.

```
if (hour() < 12) {  
    println("On aamu!");  
} else if (hour() == 12) {  
    println("On keskipäivä!");  
} else {  
    println("Ei ole aamu!");  
}
```

Tämän ohjelman tulos kello 12:59 jälkeen on:

```
Ei ole aamu!
```

TESTIEN YHDISTÄMINEN

Monta testiä voidaan yhdistää samaan päätökseen, jotta valinnasta tulee tarkempi. Operaattorit && (ja) sekä || (tai) voivat yhdistää testejä. Jos esimerkiksi tahdomme tietää onko yö vai päivä, voimme erottaa myöhäisen yön ja aikaisen aamun tunnit loppupäivästä:

```
if (hour() < 6 && hour() > 20) {  
    println("On yö!");  
} else {  
    println("Ei ole yö!");  
}
```

Ohjelman tulos kello 16:50 on:

```
Ei ole yö!
```


28. MUUTTUJAT

Muuttuja on tietty arvo, jonka tietokone tallentaa muistiinsa. Se on kuin pakkaus, jonka koko sopii tietynlaiselle tiedolle. Sitä luonnehtii nimi, jonka avulla voimme helposti käyttää sitä.

	42	491	33.145	5	« Salut tout le monde ! »	true	
	x	y	angle	i	message	vie	

On erilaisia muuttujatyyppejä: kokonaisnumeroita (int), desimaalinumeroita (float), tekstiä (String) ja tosi/epätosi -arvoja (boolean). Desimaali, kuten 3.14159 ei ole tyyppiä int vaan tyyppiä float. Huomaa, että käytämme desimaalinumeroille pistettä emmekä pilkkua. Niinpä kirjoitamme 3.14159. Muuttujat voidaan luoda näin:

```
float x = 3.14159;  
int y = 3;
```

Muuttujan nimi voi sisältää kirjaimia, numeroita ja muita merkkejä, kuten alaviivan. Kun ohjelma kohtaa tämän muuttujan nimen, se voi lukea tai kirjoittaa tähän tilaan. Seuraavana on lista muuttujista yksinkertaisten esimerkkien kanssa. Muuttujalla on tyyppi, nimi ja arvo, joka voidaan lukea ja jota voidaan muuttaa.

INT

Processingin syntaksissa voit tallentaa kokonaisluvun, kuten 3, int - tyyppiseen muuttujaan.

```
int kokonaisluku;  
kokonaisluku = 3;  
print(kokonaisluku);
```

```
3
```

FLOAT

Muuttuja float on desimaalinumero, kuten 2.3456.

```
float desimaali;  
desimaali = PI;  
print(desimaali);
```

```
3.1415927
```

DOUBLE

Muuttuja double on myös desimaalinumero, mutta tarkempi kuin float.

```
double pitka_desimaali;  
pitka_desimaali = PI;  
print(pitka_desimaali);
```

```
3.1415927410125732
```


BOOLEAN

Muuttujalla boolean on vain kaksi tilaa. Nämä tilat ovat tosi (true) ja epätosi (false). Sitä käytetään ehdoissa tutkimaan onko ilmaisu tosi vai väärä.

```
boolean onkotosi;
onkotosi = true;
println(onkotosi);
```

true

CHAR

Muuttujaan char voidaan tallentaa kirjoitusmerkki. Huomaa heittomerkkien käyttö lainausmerkkien sijasta.

```
char kirjain;
kirjain = 'A';
print(kirjain);
```

A

STRING

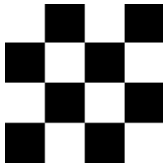
Käytetään tekstin tallentamiseen. Huomaa lainausmerkkien käyttö.

```
String teksti;
teksti = "Moi!";
print(teksti);
```

Moi!

COLOR

Tallentaa värin. On usein hyödyllinen, kun tahdot käyttää samoja värejä.



```
noStroke();
color valkoinen = color(255, 255, 255);
color musta = color(0, 0, 0);
```

```
fill(valkoinen); rect(0, 0, 25, 25);
fill(musta); rect(25, 0, 25, 25);
fill(valkoinen); rect(50, 0, 25, 25);
fill(musta); rect(75, 0, 25, 25);
```

```
fill(musta); rect(0, 25, 25, 25);
fill(valkoinen); rect(25, 25, 25, 25);
fill(musta); rect(50, 25, 25, 25);
fill(valkoinen); rect(75, 25, 25, 25);
```

```
fill(whitew); rect(0, 50, 25, 25);
fill(black); rect(25, 50, 50, 25);
fill(whitew); rect(50, 50, 75, 25);
fill(black); rect(75, 50, 100, 25);

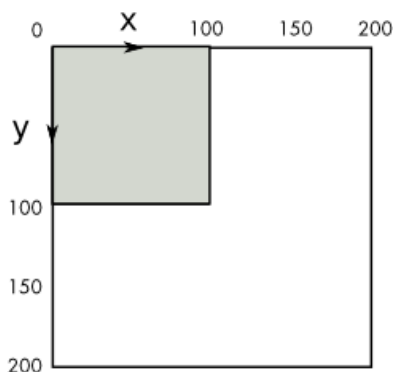
fill(black); rect(0, 75, 25, 25);
fill(whitew); rect(25, 75, 50, 25);
fill(black); rect(50, 75, 75, 25);
fill(whitew); rect(75, 75, 100, 25);
```

29. MUUTOKSET

Toistaiseksi olemme suunnitelleet muotoja sovelluksemme ikkunassa, aina suhteessa ikkunan vasempaan yläkulmaan.

Muutoksilla on mahdollista siirtää tätä alkupistettä, mutta myös määritellä uudelleen akseleiden suunta ja jopa muuttaa näiden viivojen skaalaa.

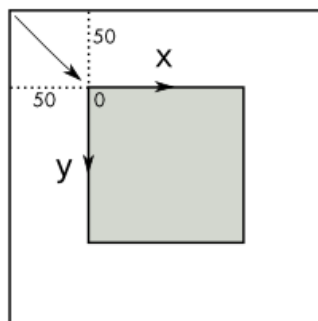
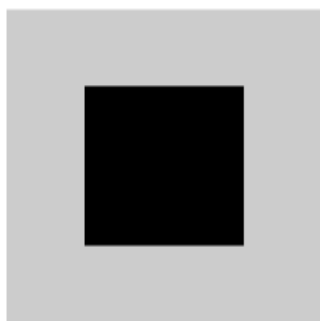
Oletusarvoisesti Processing asettaa alkupisteen seuraavasti:



```
size(200, 200);  
noStroke();  
fill(0);  
rect(0, 0, 100, 100);
```

SIIRTO

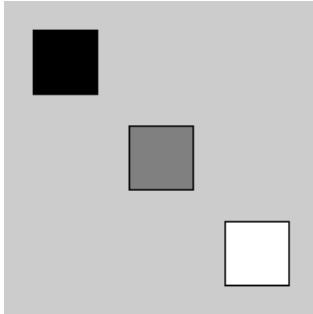
Alkupisteen muuttamiseksi on komento `translate()`. Voimme siirtää x-akselia (horisontaalisesti) ja y-akselia (vertikaalisesti). Kerromme `translate()` -komennolle, kuinka paljon tahdomme siirtää näitä akseleita. Seuraavassa esimerkissä siirrämme alkupistettä 50 pikseliä suunnassa x ja 50 pikseliä suunnassa y. Tulisi huomata, että `translate()` ei vaikuta myöhemmin piirrettyihin geometrisiin muotoihin.



```
size(200, 200);
```

```
noStroke();
fill(0);
translate(50, 50);
rect(0, 0, 100, 100);
```

Komennon `translate()` linkittäminen antaa sinun siirtää muotoja, kuten alla näkyy.



```
size(200,200);

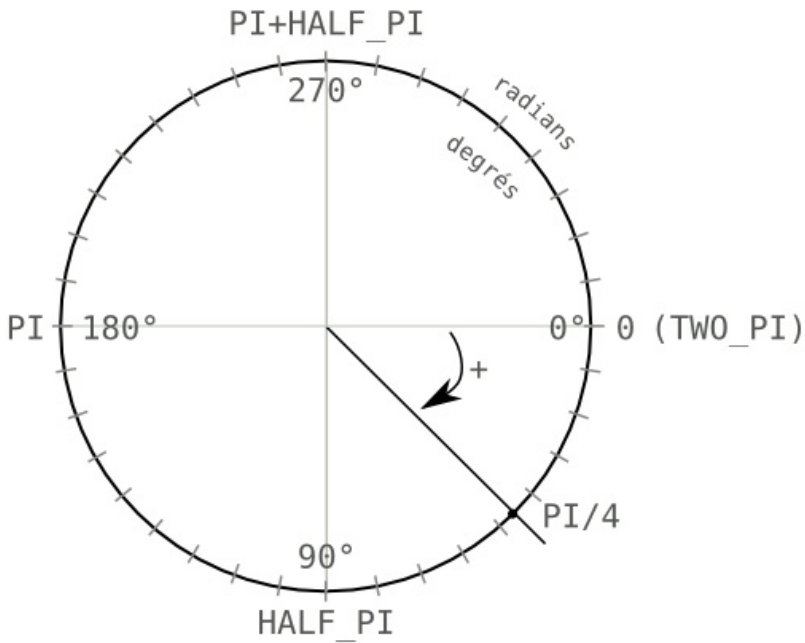
// Musta
fill(0);
translate(20,20);
rect(0,0,40,40);

// Harmaa
fill(128);
translate(60,60);
rect(0,0,40,40);

// Valkoinen
fill(255);
translate(60,60);
rect(0,0,40,40);
```

KÄÄNNÖS

Voimme siirtää koordinaattipiirroksen alkupistettä. Nyt pyöritämme akseleita. Komennolla `rotate()` x ja y vaihtavat suuntaa. `rotate()` ottaa parametriksi numeron, joka esittää pyörimisliikkeen astetta, eli kuinka akselimme pyörivät suhteessa ikkunaamme. Positiiviset numerot osoittavat pyörimistä myötäpäivään.



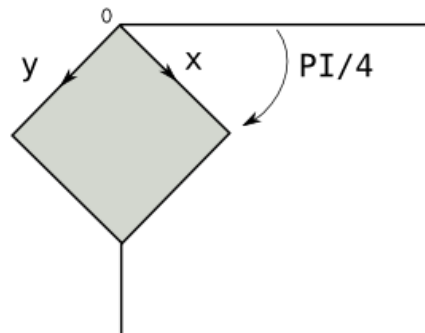
Unités de mesure

Olemassa on kaksi mittausjärjestelmää kulman mittaamiseen: radiaani ja asteet. Oletusarvoisesti Processing toimii radiaanilla, mutta yleensä on helpompi ajatella asteina. Esimerkiksi 180 astetta on u-käännös.

Processing muuttaa yhden yksikön toiseksi funktioilla `radians()` ja `degrees()`.

```
float d = degrees(PI/4); // muuttaa radiaanin asteiksi
float r = radians(180.0); // muuttaa asteet radiaaniksi
```

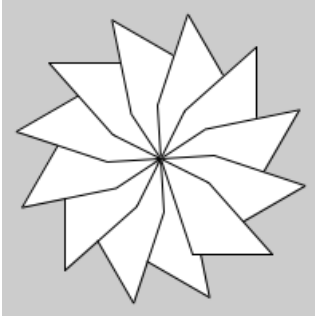
Yksinkertainen esimerkki funktion `rotate()` toiminnasta. Pyöritämme neliötä alkupisteen ympäri.



```
size(200, 200);
```

```
noStroke();
fill(0);
rotate(PI/4);
rect(0, 0, 100, 100);
```

Kuten `translate()`, `rotate()` asetetaan ennen geometrysten muotojen piirtämistä. On mahdollista yhdistää näitä muutoksia, jotka ovat kasautuvia.

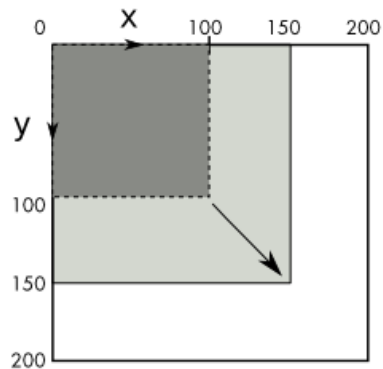


```
size(200,200);
smooth();
translate(width/2, height/2);
for (int i=0;i<360;i+=30){
  rotate(radians(30));
  quad(0, 0, 30, 15, 70, 60, 20, 60);
}
```

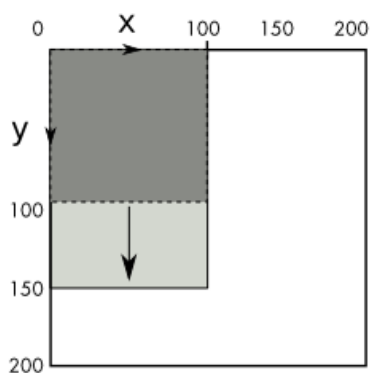
SKAALAAMINEN

Kohteiden kokoa voidaan muuttaa komennolla `scale()`. Tämä komento antaa sinun suurentaa tai kutistaa geometrysten muotojen kokoa. Se ottaa yksi tai kaksi parametriä. Esimerkiksi `scale(0.5)` puolittaa geometrysten muotojen koon, kun taas `scale(2.0)` kaksinkertaistaa sen. Komennolla `scale(1)` ei ole mitään vaikutusta.

Kahdella parametrillä kirjoittaminen yhdistää koon muuttamisen aksleilla x ja y. Esimerkiksi `scale(0.5, 2.0)` puolittaa koon suunnassa x ja kaksinkertaistaa sen suunnassa y.



```
size(200,200);
scale(1.5);
rect(0,0,100,100);
```



```
size(200,200);
scale(1.0,1.5);
rect(0,0,100,100);
```

Kuten komennot `rotate()` ja `translate()`, komennon `scale()` toistaminen kasautuu. Seuraava ohjelma esittää tämän ominaisuuden. Tässä neliöt ovat sisäkkäin kuin venäläiset nuket ja niiden koko kasvaa toistettaessa komentoa `scale()`.



```
size(200,200);
noStroke();

// Musta
fill(0);
scale(1);
rect(0,0,200,200);

// Harmaa
fill(128);
scale(0.5);
rect(0,0,200,200);

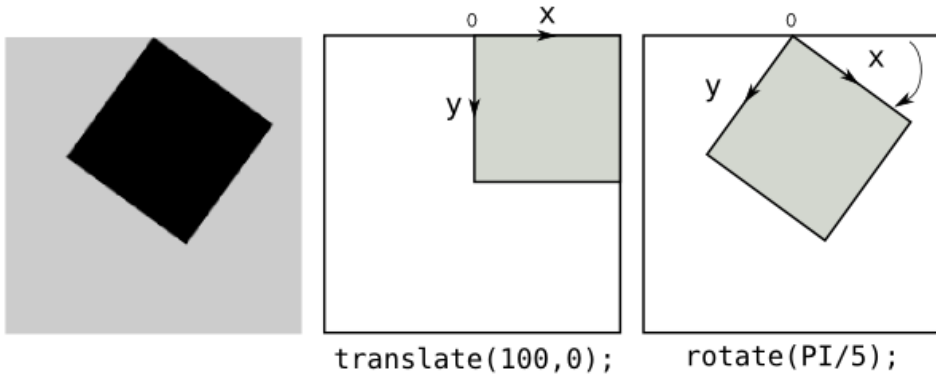
// Valkoinen
fill(255);
scale(0.5);
rect(0,0,200,200);
```

MUUTOSTEN JÄRJESTYS

On mahdollista yhdistää muutama erilainen muutostyyppi. Kuten näimme edellisessä esimerkissä, muutokset kasautuvat asteittain, kun peräkkäiset `translate()`, `rotate()` ja `scale()` -komennot kasautuvat. Jokainen muutos ottaa huomioon edelliset muutokset.

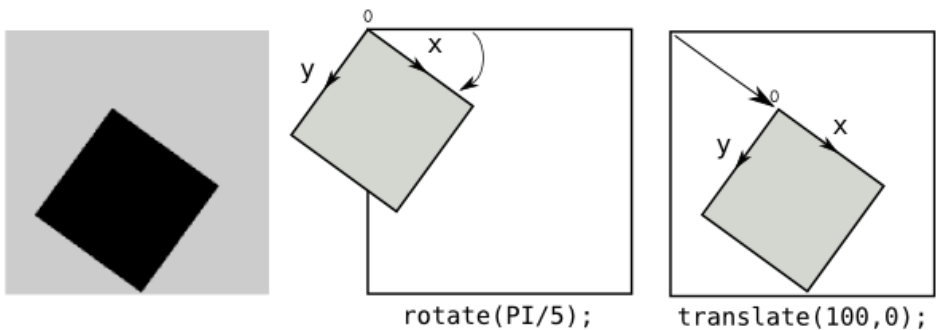
Kun käytetään useampia erilaisia muutostyyppiejä, niiden kirjoitusjärjestys on tärkeä. Ajaessasi autolla "käänny vasempaan ja aja suoraan" on eri komento kuin "aja suoraan ja käänny vasemmalle". Et välttämättä päädy samaan paikkaan. Sama pätee Processingin muutoksiin.

Esittelemme tämän komennon kääntämällä komennot `translate()` ja `rotate()` toisin päin.



```
size(200,200);
smooth();
fill(0);

translate(100,0);
rotate(PI/5);
rect(0,0,100,100);
```



```
size(200,200);
smooth();
fill(0);

rotate(PI/5);
translate(100,0);
rect(0,0,100,100);
```

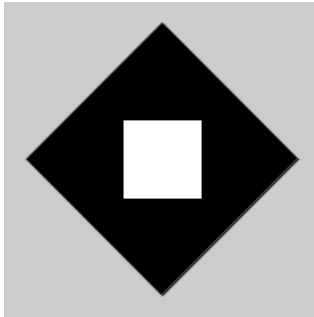
ERISTÄ MUUTOKSET

Olemme nähneet muutoksien kasautuvan asteittain käytettäessä komentoja `translate()`, `rotate()` ja `scale()`. Näemme nyt, kuinka muutokset voidaan tallentaa tiettyä aikana, ja kuinka palauttaa ne myöhemmin, komennon `draw()` aikana.

Käytämme sitä kahteen funktioon, joita käytetään aina pareittain. Nämä komennot ovat `pushMatrix()` ja `popMatrix()`. Näemme lopuksi, miksi nämä komennot on nimetty niin oudosti.

Kahdessa seuraavassa esimerkissä näemme seuraavat muutokset:

- * A: Aluksi ikkunan vasen ylänurkka.
- * B: Alkupiste ruudun keskellä.
- * C: Alkupiste ruudun keskellä, kieritetään $\text{PI}/4$.



```
size(200,200);
smooth();
rectMode(CENTER);

// Tee kirjanmerkki ruudun keskipisteeseen
translate(width/2,height/2);

// Tallennus A
pushMatrix();

// Muutos B
rotate(PI/4);

// Piirrä musta neliö
fill(0);
rect(0,0,120,120);

// Palautus A
// Tässä pisteessä kirjanmerkki palaa ruudun keskipisteeseen
popMatrix();

// Piirrä valkoinen ruutu, joka ei ota pyöritystä huomioon
fill(255);
rect(0,0,50,50);
```

Voimme käyttää funktioita `pushMatrix()` ja `popMatrix()` tallentamaan nykyisen muutoksen tilan.

Edellisen esimerkin pohjalta laitamme `pushMatrix()` / `popMatrix()` -parin, joka tallentaa ensisijaisen muutoksen (`width / 2`, `height / 2`).



```
size(200,200);
smooth();
rectMode(CENTER);
noStroke();

// Varmuuskopio A
pushMatrix();

// Muutos B
translate(width/2,height/2);

// Tallennus B
pushMatrix();

// Muutos C
rotate(PI/4);

// Piirretään musta neliö
fill(0);
rect(0,0,120,120);

// Palautus B
popMatrix();

// Piirrä valkoinen neliö, joka ei ota huomioon
// pyöritystä rotate(PI/4)
fill(255);
rect(0,0,50,50);

// Palautus A
popMatrix();

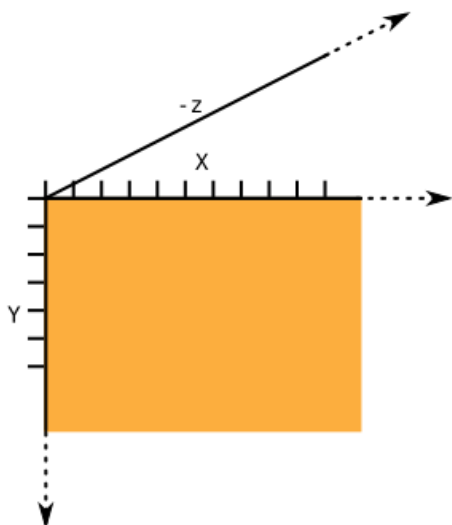
// Piiretään harmaa laatikko
fill(128);
rect(0,0,100,100);
```

KOLMIULOTTEINEN MUUTOS

Kaikki muutokset, joista olemme keskustelleet, ovat mahdollisia myös kolmiulotteisessa piirtämisessä. Processing siirtyy kolmiulotteiseksi, kun kutsumme komentoa `size()`:

```
size(300,300,P3D);
```

Tässä tilassa Processing määrittelee z-akselin osoittavan ruudun pohjaa kohden.

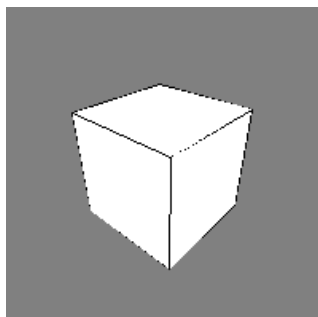


Matkan ja skaalan muutokset kirjoitetaan lisäämällä kolmas parametri. Esimerkiksi siirtyminen ruudun keskipisteeseen x ja y -akselia pitkin mahdollistaa piirtämisen kuin muodot olisivat kaukana meistä, voisimme kirjoittaa jotain tällaista:

```
translate(width/2, height/2, -100);
```

Pyöriä varten tarvitsemme kolme uutta komentoa: `rotateX`, `rotateY` ja `rotateZ`. Nämä kierittävät akseleita x, y ja z.

Processing sisältää funktiot yksinkertaisten kolmiulotteisten muotojen piirtämistä varten. Näihin sisältyvät kuutiot ja pallot. Luomme kuution, joka kiertää akselien x ja y ympäri, ja joka asetetaan hiiren sijaintiin.



```
float rx = 0;
float ry = 0;
float z = 100;

void setup() {
  size(200,200,P3D);
}

void draw() {
  background(128);
  rx = map(mouseX, 0,width,-PI,PI);
  ry = map(mouseY, 0,height,-PI,PI);

  translate(width/2,height/2,z);
  rotateX(rx);
  rotateY(ry);
  box(30);
}
```

```
}
```

Esittelemme uuden funktion `map()`, joka muuntaa arvojen joukon uudeksi arvojen joukoksi. Yksinkertainen esimerkki selittää tätä konseptia:

```
float v = 100;  
float m = map(v,0,200, 0,1); // m on 0.5
```

Tässä esimerkissä `map()` muuntaa arvon 100 välillä `[0, 200]` vastaavaksi arvoksi välillä `[0, 1]`. Palautettu arvo on 0.5 m.

Ohjelmissamme tämä funktio mahdollistaa `mouseX` -arvon muuttamisen välillä 0 ja leveys vastaavaksi arvoksi välillä `-PI` ja `PI`.

Funktiot `pushMatrix()` ja `popMatrix()` toimivat myös kolmiulotteisessa grafiikassa, jossa voidaan tallentaa ja palauttaa muutokset. Tämä on paras tapa piirtää kolmiulotteinen maailma, jossa on joukko liikkuvia esineitä. Näin vältetään monimutkaiset matemaattiset konseptit.

Kaikki Processingin muutokset tallennetaan 16 numeron sarjaan, jota kutsutaan matriisiksi (englanniksi `matrix`). Näitä numeroita muutosfunktiot muuttavat suoraan. Voit tulostaa tämän taulukon komennolla `printMatrix()`.

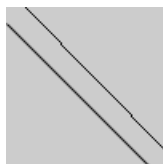
30. VIIVATYYLIT

Rajaviivojen ja geometrysten muotojen tyyliä voidaan muuttaa, jotta vältetään virheiden ilmestyminen viivojen leikkauskohtiin. Tähän tarkoitukseen käytetyt komennot esitellään alla.

SMOOTH

Metodi `smooth()` mahdollistaa reunojen tasoittamisen. Sen avulla vältetään portaikkoefekti, joka tulee näkyville diagonaalisissa viivoissa.

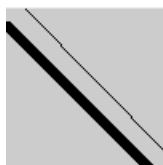
```
line(10, 0, 100, 90); // Ilman tasoitusta  
  
//Aktivoimme tasoituksen  
smooth();  
line(0, 10, 90, 100); // Viiva tasoitettu
```



STROKEWEIGHT

Metodi `strokeWeight()` mahdollistaa viivan paksuuden muuttamisen.

```
line(10, 0, 100, 90); // Yhden pikselin paksuinen viiva  
strokeWeight(5); // Asetetaan paksuudeksi viisi pikseliä  
line(0, 10, 90, 100); // Viiden pikselin paksuinen viiva
```



STROKECAP

Metodi `strokeCap()` muuttaa viivan lopun ulkoasua. Tämä metodi ei ole hyödyllinen muodoissa. Sen arvot voivat olla `SQUARE`, `PROJECT` tai `ROUND`. Oletusarvoisesti käytetään arvoa `ROUND`. Tämä metodi ei toimi OpenGL tai P3D -tilassa.

```
strokeWeight(10); // Määrittelemme paksuudeksi 10 pikseliä  
strokeCap(ROUND);  
line(20, 40, 60, 80);  
  
strokeCap(PROJECT);  
line(20, 20, 80, 80);  
  
strokeCap(SQUARE);  
line(40, 20, 80, 60);
```



STROKEJOIN

Metodi `strokeJoin()` muuttaa kulmien ulkoasua. Sillä voi olla arvot MITER, BEVEL tai ROUND. Oletusarvoinen tila on MITER. Tämä metodi ei toimi OpenGL tai P3D -tiloissa.

```
size(300, 100); // Muuttaa kuvan kokoa

strokeWeight(10); // Määritellään paksuus 10 pikseliä

strokeJoin(MITER); // Suorakulmat
rect(20, 20, 60, 60);

strokeJoin(BEVEL); // Karkeat kulmat
rect(120, 20, 60, 60);

strokeJoin(ROUND); // Pyöristetyt kulmat
rect(220, 20, 60, 60);
```



LIITTEET

- 31. YLEISIÄ VIRHEITÄ
- 32. ULKOISET KIRJASTOT
- 33. DOKUMENTAATIO VERKOSSA
- 34. ARDUINO
- 35. TÄSTÄ OPPAASTA
- 36. LISENSSI

31. YLEISIÄ VIRHEITÄ

UNEXPECTED TOKEN

Tämä virheviesti tulee usein, mikäli unohdat kirjoittaa puolipisteen ";" rivin loppuun, tai et ole sulkenut aaltosulkeita. Ohjelma näkee silloin kahden rivin komennon, jolloin syntaksi on väärin.

```
int kokonaisluku = 1
int murtoluku = 0.1;
```

unexpected token: int

Oikea koodi on:

```
int kokonaisluku = 1;
int murtoluku = 0.1;
```

CANNOT FIND ANYTHING NAMED

Tämä virhe syntyy, kun kutsut muuttujaa, jota ei ole olemassa. Tarkasta, että olet julistanut muuttujan.

```
int numero2 = numero1 + 5;
```

Cannot find anything named "nombre1"

Oikea koodi on:

```
int numero1 = 10;
int numero2 = numero1 + 5;
```

FOUND ONE TOO MANY...

Tämä virheviesti tulee, kun et ole sulkenut koodiblokkia aaltosululla.

```
void draw() {
}

void setup() {
```

Found one too many { characters without a } to match it.

Oikea koodi on:

```
void draw() {
}
void setup() { }
```

ARITHMETICEXCEPTION: / BY ZERO

```
int jakaja = 0;
println(1 / jakaja);
```


ArithmeticException: / by zero

Nollalla ei voi jakaa. Tarvittaessa voit varmistaa, että ohjelma ei yritä jakaa nolllalla.

```
int jakaja = 0;
if (jakaja != 0) {
    println(1 / jakaja);
} else {
    println("Ei voi jakaa nolllalla!");
}
```

CANNOT CONVERT ... TO ...

Tämä virhe tulee, kun yrität laittaa muuttujaan arvon, joka ei sovi muuttujan tyyppiin.

```
int a = 10;
float b = 10.5;

a = b;
```

cannot convert from float to int

Voimme laittaa murtoluvun (float) muuttujaan, joka on tarkoitettu kokonaisluvulle (int), mutta silloin meidän täytyy muuntaa luku kokonaisluvuksi. Oikea koodi on:

```
int a = 10;
float b = 10.5;

a = int(b);
```

ARRAYINDEXOUTOFBOUNDSEXCEPTION

Tämä virhe tulee, kun yrität päästä tietokentän (array) ulkopuolella olevaan osatekijään. Seuraavassa esimerkissä taulun koko on 3 ja yritämme päästä laatikkoon 4.

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 20;
numbers[2] = 5;

println(numbers[4]);
```

ArrayIndexOutOfBoundsException: 4

Oikea koodi on:

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 20;
numbers[2] = 5;

println(numbers[3]);
```

NULLPOINTEREXCEPTION

Tämä virhe tulee, kun yrität päästä muuttujaan, jota ei ole muistissa. Kun yrität esimerkiksi päästä objektiin, jota ei ole vielä alustettu.

```
Ball myBall;

void setup() {
  myBall.x = 10;
}

class Ball {
  int x;
  int y;
}
```

NullPointerException

Oikea koodi on:

```
Ball myBall = new Ball();

void setup() {
  myBall.x = 10;
}

class Ball {
  int x;
  int y;
}
```

32. ULKOISET KIRJASTOT

Elämä on täynnä jännittäviä ongelmia, jotka odottavat ratkaisuaan. Kun kirjoitat ongelman ratkaisevan ohjelman, tahdot varmaankin jakaa hyödyn muiden kanssa. Jos julkaiset koodisi vapaalla lisenssillä, kuten GNU General Public License (GPL), muut voivat hyötyä siitä, sen sijaan että keksisivät pyörän uudelleen.

Kirjasto on kokoelma luokkia, joita voidaan käyttää uudestaan kaikissa projekteissamme. Loputtomasti mielenkiintoisia Processing-kirjastoja on olemassa. Voit ladata kirjaston verkosta, purkaa sen arkiston, ja laittaa sen hakemistoon **libraries**. Hakemisto **libraries** on hakemiston **Processing** alla.

Joukko kirjastoja tulee Processingin mukana. Käyttääksesi kirjastoa ohjelmassasi voit valita sen nimen hakemistossa **Sketch > Import Library**. Nämä kirjastot ovat vapaita: käyttäjät voivat vapaasti ajaa, kopioida, jaella, tutkia, muuttaa ja parannella niiden koodia.

MISTÄ KIRJASTOJA LÖYTYY?

Monet kirjastot eivät tule Processingin mukana, vaan ne täytyy ladata kirjaston ohjelmoijan verkkosivulta. Näin jakelu on helpompaa ja käyttäjät voivat saada kirjaston uusimman version.

Yleisimmät kirjastot löytyvät kuitenkin Processingin verkkosivulta osoitteesta: <http://processing.org/reference/libraries/>.

KIRJASTON ASENTAMINEN

Asennamme nyt kirjaston, jonka lataamme netistä.

Käytämme kirjastoa ShapeTween, koska sitä on helppo käyttää. Se tarjoaa joukon tapoja elementtien animointiin. Tällainen työkalu on hyödyllinen siistien animaatioiden tekemiseen.

Sanalla "tween" kuvataan usein animaatioita ja kuvasiirtymiä. Tämä sana tulee englannista ja merkitsee kuvia, jotka piirretään avainruutujen väliin perinteisessä animaatioissa. "Tween"-ruutu luo liikkeen vaikutelman. Kuvassa luodaan liikettä lisäämällä animaatiota liikkeen alku- ja loppupisteen välille.

Processing-sivujen Libraries-sivulta löytyy linkki ShapeTween-sivuille: <http://www.leebyron.com/else/shapetween/>.

Download & Install

[shapetween.zip](#)

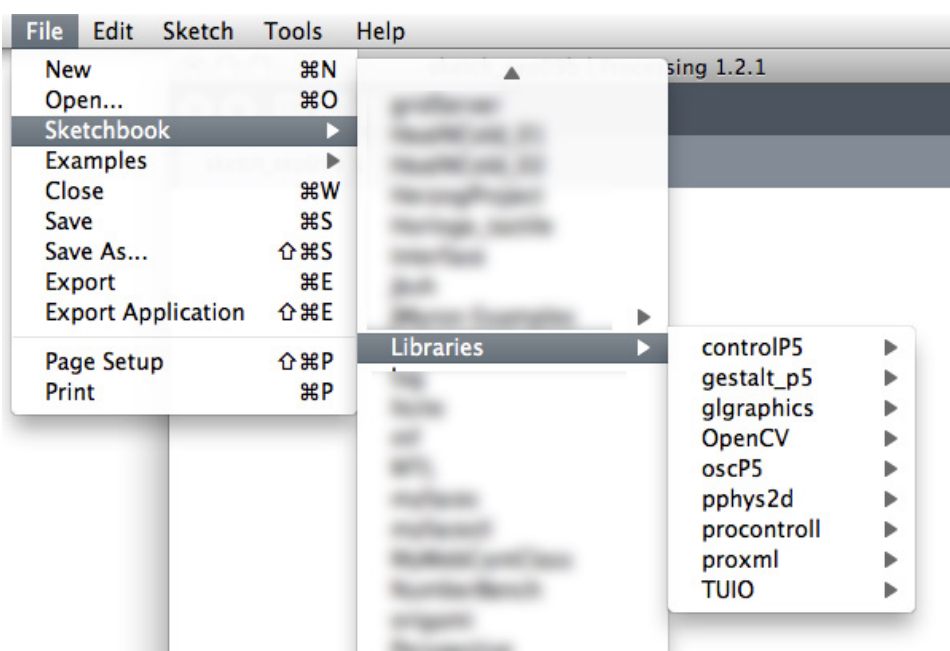
Unzip into the Processing "libraries" folder

Sketch → Import Library → shapetween

Lataa arkisto ja pura zip-tiedosto. Puretun arkiston pitäisi sisältää hakemisto, jolla on kirjaston nimi, ja joka sisältää kirjaston tiedostot. Tämä hakemisto täytyy siirtää Processing-hakemiston alla olevaan Libraries-hakemistoon. Tämän jälkeen voit käynnistää Processingin uudestaan.

KOKEILE ESIMERKKIÄ

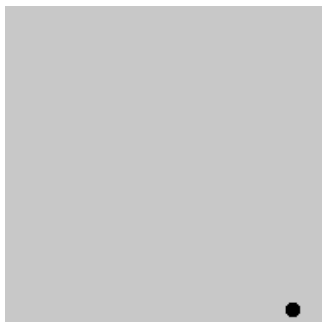
Hyvät kirjastot sisältävät luokkiensa, metodiensa ja attribuuttiensa dokumentaation. Kirjastojen mukana tulee myös esimerkkejä. Jos esimerkkejä on tullut, voit kokeilla niitä hakemistosta **File > Sketchbook > Examples**. Kirjastomme nimen pitäisi olla siellä. Huomaa, että tämä toimii vain käyttöjärjestelmissä Mac OS X ja Windows. GNU/Linuxissa et näe esimerkkejä tässä hakemistossa.



Jos kirjasto ei sisällä esimerkkejä, meidän täytyy hakea esimerkki netistä ja kopioida ja liittää se. Harjoitustyönä kopioimme ja liitämme esimerkin verkkosivulta ShapeTween.

Kirjaston verkkosivulla näet joukon esimerkkejä otsikolla Example Code. Napsauta esimerkkiä Basic Tween nähdäksesi hyvin yksinkertaisen hallitun liikkeen esimerkin.

Kun olet saanut esimerkin toimimaan, voit muokata sitä, ja käyttää sitä omien kokeilujesi pohjana.



```
import megamu.shapetween.*;

Tween ani;

void setup(){
  size( 200, 200 );
  ani = new Tween(this, 2, Tween.SECONDS);
  ani.start();
}

void draw(){
  background(200);

  float x1 = 20;
  float y1 = 30;

  float x2 = 180;
  float y2 = 190;

  float x = lerp( x1, x2, ani.position() );
  float y = lerp( y1, y2, ani.position() );

  fill(0);
  ellipse( x, y, 8, 8 );
}

void mousePressed(){
  ani.start();
}
```

KAUPALLISET KIRJASTOT

Joskus lataat kirjaston ja huomaat, että se ei sisällä lähdekoodia. Kun katsot verkkosivua, näet sanat "All Rights Reserved" ilman sen kummempia selityksiä. Tässä tapauksessa kirjasto ei ole vapaa vaan kaupallinen. Voit käyttää sitä vapaasti, mutta ilman pääsyä sen koodiin. Et voi tutkia tai muuttaa sen toimintaa. Se ei ehkä toimi Processingin seuraavan version kanssa, jolloin et voi korjata sen yhteensopivuusongelmia. Jos nämä asiat ovat sinulle tärkeitä, ole tarkkana. Tarkasta aina että kirjaston mukana tulee lähdekoodi ja että koodissa on viittaus vapaaseen lisenssiin. Free Software Foundationin sivulla on lista lisensseistä, jotka sopivat yhteen GPL:n kanssa: <http://www.fsf.org/>.

33. DOKUMENTAATIO

VERKOSSA

Processingin verkkosivu tarjoaa monta resurssia Processingin käytön aloittamiseen: oppitunteja, koodiesimerkkejä jne. Se tarjoaa myös sivun, joka listaa kaikki Processingin tarjoamat metodit, ja kuvaa tarkalleen niiden toiminnan havainnollisten esimerkkien kanssa.

OPPITUNNIT

Processing-sivujen Learning-osasto (<http://www.processing.org/learning/>) tarjoaa kahdenlaista sisältöä: askel askeleelta etenevän opetusohjelman ja monimutkaisempia koodiesimerkkejä, jotka esittelevät käytännön sovelluksia.

↳ [Tutorials](#) \ [Examples](#): [Basics](#), [Topics](#), [3D](#), [Library](#) \ [Books](#)

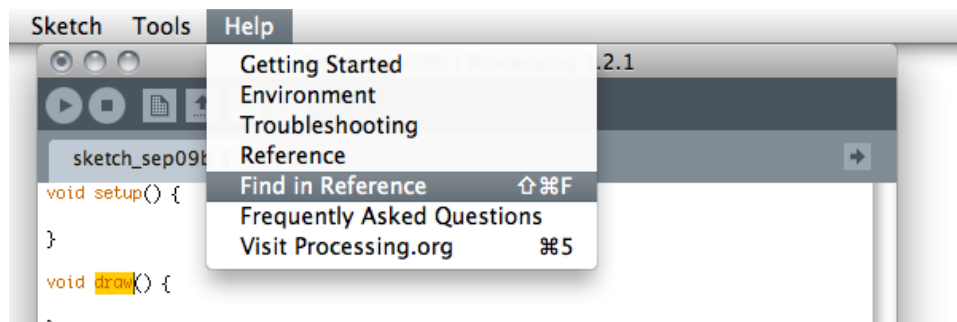
Useimpien esimerkkien koodi tulee Processingin mukana. Ne löytyvät hakemistosta **File > Examples**.

HAKEMISTO

Processing-sivuston hakemisto-osa (<http://www.processing.org/reference/>) tarjoaa kattavan listan Processingin tarjoamista metodeista. Ne on ryhmitelty aiheen mukaan. Jokaista kuvaillaan käyttäen kuvaesimerkkejä, syntaksin tarkkaa kuvailua ja listaa argumenteista.

Mielenkiintoisimpia näiden sivujen osia ovat osat Syntax ja Parameters. Niissä kerrotaan, mitä sulkujen () väliin voi laittaa syntaksin mukaan metodin ajamiseksi.

Pääset tähän hakemistoon suoraan Processing-ympäristöstä. Tehdäksesi tämän voit valita sanan (sana täytyy valita kokonaan) ja mennä hakemistoon **Help > Find in Reference**. Voit tehdä tämän myös näppäinkomennolla **CTRL + SHIFT + F** (Windows ja Linux) tai **CMD + SHIFT + F** (Mac).



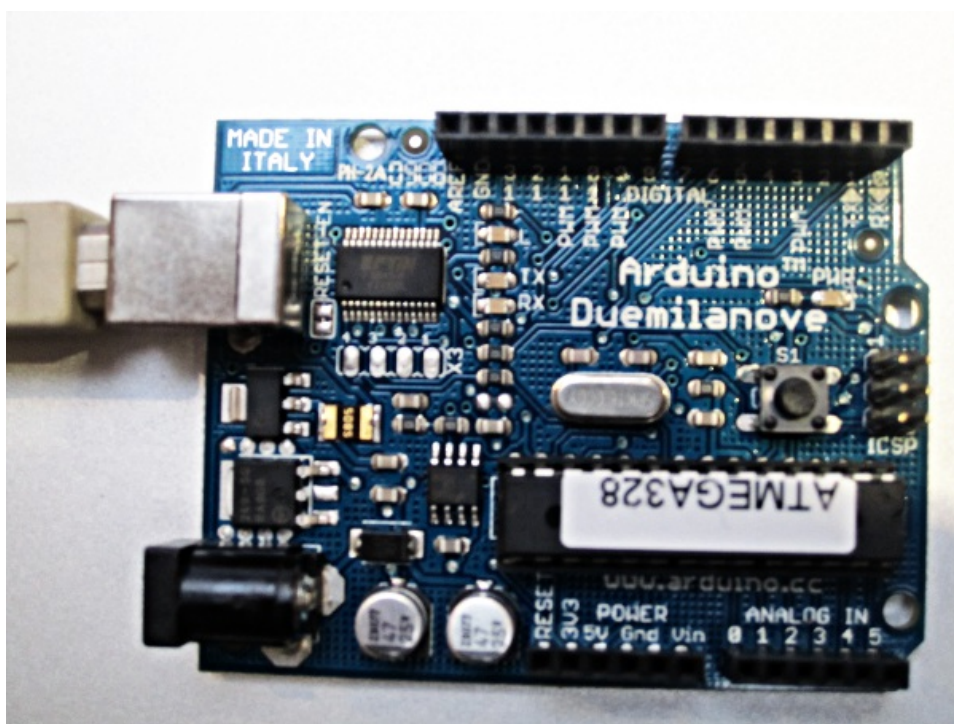
KESKUSTELUPALSTA

Processingin keskustelupalsta (<http://forum.processing.org/>) on toinen avun lähde. Monet ihmiset käyttävät sitä kuvatakseen ongelmiaan ja saadakseen vastauksia. Siellä on myös monia koodinpätkiä, joita voit lisätä ohjelmiisi.

34. ARDUINO

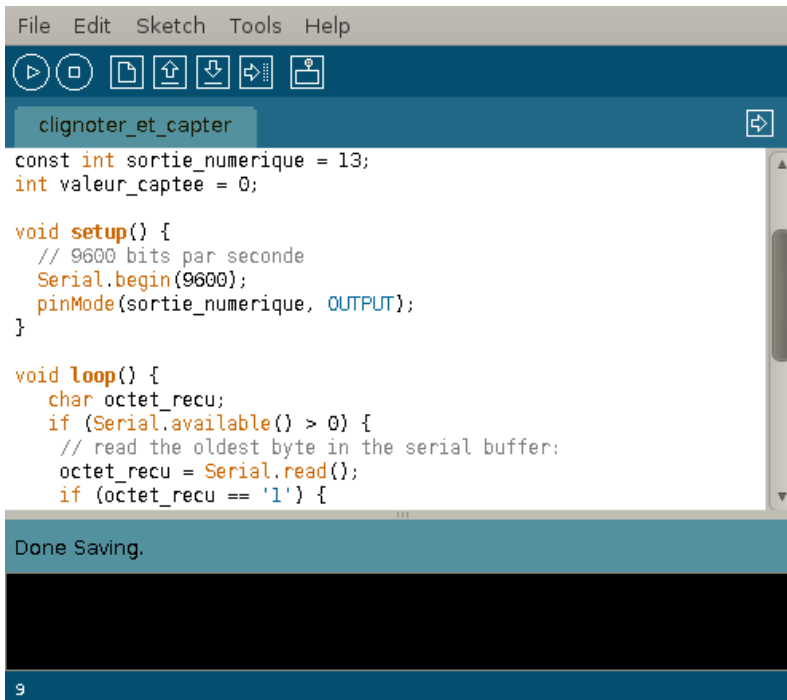
Arduino on avoin alusta elektroniikkaprototyyppien rakentamiseen. Se on mikrokontrolleri, jossa on sisään- ja ulostuloja, joihin voidaan kytkeä sensoreja ja aktuaattoreita (eli fyysiseen maailmaan vaikuttavia osia). Se sisältää myös ohjelmointiympäristön, joka perustuu Processingiin. Ohjelmointiympäristön avulla voit ohjelmoida elektronista mikrokontrolleria. Arduino on Processingin jatkokehittelyprojekti, se näyttää loogiselta seuraavalta askeleelta, kun tahdotaan kehittää fyysisessä maailmassa toimivaa interaktiivisuutta. Kun olet Processing-taituri, on Arduino sinulle jo tuttu.

ARDUINO-MIKROKONTROLLERI



Arduino on elektroninen piiri, joka voidaan kytkeä USB:n avulla tietokoneeseen, jolloin mikrokontrolleriin voidaan ladata sitä ohjaava koodi. Mikrokontrolleri on pieni prosessori, jossa on analogisia sisään- ja ulostuloja. Elektroninen laite on vapaa (muttei ilmainen), aivan kuin Arduino-ohjelmistokin. Voimme tutkia sen piirroksia ja suunnitella sen pohjalta kehittyneempiä versioita.

ARDUINO-OHJELMISTO



```
File Edit Sketch Tools Help
clignoter_et_capter
const int sortie_numerique = 13;
int valeur_captee = 0;

void setup() {
  // 9600 bits par seconde
  Serial.begin(9600);
  pinMode(sortie_numerique, OUTPUT);
}

void loop() {
  char octet_recu;
  if (Serial.available() > 0) {
    // read the oldest byte in the serial buffer:
    octet_recu = Serial.read();
    if (octet_recu == '1') {
```

Done Saving.

9

Mikrokontrolleriin lähettämämme koodi kostuu sarjasta komentoja, jotka käynnistetään, kun Arduinoon kytketään virta. Kun osaat ohjelmoida Arduinoa, voit kytkeä sen Processing-ohjelmaasi.

VIRTUAALISEN JA FYYSISEN VUOROPUHELU

Sensoreita käytetään mittaamaan fyysistä maailmaa, aktuaattorit voivat tehdä toimenpiteitä fyysiselle maailmalle. Voimme käyttää sensoreita vaikuttamaan Processing-ohjelmaan. Vastaavasti Processing-ohjelma voi käyttää aktuaattoreita vaikuttamaan fyysiseen maailmaan. Ohjelma voisi esimerkiksi lukea meteorologisista sensoreista, liikesensoreista tai yksinkertaisista ohjausnapeista tulevan tiedon. Tämän pohjalta ohjelma voisi hallita fyysiseen maailmaan vaikuttavia valoja, moottoreita ja muita laitteita. Tämä voi rikastaa ihmisten interaktiivista kokemusta.

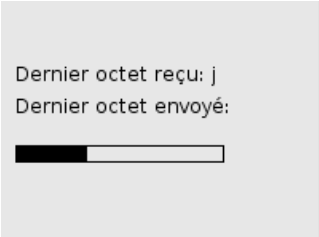
VIESTINTÄ

Tietokoneet kommunikoivat toistensa kanssa käyttäen sähköisiä impulsseja. Impulssit lähetetään yksi kerrallaan, bittien sarjana. Bitti voi olla tosi tai epätosi. Se on numero yksi tai numero nolla. Bitit ryhmitellään yleensä kahdeksan bitin paketeiksi. Me kutsumme näitä paketeiksi tai tavuiksi. ASCII-merkistöä käyttävä kirjain on esitetty kahdeksalla bitillä. Voimme helposti viestiä Arduinon kanssa käyttäen kirjaimia.

ESIMERKKI VIESTINNÄSTÄ KIRJAIMIEN

AVULLA

Tässä on yksinkertainen esimerkki kirjaimen lähettämisestä ja vastaanotosta Arduinolla. Välilyönnin painaminen kääntää päälle LED-valon Arduinon digitaalisessa ulostulossa 13. Arvot, jotka luetaan analogisesta sisääntulosta 0 muuttavat ohjelman näyttämän suorakulmion kokoa.



Dernier octet reçu: j
Dernier octet envoyé:

PROCESSING-KOODI

Lähtettäminen ja vastaanottaminen Processingilla on helppoa, kun käytämme kirjastoa `processing.serial`. Oikea sarjaporttinumero täytyy määritellä ja samoin oikea lähetysnopeus. Tässä lähetysnopeutemme on 9600 bittiä sekunnissa.

```
/**
 * Yksinkertainen koodi Arduinon ja Processingin kommunikaatioon
 */
import processing.serial.*;

// Muuta tarvittaessa portin numero
int portin_numero = 0;
Serial porttini; // Sarjaporttia hallinnoiva objekti
char viimeinen_ulos = ' '; // Viimeinen lähetetty bitti
char viimeinen_sisaan = ' '; // Viimeinen vastaanotettu bitti
void setup() {
  size(200, 150);
  println("Porttilista: \n" + Serial.list());
  String portin_nimi = Serial.list()[portin_numero];
  porttini = new Serial(this, portin_nimi, 9600);
}

void draw() {
  if (porttini.available() != 0) {
    // numero muutetaan kirjaimeksi
    viimeinen_sisaan = char(porttini.read());
  }
  background(231);
  fill(0);
  text("Viimeinen vastaanotettu tavu: " + viimeinen_sisaan, 10, 50);
  text("Viimeinen lähetetty tavu: " + viimeinen_ulos, 10, 70);
  int leveys = int(viimeinen_sisaan - 'a');
  int kerroin = 5;
  stroke(0);
  noFill();
  rect(10, 90, 26 * kerroin, 10);
  noStroke();
  fill(0);
  rect(10, 90, leveys * kerroin, 10);
}

void keyPressed() {
  if (key == ' ') {
    porttini.write('1');
  }
}
```

```

        viimeinen_ulos = '1';
    }
}

void keyReleased() {
    if (key == ' ') {
        porttini.write('0');
        viimeinen_ulos = '0';
    }
}

```

Tieto lähetetään yhtenä kirjaimena. Arduinon analogisen sisääntulon 0 lukema muutetaan kirjaimeksi a-z. Menetät hieman tarkkuutta, mutta tämä näyttää hyvin, kuinka kirjaimet koodataan numeroilla.

KOODI ARDUINOLLE

Arduinon puolella esitämme analogisen sisääntulon 0 mittaaman arvon kirjaimella a-z. Arduino sytyttää ulostuloon 13 liitetyn LEDin. Se tulee Arduinon mukana, joten ei ole tarvetta juottaa sitä tai kasata piiriä prototyyppilaudalle. Tämä esimerkki toimii Arduinossa ilman mitään piiriä. Voit kytkeä sisääntuloon 0 analogisen potentiometrin, jos tahdot näyttää jotain muuta kuin hälyä.

Ladataksesi seuraavan koodin Arduinoon joudut asentamaan Arduino-ohjelman ja opettelemaan sen toiminnan.

```

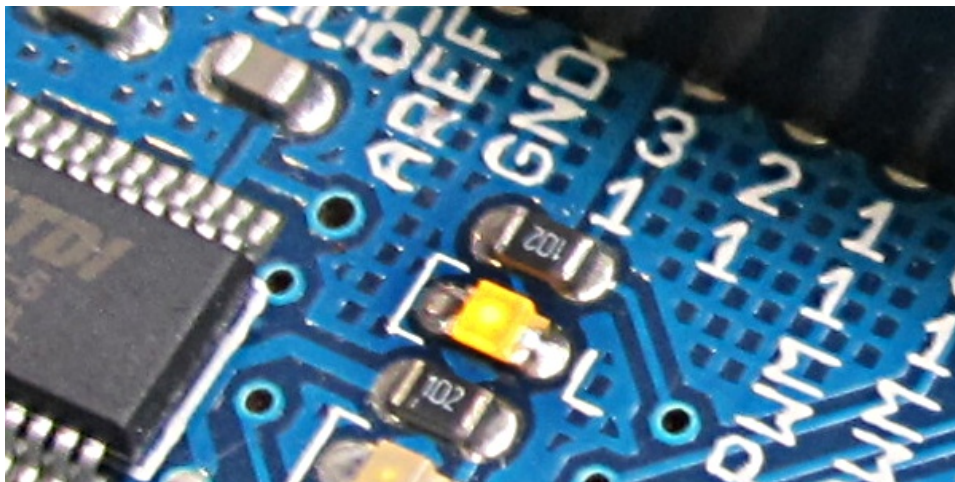
/**
 * Tämä koodi täytyy ladata Arduinoon
 *
 * Viestii tietokoneen kanssa lähettääkseen arvon
 * analogisesta sisääntulosta 0 ja käyttääkseen
 * ulostuloa numero 13, joka on pieni lamppu.
 */
const int sisan_analoginen = 0;
const int ulos_numero = 13;
int numero = 0;

void setup() {
    // Viestinopeus = 9600 baudsiaSerial.begin(9600);
    pinMode(ulos_numero, OUTPUT);
}

void loop() {
    char bitti_sisaan;
    if (Serial.available() > 0) {
        // Katsotaan saammeko signaalin
        bitti_sisaan = Serial.read();
        if (bitti_sisaan == '1') {
            digitalWrite(ulos_numero, HIGH);
        } else {
            digitalWrite(ulos_numero, LOW);
        }
    }
    // Lue analoginen sisääntulo 0:
    numero = analogRead(sisan_analoginen);
    char laheta = 'a' + map(numero, 0, 1023, 0, 25);
    Serial.print(laheta);
    // Pieni hengähdystauko
    delay(10);
}

```

Kun Arduino on kytketty USB-porttiin ja koodi on lähetetty Arduinoon, käynnistä Processing-ohjelma. Kun painat välilyöntiä, ikkunassa näkyy jotain ja Arduino-laudan ulostulon 13 LEDi syttyy palamaan.



Tämä koodi muistuttaa Processingin koodia, eikö totta? Tämä Arduinon koodi toimii mikrokontrollerissa, mutta on näin lyhyt! Itse asiassa se ei edes ole Javaa vaan C++ -ohjelmointikieltä. Koska Arduino on kehitetty samalla filosofialla kuin Processing, kehittäjät voivat tehdä kokeiluja elektroniikalla ja mikrokontrollereilla räjäyttämättä päätään.

Tällä esimerkillä loimme hyvin yksinkertaisen tiedonsiirtoprotokollan. Kaikki tekemämme on helppo ymmärtää. Tämä ei kuitenkaan ole kovinkaan joustava tai toimiva ratkaisu. Jos tarvitset tehokkaamman protokollan, voit löytää joukon kirjastoja ja esimerkkejä Arduinon sivuilta. Viestiprotokolla Messenger (<http://www.arduino.cc/playground/Code/Messenger>) on aika yksinkertainen ja sopii luultavasti tarpeisiisi.

35. TÄSTÄ OPPAASTA

Tämä opas on aluperin kirjoitettu ranskankielisessä FLOSS Manuals -dokumentaatiowikissä. Alkuperäinen opas löytyy osoitteesta <http://fr.flossmanuals.net/processing/>.

Oppaan ensimmäisessä versiossa oli noin 260 sivua. Se kirjoitettiin viidessä päivässä osana Pariisissa pidettyä kirjapyrähdyistä syyskuussa 2010. Kirjapyrähdykset on menetelmä, jossa joukko yhteen kokoontuneita ihmisiä voi luoda uuden oppaan muutamassa päivässä. Wikin käyttö kirjapyrähdykseen mahdollistaa myös etäosallistumisen internetin kautta. Lisätietoa kirjapyrähdyksistä löydät FLOSS Manualsista: <http://fi.flossmanuals.net/booksprints/>.



TÄSTÄ OPPAASTA

Suomennettu Processing-opas on tuotettu suomenkielisessä FLOSS Manuals -dokumentaatiowikissä. Suomenkielisen FLOSS Manuals löydät osoitteesta <http://fi.flossmanuals.net>.

MIKÄ ON FLOSS MANUALS?

FLOSS Manuals on dokumentaatiowiki vapaille ja avoimen lähdekoodin ohjelmille. Se on kansainvälinen verkosto ja Amsterdamissa perustettu säätiö. Alunperin wiki oli vain englanniksi, mutta se on lokalisoitu myös suomeksi, persiaksi ja hollanniksi.

Lyhenne FLOSS tulee sanoista Free/Libre/Open Source Software. Suomeksi lyhenne käännetään yleensä sanoiksi Vapaat ja Avoimen Lähdekoodin Ohjelmat eli VALO. FLOSS Manuals voi siis suomentaa VALO-käyttöohjeiksi.

FLOSS Manualsın suomenkielinen versio on avattu vuonna 2009. Tavoitteenamme on sekä suomentaa englanninkielisen sivuston käyttöohjeita että aloittaa uusien käyttöoppaiden kirjoittaminen Suomessa. Järjestämme käyttöoppaiden pohjalta myös työpajoja.

FLOSS Manuals tarjoaa ilmaisia käyttöoppaita, jotka julkaistaan GPL-lisenssin alla. Uusi käyttöopas tuotetaan kaikkein tehokkaimmin kirjapyrähdyksessä, jossa ohjelmasta kiinnostuneet ihmiset kokoontuvat kirjoittamaan käyttöopasta. Kirjapyrähdys voi usein tuottaa uuden käyttöoppaan muutamassa päivässä.

Verkosto on saanut maineen laadukkaan dokumentaation tuottajana. Sitä tukevat avainprojektit, kuten One Laptop Per Child ja Archive.org. Kieliyhteisö laajenee nopeasti - käyttöoppaita käännetään nyt yli kahdellekymmenelle kielelle.

DOKUMENTAATION TARVE

Vapaiden ja avoimen ohjelmien maailmassa on paljon ohjelmoijia. Usein teknisistä kirjoittajista näyttää kuitenkin olevan pulaa. Yleisen vitsin mukaan uusi käyttäjä yrittää käyttää avoimen lähdekoodin ohjelmaa, mutta häntä neuvotaan lukemaan lähdekoodia.

Tämän vuoksi tarvitsemme FLOSS Manualsia. Ihannetilanteessa dokumentaatio vastaa loppukäyttäjien taitotasoa. Loppukäyttäjä ei luultavasti ole kiinnostunut käytetyn ohjelmointikielen hienouksista tai vastaavista teknisistä seikoista. Loppukäyttäjä tahtoo käyttää ohjelmaa.

Niinpä dokumentaatiossa käytetyn kielen tulisi olla ymmärrettävää maallikoille. Sitä ei tulisi kirjoittaa ohjelmointigurujen teknisellä jargonilla, kaikki tulisi kirjoittaa henkilölle, joka ei ymmärrä tietojenkäsittelytiedettä - mutta tahtoo tuottaa jotain.

Tämä on FLOSS Manualsın idea. Tämän vuoksi tarvitsemme laadukasta dokumentaatiota avoimen lähdekoodin ohjelmille - rakentaaksemme sillan ohjelmien ja potentiaalisten käyttäjien välille.

KÄYTTÄJIEN JA KIRJOITTAJIEN YHTEISÖN RAKENTAMINEN

Pelkkä käyttöoppaan suomentaminen ei riitä. Sitä täytyy jatkuvasti päivittää, laajentaa ja lokalisoida sopimaan paikallisiin olosuhteisiin. Tämän vuoksi jokainen FLOSS Manualsın lokalisaatio tarvitsee kirjoittajien ja lukijoiden yhteisöä.

Wikiin pohjautuva yhteistyö merkitsee sitä, että lukemattomien käyttäjien pienistä lisäyksistä muodostuu valtava määrä sisältöä. FLOSS Manuals tahtoo antaa kaikille wikiin osallistuneille kunnian kirjoituksistaan, minkä vuoksi kirjoittajia pyydetään rekisteröitymään omalla nimellään.

FLOSS Manuals pyrkii luomaan ratkaisun uuden käyttäjän ongelmaan: mistä löydän käyttöohjeet tälle ohjelmalle?

36. LISENSSI

Kaikki kappaleet ovat kirjoittajien tekijänoikeuden alaisia. Jos muuten ei sanota, kaikki luvut tässä käyttöoppaassa on lisensoitu **GNU General Public License version 2** mukaisesti.

Tämä dokumentaatio on vapaata dokumentaatiota: voit jakaa sitä eteenpäin ja/tai muokata sitä Free Software Foundationin GNU General Public License mukaisesti; joko lisenssin version 2, tai (tahtoessasi) minkä tahansa myöhemmän version.

Dokumentaatiota jaellaan siinä toivossa, että se on käyttökelpoisa, mutta **ILMAN MITÄÄN TAKUUTA**; ilman edes **MYYTÄVYYDEN** tai **TIETTYYN KÄYTTÖÖN SOPIVUUDEN** oletettua takuuta. Katso lisätietoja GNU General Public Licensestä.

Tämän dokumentaation mukana olisi pitänyt tulla kopio GNU General Public Licensestä, mikäli sitä ei tullut kirjoita osoitteeseen Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

ALKUPERÄISEN RANSKANKIELISEN VERSION TEKIJÄT

A propos de ce manuel

© FLOSSManuals 2010

Modifications:

Alexandre Quessy 2010

Christian Ambaud 2010

Elisa Guerra 2010

Lionel Tardy 2010

L'animation de plusieurs objets

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Elisa Guerra 2010

Julien Gachadoat 2010

Lionel Tardy 2010

L'animation d'un objet

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Lionel Tardy 2010

Arduino

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Les astuces

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Jerome Saint-Clair 2010

Julien Gachadoat 2010

Lionel Tardy 2010

La documentation en ligne

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Christian Ambaud 2010

Elisa Guerra 2010

Lionel Tardy 2010

Credits

© adam hyde 2010

Modifications:

Lionel Tardy 2010

La vidéo

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Exemples d'utilisations

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Caroline Kassimo-Zahnd 2010

Christian Ambaud 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Lionel Tardy 2010

L'exportation

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Glossaire

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Lionel Tardy 2010

L'impression

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Lionel Tardy 2010

L'installation de Processing

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Caroline Kassimo-Zahnd 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Lionel Tardy 2010

Introduction

© FLOSSManuals 2006

Modifications:

adam hyde 2006, 2007, 2010

adama Dembele 2010

Christian Ambaud 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

La ligne de temps

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Lionel Tardy 2010

La typographie

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Lionel Tardy 2010

L'entrée microphone

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

L'entrée vidéo

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Les bases de Processing

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Caroline Kassimo-Zahnd 2010

Christian Ambaud 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Lionel Tardy 2010

TWikiGuest 2010

Les commentaires

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Jerome Saint-Clair 2010

Lionel Tardy 2010

Les conditions

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Douglas Edric Stanley 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Les couleurs

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Les erreurs courantes

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Elisa Guerra 2010

Lionel Tardy 2010

Les événements clavier

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Les événements souris

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Les formes

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Christian Ambaud 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jean-Francois Renaud 2010

Les images

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Caroline Kassimo-Zahnd 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Les listes

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Les méthodes

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Les objets

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Lionel Tardy 2010

L'espace de dessin

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Les répétitions

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Les styles de bordures

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Les transformations

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Les variables

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

Le texte

© FLOSSManuals 2010

Modifications:

adam hyde 2010

Alexandre Quessy 2010

Douglas Edric Stanley 2010

Elisa Guerra 2010

Horia Cosmin Samoila 2010

Julien Gachadoat 2010

Lionel Tardy 2010

La lecture du son

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Douglas Edric Stanley 2010

Horia Cosmin Samoila 2010

Lionel Tardy 2010

La méthode draw

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Douglas Edric Stanley 2010

Horia Cosmin Samoila 2010

Jerome Saint-Clair 2010

Julien Gachadoat 2010

Lionel Tardy 2010

Les librairies externes

© FLOSSManuals 2010

Modifications:

adam hyde 2010

adama Dembele 2010

Alexandre Quessy 2010

Elisa Guerra 2010

Lionel Tardy 2010

SUOMENTAJAT JA SUOMENKIELISEN VERSION TEKIJÄT

Johdanto

2012

[Tomi Toivio](#)

Esimerkkejä

2012

[Tomi Toivio](#)

Asennus

2012

[Tomi Toivio](#)

Perusteet

2012

[Tomi Toivio](#)

Suunnittelutila

2012

[Tomi Toivio](#)

Muodot

2012

[Tomi Toivio](#)

Värit

2012

[Tomi Toivio](#)

Kuvat

2012

[Tomi Toivio](#)

Tekstit

2012

[Tomi Toivio](#)

Video

2012

[Tomi Toivio](#)

Videon tuominen

2012

[Tomi Toivio](#)

Mikrofoni

2012

[Tomi Toivio](#)

Hiiri

2012

[Tomi Toivio](#)

Näppäimistö

2012

[Tomi Toivio](#)

Tulostaminen

2012

[Tomi Toivio](#)

Verkkoon vieminen

2012

[Tomi Toivio](#)

Äänet

2012

[Tomi Toivio](#)

Draw-metodi

2012

[Tomi Toivio](#)

Objektien animointi

2012

[Tomi Toivio](#)

Monien objektien animointi

2012

[Tomi Toivio](#)

Aikajana

2012

[Tomi Toivio](#)

Kommentit

2012

[Tomi Toivio](#)

Objektit

2012

[Tomi Toivio](#)

Metodit

2012

[Tomi Toivio](#)

Listat

2012

[Tomi Toivio](#)

Toistaminen

2012

[Tomi Toivio](#)

Ehdot

2012

[Tomi Toivio](#)

Muuttujat

2012

[Tomi Toivio](#)

Muutokset

2012

[Tomi Toivio](#)

Rajat

172

2012

[Tomi Toivio](#)

Yleisiä virheitä

2012

[Tomi Toivio](#)

Ulkoiset kirjastot

2012

[Tomi Toivio](#)

Dokumentaatio

2012

[Tomi Toivio](#)

Arduino

2012

[Tomi Toivio](#)

Tästä oppaasta

2012

[Tomi Toivio](#)

Tekijät

2012

[Tomi Toivio](#)

FLOSS MANUALS (SUOMI)



FI.FLOSSMANUALS.NET

Vapaa oppaat vapaille ohjelmille!

GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS