RUBY ON RAILS – Ohjelmointi

JUSSI PEKKA KASURINEN

KÄYTTÖOIKEUDET JA HUOMIOT

Alkuperäisversion 1. painos kesäkuussa 2010, uudelleenjulkaisu ebookina elokuussa 2017.

Tämä painos on e-kirjana julkaistu uusintapainos kirjasta ”Ruby on Rails-Ohjelmointi”, Jussi Pekka Kasurinen, ISBN 978-951-0-37139-8.

Erot alkuperäispainokseen johtuvat julkaisutekniikasta ja siihen liittyvästä uudelleentaitosta.

LUKIJALLE

Hyvä lukija, käsissäsi oleva kirja käsittelee Ruby-ohjelmointikieltä, ja kielelle suunniteltua Ruby on Rails-ohjelmointikehystä. Kirjan ensimmäinen osa käsittelee Ruby-ohjelmointikieltä yleisesti, ja esittelee kielessä käytettävät ohjelmointirakenteet sekä kielen perussyntaksin. Toisessa osassa puhutaan Ruby on Rails-ohjelmointikehyksestä, jonka avulla on mahdollista tehdä pienellä vaivalla alustariippumattomia verkkosivustoja vaativaankin käyttöön. Toisessa osassa keskustellaan myös jonkin verran tietokantojen toiminnasta ja verkkosivujen muotoiluista osana Ruby on Rails-ohjelmointia. Kirja on sisällöltään suunnattu ohjelmointia jo jonkin verran ymmärtäville, jollain ohjelmointikielellä perusteet hallitseville käyttäjille. Kirjan alkuosa on kuitenkin kirjoitettu siten, että kirjaa voi myös käyttää osana ohjelmoinnin perusasioiden opettelua.

Ruby on rakenteeltaan ketterä ja suoraviivainen kieli, jonka avulla on mahdollista luoda ohjelmia pienehköllä määrällä suoranaisia koodirivejä. Kieli on hyvin dynaaminen ja varsin tehokas, ja sitä onkin monesti verrattu sellaisiin kieliin kuten Python tai Perl. Kielen parhaiten tunnettu laajennus, Ruby on Rails-ohjelmistokehys tuo kielen rakenteellisen monipuolisuuden verkkosovellusten maailmaan, mahdollistaen verkkopalvelujen tekemisen ohjelmointikielen rakenteiden avulla.

Kirjan osakokonaisuudet ja harjoitustehtävät on suunniteltu siten, että ensimmäisen osan avulla opetetaan lukemaan ja käyttämään Ruby-ohjelmointikieltä. Kirjan toisessa osassa perehdytään Ruby on Rails-kehyksen toimintaan ja sen perusteisiin esimerkkiprojektien avulla. Projektien tarkoituksena on näyttää kaikki Rails-kehyksen perustoiminnot käytännössä, tarjoten täten mahdollisuuden ymmärtää monimutkaisempia aiheita ja päästä jatkamaan kehyksen kanssa harjoittelua oma-alotteisesti. Kirja on jaettu eri aiheiden mukaisesti lukuihin, jotka olisi hyvä läpikäydä järjestyksessä, mutta jokainen konsepti ja rakenne on pyritty esittelemään mahdollisimman yksinkertaisesti muista rakenteista irrallisena komponenttina. Kirja onkin suunniteltu ajatuksella *”tällaisen kirjan olisin itse halunnut lukea aloittaessani Ruby-ohjelmointikielen parissa”*. Toivottavasti tässä suunnitteluperiaatteessa on mielestäsi onnistuttu.

Miellyttäviä lukuhetkiä!

Jussi Pekka Kasurinen

Lappeenrannassa 30.05.2010

Kirjasta tehty kurssi

Ruby ja Ruby on Rails-kirjan ensimmäisestä osiosta, Ruby-ohjelmoinnista, on kirjoitettu myös erillinen itseopiskelukurssi, joka kattaa Ruby-kielen perusasiat. Kurssi esittelee perusasiat hieman yksityiskohtaisemmin sekä sisältää lisää harjoitustehtäviä ja esimerkkejä, joiden avulla on mahdollista opetella Ruby-ohjelmointikielen käyttämistä.

Verkkokurssi, ja kurssin suorittamiseen tarvittavat työkalut, löytyvät osoitteesta *www.ohjelmointikurssit.com*.

SISÄLLYSLUETTELO

OSA I

Ruby-ohjelmointikieli

Ohjelmointi ja ohjelmointikielet yleisesti

Oliot ja Ruby

Ruby-kielen perusrakenteet ja tietomuodot

Loogiset operaattorit ja valintarakenteet

Toistorakenteet ja moduulikirjasto

Tiedostot ja metodit

Virheiden käsittely ja poikkeukset

Luokat ja omien olioiden luominen

LUKU 1

Ohjelmointi ja ohjelmointikielet yleisesti

Tässä luvussa käsitellään ohjelmoinnin historiaa ja ohjelmointia yleisellä tasolla. Luvussa esitellään Ruby-ohjelmointikieli, vertaillaan sitä muihin tavallisimpiin ohjelmointikieliin sekä käydään läpi mitä kaikkea opettelun aloittamista varten tulisi tehdä.

OHJELMOINNISTA YLEISESTI

Tietokone on tietyssä mielessä varsin yksinkertainen laite. Vaikka nykyaikaiset tietokoneet mahdollistavatkin verkossa surffailun, sähköpostin lähettämisen ja kolmiulotteisien pelimaailmojen luomisen, on tietokone rautatasolla edelleen suoraviivainen laite jossa on muistipaikkoja, rekisterejä sekä näiden väliseen kommunikointiin tarkoitettuja väyliä. Tietokoneet näet toimivat joukolla yksinkertaisia konekielisiä toimintaohjeita, jotka ovat luokkaa ”lue muistipaikasta merkki”, ”Siirry seuraavaan muistipaikkaan” jne. Koska näitä komentoja voidaan suorittaa monella eri tietokoneen komponentilla, kuten suorittimen ytimillä tai näytönohjaimen piirillä miljoonia käskyjä sekunnissa, saattaa siitä saada vaikutelman, että tietokone tekee asioita jotenkin monimutkaisesti.

Tämän laitekokonaisuuden ohjaamisessa tulevatkin kuvaan ohjelmointikielet. Ohjelmointikielet toimivat siten, että niiden kääntäjät tai tulkit muodostavat ohjelmoijan kirjoittamasta lähdekoodista konekielisen version, jota tietokone lukee suorittaessaan ohjelmaa. Käytännössä tämä siis tarkoittaa sitä, että ohjelmointikieli on ainoataan joukko käskyjä, joista voidaan muodostaa tietokoneelle joukko toimintaohjeita.

Tämä perusperiaate ei ole vuosien saatossa muuttunut paljoakaan. Jos otetaan esimerkiksi COBOL-ohjelmointikieli, joka julkaistiin vuonna 1959, ja tehdään sillä ohjelma, joka tulostaa tekstin ”Moi maailma!”, on koodi kutakuinkin seuraavanlainen:

```
01  IDENTIFICATION DIVISION.
02  PROGRAM-ID. MOI-MAAILMA.
03  PROCEDURE DIVISION.
04    DISPLAY 'Moi maailma!'.
05    STOP RUN.
```

Lyhyesti koodi on seuraavanlainen: ensimmäisellä kahdella rivillä kerrotaan ohjelman nimi, kolmannella todetaan ohjelman alkavan, neljäs tulostaa halutun tekstin ja viides ilmoittaa ohjelman loppuvan. Jos kyseistä koodia verrataan tässä kirjassa käytettävään Ruby-ohjelmointikieleen, olisi vastaava koodi kutakuinkin seuraavanlainen:

```
01  # Moi maailman –tulostaja, C:\omatRubyt\moi.rb
02  puts "Moi maailma!"
```

Käytännössä ohjelma on lyhentynyt hieman; Ensimmäisellä rivillä kerrotaan ohjelman yleinen tarkoitus ja sijainti, joskin tällä kertaa kommenttina jota ei olisi pakko käyttää. Toisella rivillä ohjelma laitetaan tulostamaan haluttu rivi. Ainoa varsinainen muutos viidenkymmenen vuoden aikana onkin se, että ohjelman lopetuspaikkaa ei tarvitse käsin ilmoittaa. Periaatteessa ainoa asia, joka viidessäkymmenessä vuodessa on muuttunut on itse ohjelmoitavat laitteet; 50-luvulla tietokone saattoi helposti viedä tilaa kokonaisen kerrostalon kerroksen verran, ja sen käyttämiseen saatettiin tarvita joukko koneenkäyttäjiä. Lisäksi ohjelmointia pidettiin enempi tai vähempi tiettyjen insinööriammattien erikoisosaamisena, ja jokaiselle tietokonemerkille oli käytössä oma ohjelmointikieli. Nykyään ohjelmointitaitojen tarpeen arkipäiväistymisen alkaa kuitenkin huomata jo toimistotyötä tehdessään; on täysin normaalia, että monimutkaista taulukkolaskintaulukkoa käyttäessään tulee tarvinneeksi taitoja ja toimintoja, jotka vielä viisikymmentä vuotta sitten olisi vaatinut toteutuakseen tehokkaan laskentakeskuksen ja joukon ohjelmoijia tehtävän toteutuksen tekemistä varten.

Ohjelmointikielten eri tyypit

Nykyisellään teollisessa ohjelmistotuotannossa voidaan sanoa käytettävän useita, jopa kymmeniä, eri ohjelmointikieliä, tai ohjelmointikielten murteita. Mihin näitä kaikkia sitten käytetään ja mikä niistä tekee keskenään niin erilaisia? Käytännössä ohjelmointikieliä erottaa kaksi asiaa: käyttökohde ja kielen rakenne. Käyttökohde on ehkä helpoin tapa jakaa ohjelmia; jotkin kielet ovat hyviä verkkosivujen rakentamiseen, kuten PHP, toiset taas on tarkoitettu isojen ohjelmien tekemiseen (Java, C++). Osalla kielistä taas käyttötarkoitus voi olla esimerkiksi työskentelyä helpottavissa pienissä ohjelmissa ja kokonaisuuksia yhdistävissä ”liimaukielinä” (Ruby, Python) tai työskentelyä nopeuttavissa käskysarjoissa (Perl). Joskus valintaperusteena voi olla myös ohjelmointikielien käyttöön tarkoitetut työkalut; useimmille kielille kehitystyökalut ovat ensisijaisesti ilmaisia, mutta joissain tapauksissa kielen käyttöön sopii parhaiten kaupallinen ohjelmisto (Matlab, C#), joka käyttää kyseistä kieltä työskentelykielenään.

Teknisessä mielessä ohjelmointikielten jako eri tyyppeihin on kuitenkin mielekkäämpää. Karkein jakoperuste ohjelmointikielille on luultavasti konelähtöisyys vastaan ihmislähtöisyys. Käytännössä tämä tarkoittaa ohjelmointikielten jakoa matalan tason ohjelmointikieliin, jossa ohjelman käskyt muodostuvat tietokoneelle annettavista ohjeista lukea muistipaikkoja, siirtyä koodissa seuraavaan paikkaan tai varata käyttömuistista tilaa syötteen tallentamista varten. Tälle vastapainona voidaan pitää korkean tason ohjelmointikieliä, joissa toiminnot määritellään hyvin yleisellä tasolla, (lue tiedosto, hae verkkosivun sisältö, järjestele taulukko…) ja tarkka suoritus jätetään tietokoneen tulkin hoidettavaksi. Monesti ohjelmointikieliä jaotellaan myös niiden käyttötapojen mukaisesti. Jos kielessä on luontevinta työskennellä olioiden ja luokkien kanssa, voidaan kieltä sanoa olio-pohjaiseksi, jos funktioiden ja muuttujien, proseduraaliseksi tai imperatiiviseksi. Käytännössä tämä ainoastaan määrittelee sen, minkätyylisiä käskyjä ja toimintoja kielen perusrakenteesta löytyy. Itseasiassa, monella uudemmalla ohjelmointikielellä on mahdollista kirjoittaa ohjelmakoodia usealla eri tavalla.

Lisäksi useimmissa tavallisemmissa ohjelmointikielissä esiintyy monesti samankaltaisia rakenteita. Itseasiassa, ohjelmointitaitojen kannalta voidaan väittää, että yhden kielen oppiminen hyvin nopeuttaa merkittävästi kykyä hahmottaa ja oppia käyttämään seuraavaa ohjelmointikieltä. Esimerkiksi jotkin kielet kuten C, Ruby, PHP tai Python, ovat rakenteeltaan niin yleisluontoisia, että suoranaisesti kieltä tuntematon, muuten ohjelmointikieliä ymmärtävä ja ohjelmointitaitoinen henkilö, ymmärtää suurimman osan lähdekoodin toiminnasta vaivatta. Monesti tämä onkin uusien ohjelmointikielien kannalta hyvä asia; jos ohjelmointikieli on jo valmiiksi toisen olemassa olevan kielen kaltainen, on sen oppiminen helppoa ja mikäli kieli toteuttaa jonkin asian tehokkaammin kuin aiemmin olemassa olleet kielet, on se silloin mielekäs opeteltava.

RUBY JA MUUT OHJELMOINTIKIELET

Ruby on ohjelmointikieliä ajatellen varsin uusi tulokas. Kielen ensimmäinen julkinen versio ilmestyi joulukuussa 1995, varsinaisen 1.0-version tullessa saataville vuotta myöhemmin joulupäivänä 1996. Kesti kuitenkin vielä muutamia vuosia ennen kuin Ruby levisi kotimaastaan Japanista laajempaan käyttöön; ensimmäinen kieltä käsittelevä englanninkielinen kirja ilmestyi syyskuussa 2000.

Rubyn suunnittelun lähtökohtana on ollut käytettävyys ja tehokkuus. Kielen aluperäinen suunnittelija, Yukihiro Matsumoto, onkin korostanut tätä ajatusta toteamalla, että perinteinen ohjelmointitapa pyrkii selittämään asioita tietokoneen ymmärtämällä tavalla huolimatta siitä, että loppujenlopuksi tietokoneen tulisi ainoastaan olla laite, jonka avulla ihmiset voivat hyödyntää toistensa tekemää työtä. Tämän vuoksi Rubyn tarkoitus onkin tehdä ohjelmoinnista suoraviivaista, keskittyen ”virittelyn ja viilailun” minimointiin ja ohjelman toteuttamisen yksinkertaistamiseen. Johtuen hyvin samankaltaisista rakenteellisista tarpeista kieltä onkin helppo verrata muihin vastaavia ominaisuuksia sisältäviin kieliin kuten Pythoniin tai Perliin. Kielen suunnittelun pohjana käytetyistä kielistä voidaan mainita vielä lisäksi Lisp, Dylan ja CLU, joista kaikista voidaan nähdä samankaltaisia ominaisuuksia kuin mitä Rubyyn on sisällytetty. Jos kieltä vertaillaan teknisiltä ominaisuuksiltaan muihin kieliin, voidaan sen karkeasti sanoa tekevän asiat yhdellä kolmasosalla Javan vaatimasta koodimäärästä käyttäen neljäsosan muistimäärästä. Laskentanopeudeltaan kieli on kutakuinkin samaa luokkaa verrokkiensa Perlin ja Pythonin kanssa, häviten kuitenkin selvästi esimerkiksi optimoidulle Java-koodille tai C++:lle.

Käyttökohteet

Ruby-ohjelmointikielen tavallisin käyttökohde ei kuitenkaan ole suoranaisesti itsenäisessä ohjelmistokehityksessä, vaan eräänlaisena kokoajakielenä jolla voidaan tehdä työkaluja tai kirjastoja tukemaan suurempaa ohjelmaa. Lisäksi Ruby-ohjelmointikielen vahvuus on sen ohjelmistokehyksissä, frameworkeissa, joiden avulla kielellä voidaan toteuttaa useita erilaisia projekteja helposti. Näistä erityismaininnan arvoinen onkin kirjan toinen pääaihe Ruby on Rails, jonka avulla Ruby-kieltä voidaan käyttää verkkopalveluiden ja –sivustojen tekemiseen; esimerkiksi suositut verkkosivustot Twitter ja Hulu on luotu käyttäen on Rails-ympäristöä. Itse asiassa, Ruby on Rails-ympäristöä voidaan pitää Ruby-kielen tunnetuimpana yksittäisenä sovellusalueena, jonka vaikutus on niin voimakas, että se samalla ohjaa kielelle tarkoitettujen työkalujen kehitystä. Joidenkin Ruby-ympäristöjen yhteydessä voidaankin mainita ns. Rails-singulariteetti, joka tarkoittaa kehitysastetta, jossa ympäristö on ensi kertaa pystynyt täysin toteuttamaan Rails-kehyksen vaatimukset. Toistaiseksi ainoastaan kielen kaksi pääversiota, alkuperäinen MRI Ruby, jota kutsutaan myös nimellä CRuby koska se on kirjoitettu C-kielellä, sekä Java-toteutus JRuby ovat toteuttaneet tämän vaatimuksen. Esimerkiksi .NET-ympäristölle tarkoitettu Ruby-ympäristö IronRuby ei vielä täytä kaikkia vaatimuksia.

Rubyn toimintatavoista

Ruby eroaa useimmista muista ohjelmointikielistä siinä, että kielessä kaikki kappaleet on olioita. Rubyssa olio-rakenne on viety niin pitkälle, että jopa normaalisti pelkät staattiset objektit, kuten kokonaisluvut (1, -5, 32 …) tai operaattorit (+, -, %, & …) sisältävät jäsenfunktioita joilla toimintoa voidaan tutkia tai muokata. Käytännössä oliomaisuus ei kuitenkaan ole välttämätöntä; Rubylla on mahdollista ohjelmoida ilman kattavaa olio-ohjelmoinnin tuntemusta, mutta luonnollisesti siitä on apua, ja olioiden käyttäminen tuleekin nopeasti Rubylla ohjelmoidessa tutuksi.

Ohjelmointityyleinä Ruby tukee useampaa eri lähestymistapaa: esimerkiksi imperatiivista ohjelmointityyliä (kuten C-kieli), olio-pohjaista (Java, C++) tai funktionaalista ohjelmoitia (Lisp, Haskell). Käytännössä tämä siis tarkoittaa sitä, että Rubylla tuotettu lähdekoodi voi poiketa paljonkin eri projektien välillä.

Rubyyn siirtyminen muista ohjelmointikielistä

Tässä kappaleessa käsitellään lyhyesti Rubyyn siirtyminen muista ohjelmointikielistä joilla on samansuuntaisia käyttötarkoituksia kuin Rubylla. Näihin kieliin voidaan lukea esimerkiksi Java, Python, Perlistä ja Rails-kehyksen ansiosta myös PHP. Koska Ruby-kieli on syntaksiltaan hyvin vapaamuotoinen, on siinä yhteneväisyyksiä useisiin muihin ohjelmointikieliin ja siksi rakenne, joka läheisesti muistuttaa toisia kieliä. Tämä luonnollisesti tarkoittaa myös sitä, että Rubyssä on toimintoja, joiden samankaltaisuus, mutta rakenteellinen erilaisuus voivat aiheuttaa ei-toivottuja yllätyksiä. Tämän vuoksi seuraavissa kappaleissa on jotain huomioita kielen pääeroista muihin samaan käyttötarkoituksiin sopiviin, paremmin tunnettuihin kieliin.

Ruby verrattuna Pythoniin

Ruby on ohjelmointikielenä hyvin samankaltainen kuin Python, joten suuria yllättyksiä kieleen siirtyessä ei luultavasti tule tapahtumaan. Esimerkiksi toisto- ja valintarakenteet, moduulien käyttäminen, listat sekä poikkeustenkäsittely toimivat muutamia syntaksisanojen eroavaisuuksia lukuun ottamatta samalla tavalla.

Kuitenkin kielissä on jotain eroavaisuuksia. Olioita käytettäessä on jäsenmuuttujille ja –funktioille tehokkaampi tehokkaampi näkyvyydenmäärittely, sekä tyhjää ilmaisevien arvojen (0, 0.0, "" …) testaamisessa on joitain eroja. Lisäksi johtuen siitä, että sisennystä ja rivitystä ei Rubyssa valvota tai ylipäänsä käytetä muuhun kuin luettavuuden parantamiseen, merkitään koodilohkojen loppumista end-käskysanalla. Käytännössä eroavaisuudet kuitenkin tarkoittavat sitä, että Pythonilla ohjelmoimaan opetellut henkilö on melko lailla kotonaan Rubyn kanssa. Tämän vuoksi onkin lähinnä henkilökohtaisesta mielipiteestä kiinni pitääkö Rubyn syntaksista enemmän vai vähemmän kuin Pythonista.

Jos erityistä eroa halutaan kuitenkin etsiä, löytyy sellainen käyttökohteiden ja laajennusosien puolelta. Rails-kehyksen ansiosta Ruby-ohjelmointikieltä pidetään yleisesti Pythonia kyvykkäämpänä toteuttamaan verkkopalveluita, huolimatta siitä että Pythonille on saatavilla Rails-kehystä vataava Django.

Ruby verrattuna Javaan

Verrattuna Javaan on Ruby jonkin verran erilainen kokemus. Java-kieli on tarkkaan määritelty ja hyvin rakenteinen, kun taas Rubyssä asioita voidaan toteuttaa hyvinkin liberaalisti. Käytännössä tämä siis tarkoittaa sitä, että Ruby-kieli pystyy toteuttamaan asioita pienemmällä määrällä koodia, mutta samalla rakenteen johdonmukaisuus kärsii.

Kuten Javassa, myös Rubyssa on roskienkeruuseen perustuva automaattinen muistinkäsittely, mutta toisin kuin Java, Ruby on tulkattava kieli joten ohjelmia ei tarvitse erikseen saada kääntymään ennen suoritusta vaan se tehdään ajonaikaisesti. Tämä tietenkin tarkoittaa sitä, että ohjelmat tulisi testata jossakin määritn tarkemmin, koska tulkki ei ota kantaa virheellisiin käskyihin ennen kuin tulee aika suorittaa kyseinen rivi. Useimmat Javasta löytyvät rakenteet ovat olemassa myös Rubyssa, mutta niissä on joitain eroavaisuuksia. Eräs näistä eroavaisuuksista onkin se, että olioiden jäsenmuuttujiin viitataan aina metodien kautta. Verkko-ohjelmoinnissa Java-appletit vaativat käyttäjältä oman ajoympäristönsä (Java Runtime Environment), kun taas Rubyn Rails-kehys pääasiassa tuottaa html-kuvausta, ollen tarkoitettu enempi verkkosivustojen toteuttamiseen. eikä verkon yli toimiviin työkaluihin.

Ruby verrattuna PHP:hen

Vaikka Rubylla ja PHPllä ei ensi näkemältä olekaan paljoa yhteistä, on niitä yllättävän usein verrattu toisiinsa. Tämä johtuu Ruby-kielen ohjelmistokehyksestä Ruby on Railsista, joka mahdollistaa Rubyn käyttämisen PHPn tavoin verkkopalveluiden ja verkossa toimivien ohjelmien toteuttamiseen.

Kielissä onkin näin verrattuna paljon yhteistä; Muuttujat toimivat kutakuinkin samalla tavalla, samoin kuin luokkarakenteet. Lisäksi kuten PHPssä, Ruby sisältää suuren joukon valmiita kirjastofunktioita joiden avulla on mahdollista tehdä useita eri asioita. Syntaksin avainsanat toimivat melko lailla samalla tavalla, joskin poikkeuksena null, joka Rubyssa on nil.

Varsinaiset erot muodostuvat kielten erilaisesta toimintaperiaatteesta. Ruby on Rails käsittelee verkkosivuja datamallista, sivun rakenteen määrittelevästä näkymästä ja toiminnot sisältävästä ohjaimesta muodostuvana kokonaisuutena, kun PHP keskittyy pitkälti html-kuvauksen sisään sijoitettuihin itsenäisesti toimiviin koodilohkoihin. Tämän vuoksi Rails voi alkuun tuntua monimutkaiselta, kun taas PHP toimii yksinkertaisesti osana sivun rakennetta. Pitkällä tähtäimellä Rails on kuitenkin vähintään yhtä tehokas johtuen siihen rakennetusta automaatiosta ja oletustoiminnoista, jotka pääsevät tärkeään rooliin kun sivuston ja palvelujen koko kasvaa.

OHJELMOINNIN ALOITTAMINEN

Ruby on siis varsin monipuolinen ohjelmointikieli, johon on mahdollista tutustua usean eri ohjelmointikielen kautta. Kuitenkin jokaisella ohjelmointikielellä on harjoittelun aloittamisessa tiettyjä ongelmia, kuten työkalujen valinta, ohjelmointiympäristön asentaminen tai oikean julkaisuversion valitseminen mikäli niiden kesken on epäselvyyksiä. Usein hyvien työkalujen löytäminen onkin työlästä, ja työkalujen, kuten lähdekoodieditorin, vaihtaminen ainoastaan lisää opeteltavien asioiden määrää. Seuraavassa puhutaankin siitä, mitä kannattaa huomioida aloittaessa Ruby-kielen opiskelemisen.

Työkaluista

Mitä työkaluja sitten tarvitaan ohjelmoinnin harjoitteluun tai esimerkiksi tämän kirjan tehtävien tekemiseen? Varsinaisesti perustyökaluina voidaan pitää esimerkiksi lähdekoodieditoria sekä tietenkin kielen tulkkia tai kääntäjää. Yleisesti Rubya varten kannattaa valita jokin paremmin tuettu ohjelmointiympäristö, jossa lähdekoodin värjäys käytettyjen käskysanojen mukaan tai automaattinen sisennys on tuettu. Vaikka Ruby ei vaadikkaan esimerkiksi sisennyksien oikeaoppista käyttämistä – uusi alilohko alkaa aina sisemmältä tasolta – on se silti hyödyllinen lisä ohjelmoitaessa, semminkin kun kieli ei myöskään vaadi sulkeiden käyttöä vaan määrittelee lähdekoodin lohkoihin jaon syntaksisanojen avulla. Esimerkiksi koodi

```
01  if luku1 == 1
02    puts "Seuraava!"
03    if luku2 == 1
04      puts "Seuraava!"
05      if luku3 == 1
06        puts "Seuraava!"
07        if luku4 == 1
08          puts "Seuraava!"
09        end
10      end
11    end
12  end
```

on varmastikin miellyttävämpi muokata, ja ylipäänsä helpompitajuinen seurattava kuin

```
01  if luku1 == 1; puts "Seuraava!"
02  if luku2 == 1;
03  puts "Seuraava!"
04  if luku3 == 1; puts "Seuraava!"
05  if luku4 == 1; puts "Seuraava!"
06  end
07  end end
08  end
```

joka myös on kääntyvä ja täysin samalla tavalla toimiva Ruby-lähdekoodi. Ruby-kielen lähdekoodi voi rakenteeltaan olla niin vapaamuotoista, että se itse asiassa voi laajemmissa lähdekoodeissa muodostaa itsessään ongelman. Tämän vuoksi lähdekoodieditorina kannattaa käyttää jotain sellaista työkalua, joka automaattisesti hoitaa rakenteen muotoilun ja tunnistaa sieltä kokonaisuuksia. Lisäksi suurempia ohjelmointiprojekteja varten työkalujen olisi hyvä tukea usean tiedoston yhtäaikaista muokkausta sekä

yksinkertaisten rakenteellisten virheiden tunnistamista. Windows-työkaluista esimerkiksi Notepad++-, Netbeans- tai Eclipse-editori ovat varsin riittäviä. Yksittäisiin kokeiluihin tai perusrakenteiden kokeilemiseen myös joidenkin asennuspakettien mukana tuleva SciTE on varsin riittävä, ja sen käyttö tämän kirjan yhteydessä onkin suositeltavaa.

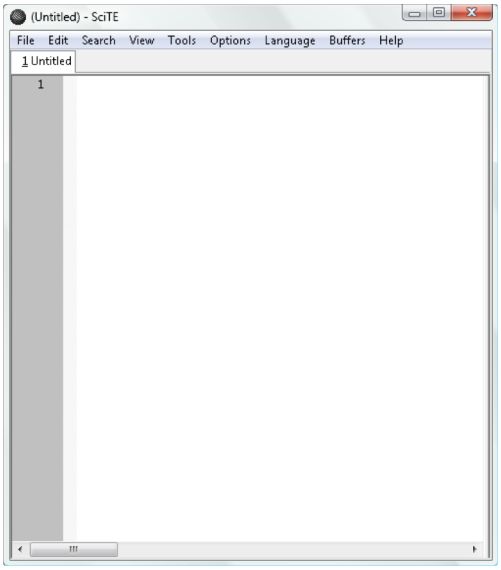
Lyhyt aloitusohje

Ruby-ohjelmointikielen harjoittelun aloittaminen on helpointa tehdä käyttämällä itse ohjelmointiympäristön paketista valmiiksi saatavia työkaluja. Vaikka Rubyn tavallisin käyttökohde onkin osana suurempaa ohjelmointiprojektia, tai käyttäen ohjelmistokehyksiä työskentelyn nopeuttamiseksi ja helpottamiseksi, on asennuspaketin mukana toimitettavat työkalut varsin riittävät perusteiden opiskeluun. Tietenkään nämä työkalut eivät ole kovin hyödyllisiä suuremman projektin toteuttamiseen, mutta kirjan ensimmäisen osan tehtäviin ne soveltuvat erinomaisesti.

Ensimmäinen työvaihe on itse asennuspaketin noutaminen. Uusin asennuspaketti löytyy verkosta Ruby-ohjelmointikielen kotisivuilta osoitteesta *www.ruby-lang.org*. Itse asennuspaketit löytyvät yläpalkin linkin *Downloads (www.ruby-lang.org/en/downloads/)* kautta. Selaimeen aukeavasta listata on mahdollista valita itse käännettäviä lähdekoodipaketteja kohdasta **Ruby Source Code**, sekä useille eri käyttöjärjestelmille tarkoitettuja valmiita asennuspaketteja. Windows-käyttöjärjestelmälle asennus on helpointa tehdä kohdasta **Ruby on Windows** löytyvästä paketista **One-click installer**, joka nimensä mukaisesti sisältää minimaalisen määrän työvaiheita tai käsin tehtäviä asetuksia. Tämä paketti on kooltaan keskimäärin 25 megatavua ja sisältää kaikki perustyökalut ja kirjastot, joita kirjan ensimmäisen osan tehtävissä tarvitaan.

Asennuspaketin asentamiseen, tarkastamiseen ja päivittämiseen on lisäohjeita Liitteessä 1. Kirjan luvussa 2 aloitetaan ohjelmointi esimerkkitehtävien avulla, joten tässä vaiheessa on suositeltavaa asentaa työkalut ja tarkastaa, että kaikki toimii oikein.

Kun paketti on ladattu ja asennettu liitteen ohjeiden mukaisesti, pitäisi tietokoneen Käynnistä-valikosta löytyä uusi kansio nimeltään **Ruby-XXX-YY**. jossa XXX ja YY kertovat asennetun Ruby-tulkin tarkan versionumeron. Tästä kansiosta löytyy kolme pikakuvaketta; **SciTE**, joka on jatkossa käytettävä lähdekoodieditori, **fxri**, joka on Ruby-kielen toiminnoista kertova interaktiivinen ohjekirja, sekä **irb**, joka avaa Ruby-tulkin komentokehotteeseen. Lisäksi kansiossa on Ruby-tulkin poistamista varten erillinen kuvake. Valitaan pikakuvakkeista **SciTE**, jonka jälkeen seuraavanlaisen ikkunan tulisi ilmestyä näytölle:



SciTE-editorin ikkuna

Tässä ikkunassa voidaan siis muokata Ruby-lähdekoodia, sekä suorittaa valmiiksi kirjoitettuja Ruby-ohjelmia. Kuitenkin, ennen kuin suoranainen ohjelmointi aloitetaan, käydään läpi jotain perusasioita Ruby-kielen tekniikasta, rakenteesta ja peruselementeistä.

LUKU 2

Oliot ja Ruby

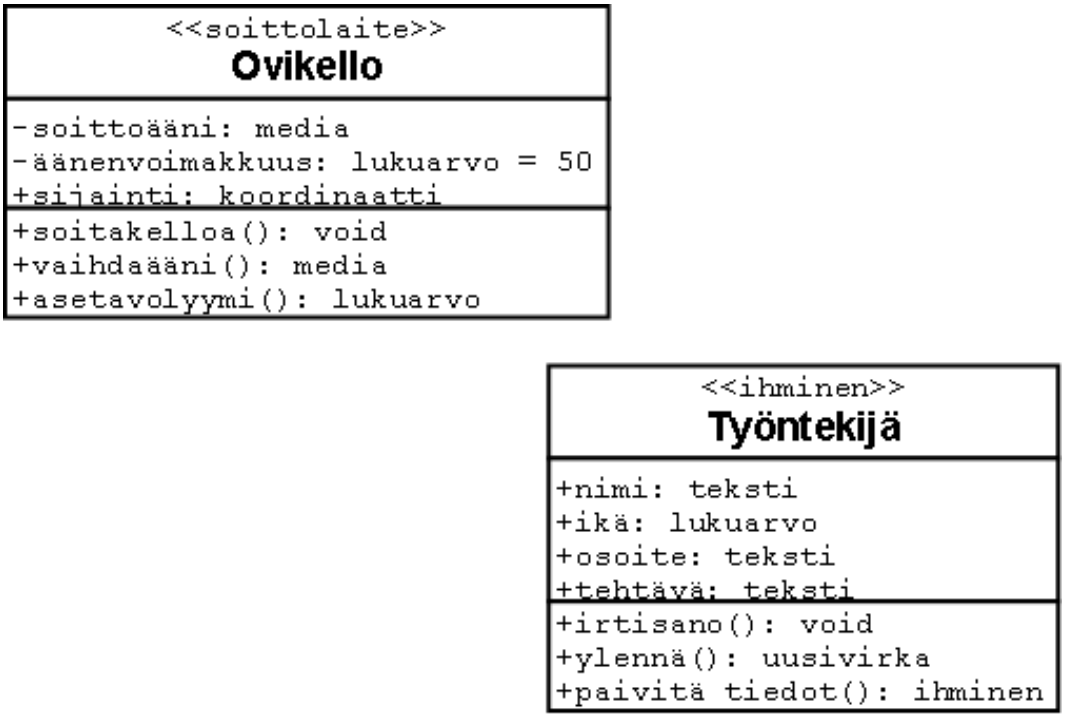
Tässä luvussa puhutaan ohjelmoinnissa tavallisesta konseptista, olioista. Luvussa tarkastellaan myös sitä, mikä olio on, mitä ne sisältävät ja miten olio määritellään. Lisäksi tarkatellaan olioita Ruby-ohjelmointikielessä sekä luvun lopuksi tehdään ensimmäiset ohjelmointikokeilut Ruby-ohjelmointikielellä.

MITÄ OLIOT OVAT?

Kuten ensimmäisessä luvussa mainittiin, eroaa Ruby-ohjelmointi monista muista ohjelmointikielistä sillä, että kielessä kaikki määritellään olioina. Tässä luvussa tarkastellaankin sitä, mitä nämä oliot tarkalleen ottaen ovat, mitä ne tuovat ohjelmointikieleen ja kuinka oliomaisuus näkyy esimerkiksi Rubyn käyttämisessä. Luvussa saatetaan käsitellä asioita, joiden merkitsevyys tai tarkoitus voi jäädä hieman epäselväksi; tässä vaiheessa kuitenkin riittää, että olioiden perusajatus ja rakenteiden perustoiminnot tulevat selväksi. Luvussa käsitellyistä asioista puhutaan enemmän jatkossa siirryttäessä ohjelmointitehtäviin, ja käytäessä läpi Ruby-ohjelmointikielen monimutkaisempia rakenteita.

Olioiden rakenne

Yleisellä tasolla olioita voidaan sanoa olevan minkä tahansa asian tai esineen; olio on ”mitä tahansa”, millä on määriteltäviä toimintoja tai ominaisuuksia. Esimerkiksi jos ajattelemme ihmistä oliona, meillä voi olla ominaisuuksina vaikkapa nimi, pituus, ikä, osoite tai paino. Toimintoina vastaavasti voidaan pitää esimerkiksi sellaisia asioita kuten kävele, puhu, syö tai työskentele. Vastaavasti esimerkiksi ovikellolle ominaisuuksia voisi olla soittoääni, äänenvoimakkuus ja sijainti, toimintoina soita merkkiääni tai aseta äänenvoimakkuus. Oikean maailman mukaan ajatteleminen onkin ollut oliomallisen ohjelmoinnin perusajatuksena.



Olioita; olioilla on arvoja ja toimintoja.

Toinen olio-ajatteluun vaikuttava toiiminto on lisäksi se, että toiminnot voidaan muodostaa täydentämällä tai luomalla uusia luokkia, joista voidaan tehdä uusia, paremmin toimintaan sopivia olioita. Tällöin puhutaan luokkien perinnästä, joka käytännössä tarkoittaa sitä, että uuden luokan pohjaksi otetaan jokin jo aiemmin määritelty luokka, jonka toiminnallisuudet ja ominaisuudet siirretään uuteen luokkaan. Tällöin uudelle luokalle riittää se, että siihen määritellään ne muutokset, jotka ovat toiminnan kannalta tarpeelliset. Tällätavoin voidaan vähentää uudelleenkirjoitettavan koodin määrää ja huolehtia siitä, että oliot toimivat tietyllä tavalla tai sisältävät tietyt ominaisuudet. Reaalimaailman esimerkkinä voidään ajatella vaikkapa autoja; pakettiauto, urheiluauto ja perhefarmari sisältävät kaikki samantyyppisiä ominaisuuksia (neljä pyörää, moottori jne.) mutta niiden yksityiskohdat vaihtelevat käyttötarkoituksen mukaisesti.

Olio-ohjelmoinnista yleisesti

Mitä tämä sitten tarkoittaa ohjelmointiympäristössä? Ensinnäkin, olio-ohjelmoinnissa (*object-oriented programming*) käytettävillä olioilla on edelleen ominaisuuksia ja toimintoja, joskin niistä puhutaan nimillä jäsenmuuttujat ja jäsenfunktiot, eli metodit. Käytännössä tämä tarkoittaa sitä, että ohjelmissa käytettävillä olioilla on niiden omistamia muuttujia, sekä funktioita jotka käyttävät niiden omia muuttujia.

Periaatteessa olio-pohjaisen ohjelmoinnin ja funktioiden sekä muuttujien toiminnan ero tulee ilmi siitä, kuinka kieli on rakennettu. Esimerkiksi Rubyssä luokka – eli kuvaus josta olio muodostetaan – määritellään seuraavalla tavalla:

```
class Luokannimi
  <jäsenmuuttuja> = <alkuarvo>
  <jäsenmuuttuja> = <alkuarvo>

  def <metodin nimi> (argumentit)
    <metodin koodi>
    <metodin koodi>
    ...
  def <seuraava metodi>
    ...
```

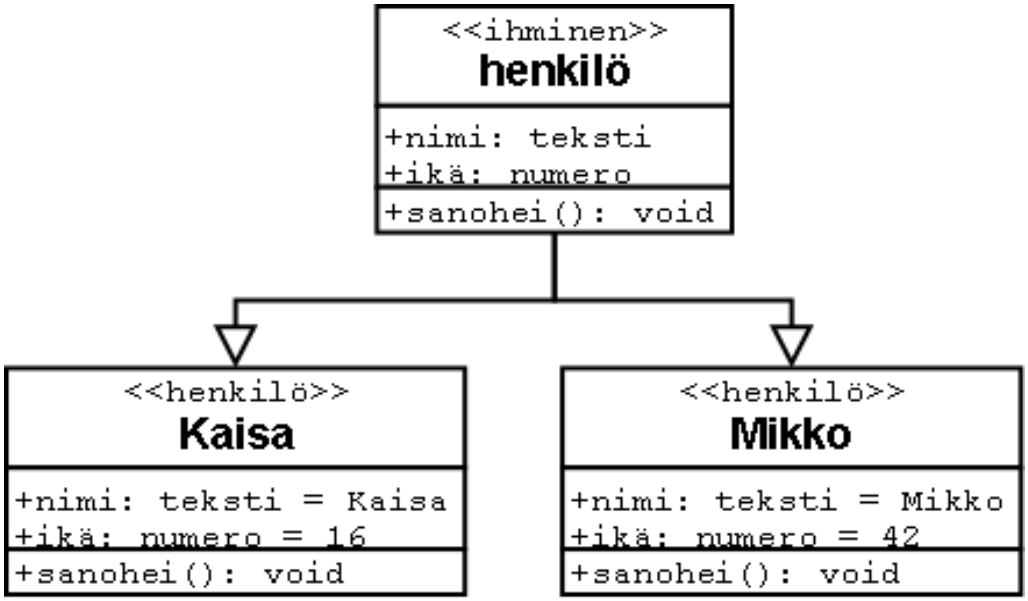
Tarkoittaa se sitä, että kaikki toiminnot ovat osa jotain oliota. Tämä on periaatteellinen ero verrattuna funktioihin ja muuttujiin, jossa jokainen muuttuja, ja erityisesti funktio, oli eräässä mielessä itsenäinen kokonaisuus. Jos luokasta määritellään olio nimeltä `Elukka`, voidaan se tehdä komennolla

```
Elukka = Luokannimi.new (argumentit)
```

jonka jälkeen luokan metodeja voitaisi kutsua komennoilla

```
Elukka.<metodinnimi>(argumentit)
```

Olio siis muodostaa yhden kokonaisuuden, joka sisältää erilaisia muuttujia sekä toimintoja. Tähän malliin tulee luonnollisesti joitain poikkeuksia, kuten jäsenmuuttujien näkyvyydet olion ulkopuolelle, luokkien periminen, metodin korvaaminen tai irralliset metodit, mutta perusperiaate olio-ohjelmoinnissa on aina sama: Olio on luokasta muodostettu kokonaisuus, joka sisältää omat muuttujat ja omat metodit.



Luokka ja oliot; luokista tehdään uusia olioita, joilla on omat muuttujat ja arvot. Kuvassa henkilö on luokka, josta on luotu oliot Kaisa ja Mikko

Ruby-ohjelmointikielessä kokonaisluku on olio, jossa on olion tarjoamina toimintoina mm. mahdollisuus testata, onko luku nolla, tai vaihtoehtoisesti palauttaa itseisarvo. Tämä onkin merkittävä ero useimpiin muihin ohjelmointikieliin verrattuna; periaatteessa toimintojen ero on syntaksinen, eli ero oliorakenteisuuden ja käskyrakenteisen kielen välillä on tavassa, miten kieltä kirjoitetaan. Otetaan esimerkiksi pseudokoodi-esitys - eli koodia joka ei suoranaisesti ole mitään ohjelmointikieltä, mutta samantyylistä ja helposti käännettävissä oikeaksi koodiksi - ohjelmasta, joka testaa onko muuttujan `luku` sisältämä arvo parillinen:

```
luku = 2
tulos = parillinen(luku)
if tulos == true:
    print(”Luku on parillinen!”)
```

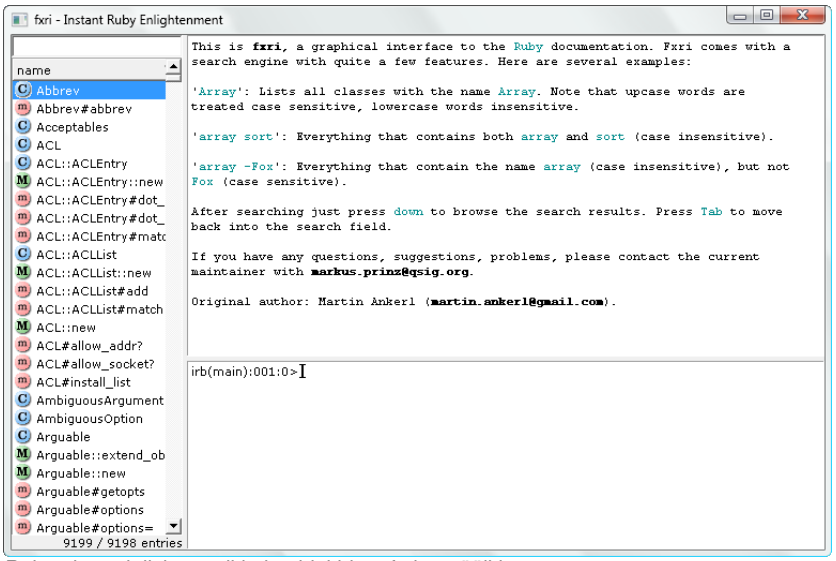
Koodi toimii kutakuinkin siten, että `muuttuja` luku lähetetään funktiolle `parillinen()` testattavaksi, ja se palauttaa paluuarvona tiedon true, eli tosi, jos annettu arvo on parillinen. Olioita käytettäessä vastaava koodi olisi kutakuinkin

```
luku = 2
tulos = luku.parillinen?
if tulos == true:
   print(”Luku on parillinen!”)
```

Eli periaatteessa ainoa merkittävä ero koodissa on, että tällä kertaa ohjelmassa ei kutsuttu jotain irtonaista funktiota, vaan olion `luku` metodia nimeltä `.parillinen?`, joka palautti vastaavan vastauksen kuin aiempi proseduraalisen ohjelmoinnin funktio `parillinen()`. Toisessa ohjelmassa funktion koodi on määritelty erilliseksi funktioksi, joka on kirjoitettu johonkin lähdekooditiedostoon erikseen, ja toisessa metodi on kirjoitettu osaksi olion määritelmää. Käytännössä esimerkki osoittaa sen, että olio-rakenteinen ohjelmointi ja ”perinteinen” ohjelmointi eivät välttämättä aina eroa paljoa toisistaan. Tärkein ero kuitenkin on se, että erilaiset toiminnot eivät enää ole irtonaisina, toisistaan erillään olevia funktioina – eli käskysarjoina jotka suoritetaan funktiokutsun jälkeen – vaan osa suurempia kokonaisuuksia, joita sanotaan olioiksi. Tähänkin tietysti on jotain poikkeuksia irtometodien puolella, mutta käytännössä helpoin tapa ymmärtää olio-ohjelmoinnin toimintaa on ehkä aloittaa kielen yleisistä rakenteista ja siirtyä kohti olioiden käyttämistä ohjelmointitehtävissä.

OLIOT RUBY-OHJELMOINTIKIELESSÄ

Seuraavaksi voidaan kuitenkin ottaa muutama oikea esimerkki siitä, miten olioiden läsnäolo näkyy Ruby-ohjelmointikielessä. Avataan *fxri*, interaktiivinen Ruby-ohjekirja, jossa on mukana tulkin ikkuna, ja tarkastellaan miten oliot näkyvät Rubyn toiminnassa. *fxri*:n autessa ruudulla pitäisi olla kolmeen osaan jaettu ikkuna, joka on kutakuinkin tämän näköinen:



Rubyn interaktiivisen tulkin ja ohjekirjan, fxri:n, pääikkuna

Ikkunan alareunassa on ikkuna, jossa on teksti `irb(main) 001:0>`. Tämä on itse asiassa Ruby-tulkin komentoikkuna, johon voidaan syöttää kaikenlaisia komentoja ja kokeilla erilaisia rakenteita. Kirjoitetaan tulkkiin esimerkiksi tulostuskäsky `puts "MOI"`. Ruutuun tulostuu seuraava teksti:

```
irb(main):001:0> puts "MOI"
MOI
=> nil
irb(main):002:0>
```

`puts` on Rubyn tulostuskäsky ja se tulosti sille argumenttina annetun tekstin MOI. Jatkossa suoraan tähän tulkin ikkunaan kirjoitettavat käskyt ja esimerkkikomennot ilmaistaan rivin aloittavalla >-merkillä, esimerkiksi näin:

```
> arvo = 1
=> 1
> arvo == 10
=> false
```

Kahdella rivillä annetaan muuttujaa arvo koskevia tietoja, ja tulkki vastaa niihin riveillä, jotka alkavat merkeillä =>. Kaikki käskyt kirjoitetaan siinä järjestyksessä, missä ne esimerkeissä esiintyy. Mutta palataan tässä vaiheessa takaisin olio-aiheisiin; kuinka oliot siis näkyvät Ruby-ohjelmointikielen rakenteessa?

Kaikista helpoin tapa osoittaa olioiden olemassaolo, ja se tosiseikka että kaikki asiat Rubyssa ovat jonkinlaisia olioita, on käytettäessä kokonaislukuja. Toisin kuin monissa muissa ohjelmointikielissä, on mikä tahansa arvo itsessään olio, ja se sisältää esimerkiksi jäsenmetodin `class`, joka kertoo minkätyyppistä tietoa kyseinen olio sisältää:

```
> 5 + 6
=> 11
> 5.class
=> Fixnum
> "Sanoja".class
=> String
```

Kuten monissa muissakin interaktiivisissa kielissä, voi tulkin ikkunassa käyttää numeroita vaikkapa laskemiseen. Rubyssa kuitenkin numerolla 5 on metodi, joka kertoo että olion arvo on muotoa Fixnum, eli luku. Vastaavasti tulkille annettu merkkijono - teksti ”Sanoja” - sisältää saman metodin, joka kertoo kyseessä olevan tekstiä huolimatta siitä, että merkkijonoa ei ole edes tallennettu muuttujan arvoksi.
Olioiden rakenteista puhuttaessa mainittiin myös jäsenfunktiot eli metodit, jotka suorittavat erilaisia toimintoja riippuen siitä mitä olioden on tarkoitus tehdä. Paras esimerkki tällaisesta metodista on kaikkiin muuttuja-olioihin kytketty metodi `empty?`, joka kysyy onko olion sisältämä arvo tyhjä vai ei.

```
> jotain = "Sihijuomaa."
=> "Sihijuomaa."
> jotain.empty?
=> false
>  jotain = ""
=> ""
> jotain.empty?
=> true
```

Esimerkkikoodissa luodaan muuttuja-olio jotain, jolle annetaan arvo ”Sihijuomaa.”. Toisessa käskyssä oliota kysytään onko se tyhjä, johon olio vastaa, ettei ole (`false`). Kolmannella rivillä olio vastaavasti tyhjennetään, jonka jälkeen vastaava kysymys tuotta vastauksen kyllä tai tosi (`true`). Myöhemmin kirjassa opetellaan lisäksi, kuinka olemassa olevista olioista voidaan periä uusia, tarkemmin määriteltyjä tai mielekkäämpiä olioita, sekä samalla lisätä olemassa oleviin luokkiin uusia metodeja tai jäsenmuuttujia. Tätä ennen kirjassa kuitenkin käsitellään Ruby-ohjelmointikielen perusasiat, joiden kautta olio-rakenteiden hahmottaminen on helpompaa.

ENSIMMÄINEN RUBY-OHJELMA

Luvun viimeisenä uutena asiana tehdään ensimmäisen omaan tiedostoonsa tallennettu ja toistuvasti ajettavissaoleva Ruby-ohjelma. Tässä vaiheessa käynnistetään varsinainen lähdekoodieditori SciTE pikakuvakkeesta *SciTE*. SciTE on lähdekoodieditori, jonka ominaisuuksiin kuuluvat mm. lähdekoodirakenteiden tunnistaminen ja värjäys, useamman lähdekooditiedoston yhtäaikainen editointi, sekä mahdollisuus kutsua Ruby-tulkkia suoraan editoriohjelmasta suoritamaan auki oleva ohjelma. Ohjelmassa on toki jotain rajoitteita; esimerkiksi syötteiden luku käyttäjältä toimii heikosti, mutta esimerkkitehtäviin editori riittää mainiosti.

Päällisin puolin ohjelma toimii kuten muutkin yleiset editoriohjelmat; valikon kohdasta File löytyy kaikki normaalit asetukset tiedostojenkäsittelyyn, *New, Save, Save as…* tai *Exit*, valikosta *Edit* valinnat kopiointia, leikkausta ja liittämistä varten. Lisäksi valikosta *Search* voi etsi lähdekoodirivejä haetuilla asetuksilla, tai valikosta *View* määritellä, mitä työkalurivejä ja ikkunoita ruudulla näytetään.

Koska SciTE tukee useita erilaisia ohjelmointikieliä – täydellinen lista kielistä on valikossa *Language*, mistä käytetyn kielen voi myös määritellä käsin – kannattaa ohjelmointi aloittaa tallentamalla käytettävä tiedosto. Kun ohjelmassa tallennetaan tiedosto, jonka tiedostopääte on `.rb`, ymmärtää SciTE että käytettävä tiedosto on Ruby-lähdekoodia, ja säätää aputoiminnot sen mukaisiksi. Tallennetaan tiedosto vaikkapa nimellä *moimaailma.rb*, hakemistoon *C:\koodit*. Vaikka tiedostojennimissä voi periaatteessa olla erikoisempiakin merkkejä, kuten skandinaavisia kirjaimia tai välilyöntejä, kannattaa Rubylla ohjelmoidessa pidättäytyä yksinkertaisissa nimissä, joissa hakemistojen tai tiedostojen nimissä ei ole kuin englanninkielisen aakkoston kirjaimia (A-Z, a-z). Lisäksi erikoismerkkejä (#, ¤, !) ja esimerkiksi välilyöntejä kannattaa välttää. Vaikka ne voivat toimia joissain ohjelmissa, ei asia välttämättä ole aina näin, joten ongelmien välttämiseksi nimeämistapa kannattaa pitää yksinkertaisena.

Kun tiedosto on luotu, voidaan siihen kirjoittaa lähdekoodia. Kirjoitetaan koodiin seuraavat rivit:

```
#Eka ohjelma!
puts "Moi maailma!"
puts "Tästä se sitten lähtee."
```

Kuten ohjelmointikielissä yleisesti, täytyy tässäkin tapauksessa koodi kirjoittaa merkilleen oikein. Erityisesti kannattaa huomata ensimmäinen rivi, jonka nyt pitäisi olla värjäytynyt vihreäksi; kyseessä on huomio siitä, että ohjelman rivi on kommenttien sisällä. Seuraavaksi suoritetaan ohjelma. Se tapahtuu valitsemassa valikosta *Tools* valinta *Go*, tai käyttämällä pikanäppäintä *F5*.

Esimerkki 2-1: Ensimmäinen ohjelma

Lähdekoodi

```
01  #Eka ohjelma!
02  puts "Moi maailma!"
03  puts "Tästä se sitten lähtee."
```

Tuloste

```
>ruby moimaailma.rb
```

```
Moi maailma!
Tästä se sitten lähtee.
>Exit code: 0
```

Tuloste-ikkuna aukeaa automaattisesti editorin ikkunan oikeaan reunaan. Tulosteessa olevat merkinnät ensimmäisellä ja viimeisellä rivillä ovat SciTEn automaattisesti luomia. Ne kertovat, että Ruby-tulkki suoritti juuri ohjelmakoodin tiedostosta *moimaailma.rb*, ja että ohjelman suoritus loppui luonnollisesti ilman ongelmia, eikä esimerkiksi virhetilanteeseen. Esimerkin esitystavasta kannattaa lisäksi huomata, että pidemmissä kooditeksteissä ja esimerkkikoodeissa käytetään kirjassa rivien numerointia auttamaan ohjelman toiminnan hahmottamisessa sekä helpottamassa lähdekoodiin viittaamista tekstissä. Rivinumeroita ei tietenkään kirjoiteta lähdekoodieditoriin. Jos tulkki antaa tulokseksi jotain tämän suuntaista

```
>ruby moimaailma.rb
moimaailma.rb:2: undefined method `pust' for main:Object (NoMethodError)
Moi maailma!
>Exit code: 1
```

eli jonkin Error-viestin (tässä tapauksessa `NoMethodError`) sekä viimeisellä rivillä tiedon `Exit code: 1` tarkoittaa se sitä, että koodissa on jokin virhe. Tässä esimerkissä todennäköisin syy voisi olla puuttuva kommenttimerkki ensinmmäiseltä riviltä, väärin kirjoitettu käskysana tai rivin lopusta unohtunut sitaattimerkki.

Joka tapauksessa, jos ohjelma toimi oikein ja SciTE antoi tulosteen ilman ongelmia, on kaikki valmiina kirjan ensimmäisen osan ohjelmointiaiheita varten. Seuraavissa kuudessa luvussa käsitelläänkin Rubyohjelmointia tarkemmin, ja tutkitaan miten kieli toimii ja mitä rakenteita siitä löytyy.

LUKU 3

Ruby-kielen perusrakenteet ja tietomuodot

Rubyn peruskielioppi
Tulostuskäskyt ja muotoilumerkit
Tiedon pyytäminen käyttäjältä
Muita perusrakenteita

Tässä luvussa käsitellään Ruby-kielen perusasioita, kommentointia, koodin rakennetta eli syntaksia, muuttujien käyttämistä sekä perustoimintoja kuten tulostamista tai syötteen pyytämistä käyttäjältä. Luvun loppuosassa käsitellään Ruby-kielen erikoisuuksia ja hyödyllisiä perusominaisuuksia, joista on hyötyä kielellä ohjelmia rakentaessa.

RUBYN PERUSKIELIOPPI

Kuten edellisessä luvussa puhuttiin, on Ruby rakenteeltaan toteutettu täysin olioiden avulla. Tämä ei kuitenkaan tarkoita sitä, että Rubylla ohjelmointi poikkeaisi paljoakaan muista kielistä; Niin kuin muissakin kielissä, on Rubyssa normaaliin tapaan muuttujat, valinta- ja ehtorakenteet, mahdollisuus virheidenkäsittelyyn ja tiedostojen kanssa työskentelyyn. Tässä luvussa puhutaankin Ruby-kielen perusasioista, jotka kielestä olisi hyvä tietää ennen monimutkaisempien asioiden opettelua. Luvussa käsitellään lyhyesti muuttujien käyttäminen, erilaiset muuttujatyypit, rivitys- ja kirjoitussäännöt sekä esimerkiksi satunnaisluvun ottaminen. Ensimmäisenä näistä aiheista käsitelläänkin varatut sanat, sekä Ruby-kielen muuttujia koskevat nimeämissäännöt.

Varatut sanat ja muuttujien nimeämissäännöt

Muuttujien, funktioiden ja olioiden nimen voi Ruby-kielessä määritellä vapaasti, kuitenkin seuraavia sääntöjä noudattaen. Ensinnäkin, nimeksi ei käy yksikään seuraavista sanoista, koska kyseiset sanat on varattu syntaksille:

Taulukko 3.1 Ruby-kielen varatut sanat

__FILE__	def	in	self
__LINE__	defined?	module	super
BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	yield

Lisäksi nimen pitää alkaa jollain englanninkielen isolla tai pienellä kirjaimella (a-Z, A-Z) tai alaviivalla (_). Ei-aloittavina merkkeinä voi käyttää lisäksi numeroita (0-9). Muut merkit (¤, £, ”, =) eivät oletusarvoisesti kelpaa, joskin muutamat merkkikombinaatiot ($nimi, @nimi, @@nimi) toimivat teknisesti, mutta samalla määrittelevät lisätoimintoja, kuten tekevät muuttujasta globaalin tai määrittelevät ne luokan muuttujiksi. Lisäksi skandinaavisten kirjainten, eli ns. ääkkösten (ä,ö,å,æ,ø) käyttäminen aiheuttaa virheilmoituksen. Yleisesti osittain syntaksin pakottama ja hyväksytyin, ja samalla myös tässä kirjassa käytetty, nimien erottelumekanismi meneekin Rubyssa seuraavalla tavalla:

Taulukko 3.2 Ruby-kielen nimeämiskäytännöt

Esimerkki	Tyyppi	Määritelmä
alkaa_pienella	muuttuja	Normaali muuttujan nimi tulisi alkaa pienellä kirjaimella erotuksena metodeista
$nimi	globaali muuttuja	muuttuja, joka näkyy kaikkialle
@nimi	olion muuttuja	muuttuja, joka näkyy ainoastaan olion sisällä
@@nimi	luokan muuttuja	muuttuja, joka kuuluu luokalle eikä periydy olioihin
IsoAlkukirjain	luokan tai metodin nimi	luokille ja luokkien jäsenmuuttujille sekä metodeille annetaan tämänmuotoinen nimi
huutomerkki_lopussa!	metodi	käytettävä metodi muokkaa tai saattaa muokata olion sisältämiä tietoja
kysymys_lopussa?	metodi	palauttaa joko arvon true tai false

Näiden nimeämismenetelmiä käytetään nimenomaan helpottamaan ohjelmoijan työtä kertomalla jotain perustietoja muuttujan tai metodin toiminnasta. Esimerkiksi huutomerkkiin päättyvä metodi on arvokasta tietoa, koska tämä kertoo ohjelmoijalle, että olioon tallennettu tieto saattaa muuttua kutsun seurauksena, tai että kysymysmerkkiä käyttävä metodi palauttaa ainoastaan totuusarvoja tai jonkin muun hyvin yksiselitteisen vastauksen.

Muuttujat

Kuinka muuttujia sitten Ruby-ohjelmoinnissa käytetään? Koska kieli on hyvin dynaaminen ja sisältää paljon tulkin puolesta tapahtuvaa automatiikkaa, ei muuttujien kanssa työskentelyyn tarvita mopniakaan muistisääntöjä. Käytännössä muuttuja luodaan antamalla jokin nimi ja määrittelemällä sille jokin arvo, kuten esimerkiksi `munmuuttuja = 313`. Tämän jälkeen muuttuja nimeltä `munmuuttuja` on luotu, ja se sisältää kokonaislukuarvon 313.

Toisin kuin esimerkiksi C-kielessä, ei muuttujien tyypeistä – eli millaisia arvoja muuttuja sisältää – tarvitse välittää. Kokonaisluvulla määriteltyyn muuttujaan voi myöhemmin kopioida tilalle merkkijonon tai vaikkapa taulukon ilman mitään rajoituksia. Samaten muuttujan luomisen voi tehdä missä tahansa osassa ohjelmakoodia, joskin käytännöllisintä on luoda muuttujat aina ohjelman alussa keskitetysti. Ainoa rajoitus onkin siinä, että Ruby ei automaattisesti luo muuttujia; jos esimerkiksi valintarakenteessa on käytetty tuntematonta muuttujaa osana valintaehtoa, lopettaa ohjelma toimintansa virheilmoitukseen, eikä esimerkiksi oleta kyseessä olevan uusi, tyhjäksi määritelty muuttuja. Seuraavassa esimerkissä tarkastellaankin muuttujien tekemistä ja käyttämistä:

Esimerkki 3-1: Muuttujan määrittely

Lähdekoodi

```
01  #Tehdään joukko muuttujia ja kokeillaan niitä
02
03  luku1 = 10
04  luku2 = 40
05  sana = ”Testataan muuttujia!”
06
07  puts sana
08
09  #käytetään muuttujille annettuja arvoja
10  luku3 = luku1 + luku2
11  puts luku3
```

Tuloste

```
>ruby 3-2muuttujat.rb
Testataan muuttujia!
50
>Exit code: 0
```

Ohjelmassa luodaan kolme muuttujaa `luku1`, `luku2` ja `sana`. Muuttuja luodaan yksinkertaisesti antamalla ohjelmassa nimi, ja määrittelemällä sille jokin arvo. Myöhemmin puhumme lisää muuttujien näkyvyydestä ja nimiavaruudesta, mutta tässä vaiheessa riittää kun muuttuja vain on luotu. Rivillä 7 tulostetaan muuttujan `sana` sisältämä arvo, joka siis on merkkijono ”`Testataan muuttujia!`”, ja rivillä 10 käytetään kahden muuttujan yhteenlaskua määrittelemään kolmannen muuttujan, `luku3`, arvo. Jälleen rivillä 11 on `puts`-käsky, eli Rubyn oletus-tulostus, joka vastaanottaa argumenttina arvon, ja tulostaa sen ruudulle. Esimerkissä lähinnä siis tarkastellaan sitä, kuinka muuttujia luodaan ja kuinka ne toimivat.

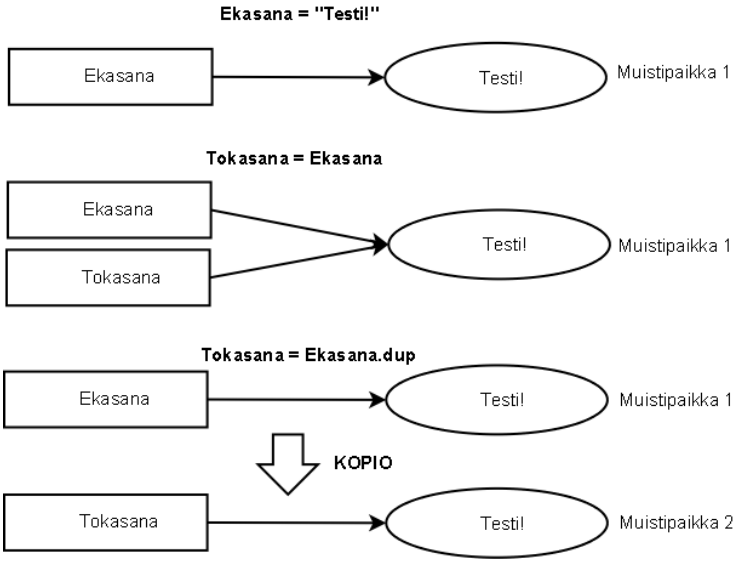
Muuttujista kannattaa huomata kuitenkin eräs yksityiskohta joka liittyy arvojen syöttämiseen. Rubyssa muuttujan antaminen arvon sijoituksena toimii hieman yllätyksellisellä tavalla. Luodaan muuttuja `eka`, ja annetaan sille arvo ”`Alussa!`”, sekä annetaan muuttujalle `toka` muuttuja `eka` arvona, tarkoituksena ottaa kopio muuttujan `eka` saamasta arvosta:

```
eka = ”Alussa!”
toka = eka
```

Tämän jälkeen muutoksen muuttujan `toka` arvoon vaikuttavat myös muuttujaan `eka`:

```
toka.chop!
puts eka
=> Alussa
```

Metodi `chop!` poistaa muuttujasta `toka` viimeisen merkin, mutta johtuen siitä, että ohjelmalle on kerrottu muuttujan `toka` olevan sama kuin muuttuja `eka`, se samalla muuttaa myös `eka`:n arvoa. Helpointa asia on varmaankin ymmärtää seuraavan kuvan avulla:



Muuttujien arvojen kopioituminen

Kuten kaavioista nähdään, on ongelma helpointa on välttää kopioimalla muuttujan arvoja muuttujista toiseen metodin `dup`-avulla. Tällöin Ruby ei määritä muuttujan arvoa kirjaimellisesti toiseksi muuttujaksi, vaan ainoastaan kopioi kopiointihetken arvon uuteen muuttujaan:

```
eka = ”Alussa!”
toka = eka.dup
toka.chop!
puts eka
=> Alussa!
```

Kuten esimerkistä nähdään, tällä kertaa muuttujan eka arvo ei ole kopioitunut:. muuttujaan toka tehdyt muutokset eivät vaikuta kuin muuttujaan itseensä. Lisäksi numeroarvoja kuten kokonaislukuja käsitteltäessä virhettä ei tapahdu.

Perustietomuodot

Huolimatta siitä, että Ruby on dynaamisesti muuttujien tyypityksen hoitava kieli, eli muuttujaa luotaessa muuttujan käyttämää tietotyyppiä ei tarvitse määritellä, voidaan siinä sanoa olevan erilaisia tietomuotoja. Vaikka dynaaminen tyypitys huolehtiikin siitä, että ohjelmoijan ei tarvitse muuttujaa käyttäessään määritellä tietomuotoa kuten *int (integer, kokonaisluku*) tai *str* (*string, merkkijono*) saatikka noudattaa määritelmää myöhemmin ohjelmassa, mahdollistaa jotkin tietomuodot erikoisempia toimintoja jotka ovat sidottuja muuttujan tyyppiin. Näitä toimintoja ja tapahtumia kutsutaan metodeiksi, ja niitä käytetään syntaksilla

```
<muuttujannimi>.<metodinnimi>
```

Kuten toisessa luvussa kerrottiin, toimii Rubyssa kaikki olioiden avulla. Tämän vuoksi kaikista muuttujista, joita ohjelmiin tehdään, löytyy metodi `class`, joka kertoo minkätyyppistä tietoa muuttuja sisältää:

```
>eka = 125
=> 125
>eka.class
=> Fixnum
>toka = ”Sihijuoma”
=> “Sihijuoma”
>toka.class
=> String
>kolmas = 4.24
=> 4.24
>kolmas.class
=> Float
```

Käytännössä `class`-metodia käyttäen on helppo tarkastaa esimerkiksi tiedostoa luettaessa muuttujaan tallennetun tiedon kelvollisuus; jos luettu rivi palauttaa muodon `String` ja sillä olisi tarkoitus tehdä laskutoimituksia, tarvitaan väliin tyyppimuunnos ja samalla mahdollisesti virhekäsittelytoiminnot.

Toinen kaikista muuttujaolioista löytyvä metodi on `to_s`, ja se mahdollistaa kaikkien muuttujatyyppien esittämisen jonkinlaisena merkkijonoja:

```
>luku = 1
=> 1
>luku.to_s
=> “1”
>puts ”Tekstiä ” + luku.to_s
Tekstiä 1
=> nil
```

Miksi tämä sitten on hyödyllinen metodi? Syy siihen paljastuu tulostuskäskyn kanssa yhteiskäytöstä. Jos tulostamme tekstiä siten, että kokoamme ensin koko testin yhteen merkkijonoon joka tulostetaan puts-käskyllä, saamme yrityksestä liittää merkkijonon ja luvun toisiinsa virheilmoituksen

```
puts ”Tekstiä ” + luku
TypeError: can’t convert Fixnum into String
            from (irb):7:in `+’
            from (irb):7
            from :0
```

joka voidaan ohittaa käyttämällä `to_s`-metodia. Tämä ongelma voidaan myös ohittaa käyttämällä Rubysta löytyvää `#{muuttujannimi}`-syntaksia tulostuksen toteuttamiseen:

```
>lukumaara = 1000
=> 1000
>puts ”Tästä asiasta on kirjotettu ainakin #{lukumaara} kirjaa!”
Tästä asiasta on kirjotettu ainakin 1000 kirjaa!
=> nil
```

Vaikka molemmat ratkaisut toimivat yhtä hyvin, on `#{muuttuja}` lopulta hieman näppärämpi ratkaisu koska se voidaan sijoittaa suoraan merkkijonon sekaan. Lisäksi perustietomuodoista kannattaa huomata se, että kokonaislukujen käyttäminen poikkeaa liukuluvuilla työskentelystä. Parhaiten tämä voidaan huomata jakamalla kaksi kokonaislukua keskenään:

```
> 10/3
=> 3
> (5+5) / 3
=> 3
```

Kokonaisluvuilla jakaminen jättää kokonaan pois desimaaliosan vastauksesta, antaen monessa tapauksessa virheellisen tai ainakin riittämättömän vastauksen. Tämä virhe korjaantuu, kun yksikin laskutoimituksen komponentti määritellään liukuluvuksi, eli jos ei muuta niin sille annetaan desimaaliosa `.0`. Tällöin ohjelma suorittaa koko laskutoimituksen käyttäen desimaaleja ja tuottaa tarkan vastauksen.

```
> 10/3.0
```

```
=> 3.33333333333333
> (5.0 + 5) / 3
=> 3.33333333333333
```

Merkkijonot

Perustietotyypeistä merkkijono on kuitenkin kaikista mielenkiintoisin, sillä se sisältää metodeinaan useita erilaisia aputoimintoja, joiden avulla merkkijonojen muokkaus on tehty helpoksi. Merkkijonojen metodeilla on esimerkiksi mahdollista muuttaa merkkijono isoiksi kirjaimiksi, alkamaan isoilla alkukirjaimilla tai poistaa välimerkkejä. Seuraavassa mallissa on joitain merkkijonojen metodeja:

```
>tekstia = ”lause jossa on paljon sanoja\n”
=> ”lause jossa on paljon sanoja\n”
>tekstia.reverse
=> “\najonas nojlap no assoj esual”
>tekstia.length
=> 29
>tekstia.sub(“s”,”z”)
=> ”lauze jossa on paljon sanoja\n”
>tekstia.upcase
=> ”LAUSE JOSSA ON PALJON SANOJA\n”
```

Ruby-kielen merkkijonoille on yhteensä käytössä yli 50 erilaista metodia, joskin varsinaisten toimintojen määrä ei ole näin korkea. Useista metodeista on olemassa kaksi versiota, kopion palauttava versio sekä alkuperäistä muuttujan arvoa muokkaava, joka on merkitty lisäämällä metodin nimen perään huutomerkki:

```
>sana = ”Tekstijono”
=> ”Tekstijono”
>sana.upcase
=> ”TEKSTIJONO”
>sana
=> ”Tekstijono”
>sana.upcase!
=> ”TEKSTIJONO”
>sana
=> ”TEKSTIJONO”
```

Kuten esimerkistä nähdään, muuttaa huutomerkillä nimetty metodi pysyvästi alkuperäistä muuttujan `sana` arvoa, kun normaaliniminen ainoastaan palauttaa siitä kopion. Toinen nimeämissääntö koskee kysmysmerkiin päättyviä metodeja, ne palauttavat ainoastaan joko arvon `true` tai `false`. Oheiseen taulukkoon on kerätty joukko hyödyllisimpiä metodeja. Mikäli nimen perässä on huutomerkki, tarkoittaa se sitä että metodista on myös olemassa pysyvän muutoksen tekevä variaatio.

Taulukko 3.3 Merkkijonojen tavallisimpia muokkausmetodeja

Metodin nimi, variaatiot	Selite	Esimerkki
`capitalize, !`	Palauttaa rivin kirjoitettuna Isolla alkukirjaimella, muiden kirjainten ollessa pieniä.	`> "HILJAA!".capitalize` `=> "Hiljaa!"`
`chomp, !`	Palauttaa rivin ilman lopussa olevaa rivinvaihtomerkkiä.	`> "Moikka\n".chomp` `=> "Moikka"`
`chop, !`	Palauttaa rivin ilman viimeistä merkkiä.	`> "Moikka".chop` `=> "Moikk"`
`count`	Laskee kuinka monta kertaa annetut merkit esiintyy merkkijonossa.	`> "ukulele".count("e")` `=> 2`
`delete, !`	Poistaa merkkijonosta argumenttinä annetut merkit.	`> "s.a.n.a.".delete(".")` `=> "sana"`

downcase, !	Palauttaa rivin kirjoitettuna pelkillä pienillä kirjaimilla.	> sana = ”HAHAA” > sana.downcase => ”hahaa”
dup	Luo muuttujan arvosta aidosti uuden kopion.	
empty?	Testaa, onko merkkijonon pituus 0.	> "".empty? => true
gsub, !	Vaihtaa kaikki ensimmäisen argumentin sisältävät arvon toisen argumentin arvoksi.	jono = "papupata" > jono.gsub("pa","ke") => "kepuketa"
include?, member?	Kertoo onko kysytty jäsen testattavassa oliossa tai olion lukualueella.	> jono = 1..10 > jono.include?(4.53) => true
length	Palauttaa merkkijonon pituuden lukuarvona.	> sana = "pituus" > sana.length => 6
strip,rstrip, lstrip, !	Palauttaa merkkijonosta kopion, josta on poistettu ylimääräiset sisennykset ja välilyönnit (rstrip pelkkä oikea, lstrip pelkkä vasen.)	> " teksti ".strip() => "teksti"
reverse, !	Palauttaa merkkijonon ympäri käännettynä.	> "isotupakka".reverse => "akkaputosi"
split	Jakaa merkkijonon taulukoksi annetun merkin kohdalta	> "Teksti jossa on sanoja".split(" ") => ["Teksti", "jossa", "on", "sanoja"]
upcase, !	Palauttaa merkkijonon pelkillä isoilla kirjaimilla kirjoitettuna.	> "tEKsTi".upcase => "TEKSTI"
<<	Lisää merkkijonon loppuun toisen merkkijonon	> ”merkki” << ”jono” => ”merkkijono”

Periaatteessa osa taulukon metodeista vaikuttaa hieman tarpeettomilta, mutta useimmille niistä löytyy yllättävääkin käyttöä: esimerkiksi metodi `chomp` voi ensialkuun vaikuttaa tarpeettomalta, mutta se itseasiassa nopeuttaa merkkijonojen lukemista tiedostoista merkittävästi. Koska `chomp` poistaa viimeisen merkin ainoastaan, mikäli kyseessä on järjestelmän rivinvaihtomerkki, voidaan sitä käyttää turvallisesti varmistamaan, ettei luettuun syötteeseen jää ylimääräisiä rivinvaihtoja, jotka myöhemmin sotkevat luetun tekstin tulostuksen. Vastaavasti esimerkiksi metodi `upcase`, joka palauttaa koko merkkijonon isoilla kirjaimilla esitettynä, helpottaa tekstin järjestelemistä aakkosjärjestykseen käsin, koska tietokoneet eivät ymmärrä isojen ja pienten kirjainten eroa, normaalisti aina sijoittaen kaikki pienet kirjaimet isojen kirjainten jälkeen.

Tyyppimuunnokset

Muuttujia tulostaessa törmätään helposti ongelmaan, joka johtuu siitä, ettei Ruby osaa aina automaattisesti muuttaa kokonaislukua merkkijonoksi, esimerkiksi tulostettavaa tekstiriviä parsittaessa tai laskutoimituksen desimaaliosaa tarvittaessa. Tämän vuoksi joskus joudutaan tilanteeseen, jossa on tarpeellista käsin määritellä mitä tyyppiä muuttujasta käytetään.

Tähän Rubyssa voidaan käyttää tyyppiluokkien (`Fixnum`, `String`, `Float`) tyyppimuunnosmetodeja, joista `to_s` esiteltiinkin jo aikaisemmin. to_s muuttaa siis kaikki muuttujatyypit merkkijonoiksi, jotka voidaan tulostaa puts-käskyllä. Lisäksi Rubyssa on olemassa `to_i`, joka muuttaa lukuarvot kokonaisluvuiksi poistamalla niistä desimaaliosan, ja `to_f` joka määrää luvun käsiteltäväksi liukulukuna:

```
> luku = 1234
=> 1234
> luku.to_s
```

```
=> "1234"
> luku.to_f
=> 1234.0
> luku = 1.9842
=> 1.9842
> luku.to_i
=> 1
```

Viimeisestä esimerkistä kannattaa huomata, että tyyppimuunnos ei pyöristä lukua oikein, vaan ainoastaan katkaisee siitä desimaaliosan! Pyöristys pitää tehdä `round.to_i`-metodia käyttäen:

```
> luku = 5.78
=> 5.78
> luku.round.to_i
=> 6
```

Ikävä kyllä `round.to_i` ei osaa pyöristää desimaalilukuja muuten kuin kokonaislukuihin, ja `round.to_f` ainoastaan palauttaa pyöristyksen muodossa `.0`. Jos desimaaliosa halutaan pyöristää, voidaan se toteuttaa esimerkiksi kertomalla ja jakamalla luku pyöristyksen yhteydessä:

```
> luku = 1.936
=> 1.936
> luku = luku * 100
=> 193.6
> luku =luku.round.to_i
=> 194
> luku = luku.to_f / 100
=> 1.94
```

Merkkijonojen kanssa tyyppimuunnosmetodeja harvemmin tarvitaan, mutta siltä varalta, että ohjelmaan olisi päässyt virhe, toimii tyyppimuunnosfunktiot kaikesta huolimatta. Lukumuunnokset palauttavatkin arvon 0, jos niille annetaan argumenttina merkkijono.

```
> sana = "merkki"
=> "merkki"
> sana.to_i
=> 0
> sana.to_f
=> 0.0
```

Eniten hyötyä tyyppimuunnosmetodeista on silloin, kun niiden avulla voidaan vaatia että jokin arvo käsitellään laskutoimituksessa desimaalilukuna. Tämän ansiosta voidaan esimerkiksi varmistaa, että ohjelma antaa varmasti riittävän tarkan vastauksen laskutehtävään:

```
> eka = 10
=> 10
> toka = 3
=> 3
> eka / toka
=> 3
> eka.to_f / toka
=> 3.33333333333333
```

Metodit `to_s` ja `to_i` sisältävät lisäksi erään hyvin mielenkiintoisen ja samalla näppärän ominaisuuden; niiden avulla voidaan lukuarvoista tehdä helposti binaari-, okta- tai heksalukuja antamalla muunnosparametriksi arvo 2, 8 tai 16, eli käytettävän numerojärjestelmän kantaluku:

```
> luku = 1234
=> 1234
> luku.to_s(2)
=> "10011010010"
> luku.to_s(8)
=> "2322"
```

```
> luku.to_s(16)
=> "4d2"
> "111".to_i(2)
=> 7
> "ff4d".to_i(16)
=> 65357
```

Tästä toiminnosta on hyötyä esimerkiksi cryptografiassa tai silloin, kun halutaan laskea heksa-arvoja esimerkiksi värien määrittelemiseen RGB-arvoina CSS-tyylikarttoihin.

Rakenteiset tietomuodot, taulukko

Perustietotyyppien lisäksi Rubyssa on mahdollisuus muodostaa muuttujia, jotka sisältävät useamman merkkijonon tai useita lukuarvoja. Tavallisin, ja samalla käytännöllisin tällainen tietomuoto on taulukko (*array*), johon voidaan tallentaa useita muuttujia samalla kertaa. Periaatteessa taulukkoa kannattaa ajatella eräänlaisena listana, johon voidaan laittaa muistiin useita muuttujan arvoja ja siirtää kerralla esimerkiksi antamalla taulukko aliohjelman tai metodin argumenttina.

Taulukko voidaan määritellä usealla erilaisella tavalla. Yksinkertaisin tapa on määrätä, että jokin muuttuja on taulukko antamalla käsky:

```
> taulu = Array.new()
=> []
```

Tämä ei kuitenkaan aina riitä, vaan esimerkiksi taulukon koko olisi hyvä saada määrättyä luonnin yhteydessä. Tämä onnistuukin argumenttiä käyttäen; taulukkoa luotaessa voidaan myös haluttaessa kertoa, kuinka monta alkiota taulukkoon annetaan:

```
> taulu = Array.new(5)
=> [nil, nil, nil, nil, nil]
```

Tässä tapauksessa Ruby luo uuden taulukon, jossa on viisi paikkaa, jotka on alustettu arvolla nil, eli ”tyhjä”, tai oikeammin ”ei mitään”. Jos taulukkoon halutaan määritellä joku valmis arvo kuten vaikkapa nolla, kerrotaan se taulukkoa luotaessa:

```
taulu = Array.new(5, "sisus")
=> ["sisus", "sisus", "sisus", "sisus", "sisus"]
```

Jos taulukolle voidaan määritellä suoraan aloitusarvot, voidaan se myös määritellä suoraan taulukkoa luotaessa:

```
taulu = Array["Ma","Ti","Ke","To","Pe","La","Su"]
=> ["Ma", "Ti", "Ke", "To", "Pe", "La", "Su"]
```

tai lyhyemmin

```
taulu = [1,2,3,4,5,6,7]
=> [1, 2, 3, 4, 5, 6, 7]
```

Miksi taulukot sitten ovat hyödyllisiä? Ensinnäkin, ne voivat tallentaa yhteen muuttujaan paljon asioita kerralla, jolloin tiedot pysyvät helposti yhdessä paikassa. Lisäksi taulukon käyttäminen ei paljoakaan vaikeuta itse tiedon käyttämistä: jos esimerkiksi taulukossa tulee tarve viitata taulukon kolmanteen alkioon, voidaan siihen viitata muodossa

```
<taulukonnimi> [<alkion järjestysnumero>]
```

Eli vaikkapa näin:

```
varit = ["sininen","punainen","virhea"]
=> ["sininen", "punainen", "virhea"]
> varit[1]
=> "punainen"
```

Kuten monessa muussakin ohjelmointikielessä, myös Rubyssa pätee sääntö että numerointi alkaa aina nollasta. Tämän vuoksi kannattaakin muistaa, että ensimmäinen alkion on aina paikalla 0 ja viimeinen paikalla pituus-1:

```
> varit[0]
=> "sininen"
varit.length
=> 3
> varit[2]
=> "virhea"
> varit[3]
=> nil
```

Lisäksi taulukon alkioihin voi viitata negatiivisilla luvuilla, jolloin viimeinen alkio on paikalla `-1`, toiseksi viimeinen paikalla `-2` ja niin edelleen. Koska takaperin laskettuna numeroiti alkaa `-1`:stä, on ensimmäinen alkio tällöin `-[pituus]`.

```
> varit[-1]
=> "vihrea"
> varit[-3]
=> "sininen"
```

Toinen hyödyllinen ominaisuus, joka taulukoita käytettäessä helpottaa työskentelyä, on se että taulukoilla, samoin kuin merkkijonoilla, on metodeja joiden avulla niitä voidaan hyödyntää tehokkaammin.

```
lukuja = [23,4,12,55,101,55,23,213,1]
=> [23, 4, 12, 55, 101, 55, 23, 213, 1]
> lukuja.empty?
=> false
> lukuja.include?(12)
=> true
> lukuja.index(55)
=> 3
> lukuja.sort
=> [1, 4, 12, 23, 23, 55, 55, 101, 213]
```

Kaikenkaikkiaan Rubyssa on yli kolmekymmentä erilaista metodia, joista seuraavaan on taulukoita kaikista tärkeimmät ja hyödyllisimmät. Kuten merkkijonoissakin, on taulukoissakin osasta metodeja versiot huutomerkillä ja ilman, merkiten samaa kuin aikaisemmin; huutomerkitön metodi palauttaa vastauksen ja se pitää erikseen tallentaa, huutomerkillinen metodi muokkaa suoraan käytettyä taulukkoa:

Taulukko 3.4 Taulukon tärkeimmät metodit

Metodi	Selite	Esimerkki

`clear`	Tyhjentää taulukonn poistaen kaikki sen alkiot.	`lista = [1,2,3]`
		`> lista.clear`
		`=> []`

`collect, !`	Suorittaa taulukon kaikille alkioille koodiblokissa määritellyt muutokset.	`lista = ["a","b"]`
		`> lista.collect{|z| z +"?"}`
		`=> ["a?", "b?"]`

`compact, !`	Palauttaa taulukon muodossa, jossa siitä on poistettu kaikki tyhjät (`nil`) alkiot.	`lista = [1,2,nil,3]`
		`> lista.compact`
		`=> [1, 2, 3]`

`delete`	Poistaa annetun argumentin arvoisen alkion taulukosta. Palauttaa arvon jos jotain poistettiin, muuten `nil`.	`> taulu = [1,2,3]`
		`=> [1, 2, 3]`
		`> taulu.delete(1)`

		=> 1 > taulu => [2, 3]
delete_at	Kuten delete, mutta ottaa argumenttina poistettavan alkion sijainnin.	> taulu = [1,2,3,4] => [1, 2, 3, 4] > taulu.delete_at(3) => 4 i> taulu.delete_at(313) => nil
delete_if	Ottaa argumenttina koodilohkon, poistaa ne alkiot jotka eivät täytä annettua ehtolausetta.	> taulu => [1, 2, 3, 4, 5] > taulu.delete_if {|arvo| arvo < 3} => [3, 4, 5]
each	Ottaa syötteenä koodilohkon, suorittaa annetun koodin yhtä monta kertaa, kuin taulukossa on alkioita. Koodilohkoon määritelty muuttuja saa alkion arvon.	> taulu => [1, 2, 3] > taulu.each{|arvo|puts arvo*10} 10 20 30
empty?	Palauttaa arvon true jos taulukko on tyhjä ja alkioton, muutoin false	
eql?	Testaa onko taulukko samanlainen kuin argumenttina annettu taulukko. Ollakseen samanlaisia pitää taulukkojen olla yhtä pitkiä ja niissä pitää olla samanarvoiset alkiot samassa järjestyksessä.	> [1,2,3].eql?([1,2,3]) => true > [1,2,3].eql?([1,3,2]) => false > [1,2,3].eql?([1,2,3,4]) => false
first	Palauttaa ensimmäisen alkion sisältämän arvon.	
include?	Ottaa syötteenä alkion arvon.Palauttaa `true` jos taulukossa on argumenttia vastaava alkio, muuten `false`.	> [1,2,3].include?(1) => true > [1,2,3].include?(313) => false
join	Muodostaa taulukosta merkkijonon yhdistämällä alkiot toisiinsa. Ottaa argumenttina liitoksessa käytettävän välimerkin.	> ["A","B","C"].join("-") => "A-B-C"
last	Palauttaa viimeisen alkion sisältämän arvon.	
length	Palauttaa kokonaislukuna tiedon siitä, kuinka monta alkiota taulukossa on.	
pop	Poistaa tauukosta viimeisen alkion. Antaa paluuarvona poistetun alkion arvon.	> taulu = ["A","B","C"] => ["A", "B", "C"] > pois = taulu.pop => "C" > pois => "C"
push	Lisää argumentteina annetut arvot taulukon viimeiseksi alkioksi.	> taulu => ["A", "B"] > taulu.push("C","D") => ["A", "B", "C", "D"]
reverse, !	Palauttaa taulukon alkioiltaan käänteisessä järjestyksessä.	> [1,2,3].reverse => [3, 2, 1]

`sort, !`	Järjestelee listan aakkos- tai numerojärjestykseen. Vaihtoehtoisesti ottaa koolilohkon, jossa määritellään lajitteluun käytettävä koodi. Oletuslajittelijana käytetään lohkoa `{|x,y| x<=>y}`	`> lista = [1,5,3,7]`
		`=> [1, 5, 3, 7]`
		`lista.sort`
		`=> [1, 3, 5, 7]`
		`> lista =`
		`["ab","cd","aa","bd"]`
		`=> ["ab", "cd", "aa", "bd"]`
		`> lista.sort`
		`=> ["aa", "ab", "bd", "cd"]`

`uniq, !`	Palauttaa taulukon, josta on poistettu kaikki ne alkiot, jotka ovat kopioita aiemmista.	`>["aa","cd","bb","cd"].uniq` `=> ["aa", "cd", "bb"]`

Oheisilla työkaluilla on helppo muokata taulukon sisältöä, mutta tietenkin jossain vaiheessa tullaan tilanteeseen, jossa listaa olisi päästävä muuttelemaan. Tämä ei Rubyssa muodosta ongelmaa. Ensinnäkin, jos taulukon yksittäistä alkiota ainoastaan halutaan muuttaa, onnistuu se ylikirjoittamalla alkio samaan tapaan kuin esimerkiksi muuttujan arvo:

```
> lista = ["eka", "toka", "kolmas"]
=> ["eka", "toka", "kolmas"]
> kohta = 1
> lista[kohta] = "muutettu"
> lista
=> ["eka", "muutettu", "kolmas"]
```

Jos taulukkoon halutaan lisätä alkioita, voidaan se tehdä metodilla `.push(argumentti)`, jolloin annetusta argumentista tulee taulukon viimeinen alkio:

```
> lista
=> ["eka", "muutettu", "kolmas"]
> lista.push(123)
=> ["eka", "muutettu", "kolmas", 123]
```

Vastaavasti alkioiden poistaminen taulukosta onnistuu metodilla pop, joka poistaa taulukon viimeisen alkion ja antaa sen paluuarvona:

```
> lista.pop
=> 123
> lista
=> ["eka", "muutettu", "kolmas"]
```

Jos metodia pop käytetään tyhjään taulukkoon, antaa se paluuarvona arvon `nil`. Vastaavasti voidaan myös käyttää metodeja delete tai delete_at. Ensimmäiselle annetaan argumenttina poistettavan alkion arvo, toiselle alkion sijainti:

```
> lista = [1,2,3,4,5,6]
=> [1, 2, 3, 4, 5, 6]
> lista.delete(4)
=> 4
> lista
=> [1, 2, 3, 5, 6]
> lista.delete_at(0)
=> 1
> lista
=> [2, 3, 5, 6]
```

Taulukkoon voidaan myös antaa alkiona taulukko. Tämän menetelmän avulla voidaan koota moniulotteisia taulukoita, kuten esimerkiksi kaksiulotteisia matriiseja:

```
> lista
=> [[0, 0, 0, 0, 0], 0, 0, 0, 0]
```

```
> lista[1] = Array.new(5,0)
=> [0, 0, 0, 0, 0]
> lista
=> [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], 0, 0, 0]
```

Tässä tapauksessa alkioiksi annetut taulukot toimivat normaalisti, ainoana erona ylimääräinen viittaus toisen alkion sisään antamalla tarkennettu sijainti taulukossa:

```
> lista[1][0] = 1
=> 1
> lista
=> [[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], 0, 0, 0]
```

TULOSTUSKÄSKYT JA MUOTOILUMERKIT

Aivan ensimmäisessä ohjelmointitehtävässä toisen luvun lopussa käytettiin Rubyssa on tulostuskäskyn `puts` lisäksi toinenkin tulostuskäsky, joka on nimeltään `print`. Käskynä `print` toimii täysin samalla tavalla kuin puts, joskin sillä erotuksella, että rivin loppuun ei automaattisesti lisätä rivinvaihtomerkkiä:

```
puts ”Tämän rivin loppuun tulee rivinvaihto.”
print ”Tämän rivin loppuun ei tule.”
puts ”Siksi tämä rivi tulee suoraan edellisen perään.”
```

Tulostaa vastauksen

```
Tämän rivin loppuun tulee rivinvaihto.
Tämän rivin loppuun ei tule.Siksi tämä rivi tulee suoraan edellisen perään.
```

Eli periaatteessa `print`-käskyn ainoa ero on se, että puts-käskyllä rivin loppuun lisätään automaattisesti rivinvaihto, jota tekstissä kuvataan muotoilumerkillä `\n`. Jos tulostuskäskyyn laitetaan merkki `\n`, tuottaa se tulostukseen ylimääräisiä rivinvaihtoja:

```
>  puts ”Rivinvaihtoja! \n\n\n\n Loppu.”
=> Rivinvaihtoja!



 Loppu.
```

Eli `\n`-merkin avulla ohjelma voidaan pakottaa aloittamaan teksti uudelta riviltä. Rivinvaihtomerkin lisäksi Ruby-kielessä on seuraavat muotoilumerkit:

Taulukko 3.5 Rubyn erikois- ja muotoilumerkit

Merkki	Selite
\n	Rivinvaihtomerkki, tulostettaessa korvautuu rivinvaihdolla
\r	Rivin alkuunpaluumerkki; joissain käyttöjärjestelmissä toimii kuten \n.
\b	Backspace, eli pyyhkimismerkki; poistaa merkkiä edeltävän merkin.
\t	Sisennys, tuottaa käytetystä ohjelmasta riippuen joko sisennyksen tai sisennyksen verran välilyöntejä.
\s	Välilyönti, vastaa normaalia välilyöntiä merkkijonossa
\a	pääteohjelman hälytysmerkki; tuottaa järjestelmän oletushälytyksen, kuten vaikkapa vilkuttaa ajettavan ohjelman ikkunaa, tuottaa virheäänen tai vastaavaa. Ei toimi kaikissa ohjelmissa.
\	Ohitusmerkki; seuraavaa merkki kuuluu merkkijonoon eikä sitä käsitellä muotoilumerkkinä.

Taulukossa on myös esitelty ohitusmerkki (*escape* sequence), jota voidaan käyttää tapauksissa, joissa halutaan tulostaa merkkisarja, jonka Ruby voisi muutoin tulkita väärin. Esimerkiksi jos ohjelmassa tulee tarve tulostaa merkkipari `\` ja `n` niin tällöin tarvitaan ohitusmerkkiä. Ohjelmassa voidaan muotoilun ohitukseen käyttää yhtä `\`-merkkiä, eli tyyliin `puts "\\n"`, joka tulostaa vastaukseksi pelkän `"\n"`. Muotoiluohitus toimii myös niissä tapauksissa, että tekstiin halutaan tulostaa sitaattimerkki. Esimerkiksi tulostuskäsky

```
puts 'vaa'an alla'
```

Antaa virheilmoituksen, koska tulkki luulee keskimmäistä sitaattimerkkiä merkkijonon lopettavaksi merkiksi. Luonnollisesti virhe voitaisi myös korjata käyttämällä lainausmerkkejä (") korvaamaan sitaattimerkit, mutta ongelman voi myös kiertää muotoilunohituksella:

```
puts 'vaa\'an alla'
```

Joka siis tulostaa

```
vaa'an alla
```

TIEDON PYYTÄMINEN KÄYTTÄJÄLTÄ

Luonnollisesti ohjelmat jossain vaiheessa saavuttavat kokoluokan, jossa niihin tarvitaan käyttäjän kanssa vuorovaikutusta pelkkien tulostuskäskyjen ja muuttujien kanssa työskentelyn lisäksi. Esimerkiksi seuraavan valinnan, tai käytettävän laskukaavan valintaan, voidaan tarvita käyttäjältä päätös, jonka avulla ohjelma voi jatkaa suoritustaan. Helpoin tapa luoda vuorovaikutusta käyttäjän kanssa on varmaankin laittaa ohjelma toimimaan komentokehotteessa, eli merkkipohjaisessa käyttöliittymässä, joka on tuttu pitkälti Linux- ja Unix-käyttöjärjestelmistä.

Rubyssa syötteen lukemisesta vastaa komento gets, joka lukee käyttäjältä yhden rivin tekstiä (eli ensimmäiseen rivinvaihtomerkkiin asti), ja tallentaa se annetun kohdemuuttujan arvoksi:

```
> arvo = gets
teksti!<enter>
=> "teksti!\n"
> arvo
=> "teksti!\n"
```

Käyttäjän ajonaikaisesti kirjoittama teksti lihavoituna. Huomaa, että enter tulkitaan syötteen antamisen lopettamisen lisäksi rivinvaihtomerkiksi syötteen loppuun

Komentona `gets` on kutakuinkin vastine komennolle `puts`; `gets` lukee syötteen ja tallentaa sen annettuun muuttujaan, kuten yllä olevan koodin muuttuja arvo. Jos gets-komennolle ei määrätä muuttujaa, ei arvo tallennu minnekään. Tätä voidaan kuitenkin käyttää hyväksi siinä, että käyttäjän pitää painaa *enter* että ohjelma jatkaa eteenpäin. Tästä on esimerkiksi silloin hyötyä, jos halutaan varmistaa että käyttäjä lukee tai huomaa ohjelman antaman ilmoituksen.

Kirjoitettaessa puts-käskyä käyttäviä koodeja törmätään kuitenkin ongelmaan, koska esimerkiksi SciTE ei osaa kovinkaan tehokkaasti käsitellä ohjelman ajon aikana annettuja syötteitä. Esimerkiksi koodi

```
tulos = gets
puts gets
```

aiheuttaa jo sen, että SciTE ei enää saa suoritettua koodia. Tämän vuoksi ohjelmat, joissa käytetään käyttäjältä saatuja syötteitä, pitää suorittaa ajamalla ne tiedostonhallinnasta valitsemalla ne käsin. Tästä puhutaan lisää seuraavan esimerkin yhteydessä.

Esimerkki 3-2: Syötteen lukeminen käyttäjältä

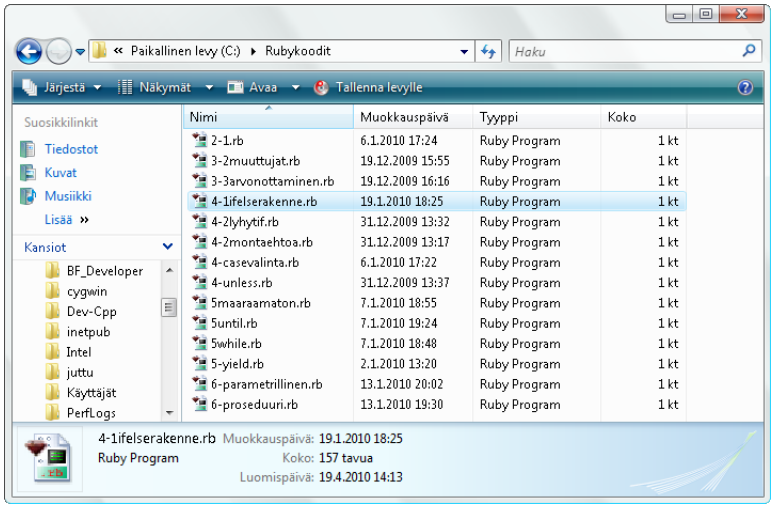
Lähdekoodi

```
01  #Pyydetään käyttäjältä syöte ja tulostetaan se
02
03  puts "Hei, kerro nimesi: "
04  nimi = gets
05  puts "Sanoit nimeksesi " +nimi
06  puts "Hauska tavata!"
07  #Lisätään loppuun vielä yksi keskeytys että näemme tuloksen
08  gets
```

Tuloste

```
Hei, kerro nimesi:
Jussi
Sanoit nimeksesi Jussi
Hauska tavata!
```

Kuten tulosteesta voidaan huomata, ei tätä ohjelmaa tavallisuudesta poiketen suoritetakaan SciTEssä, vaan suoraan valitsemalla lähdekoodin sisältävä tiedosto Windows-ikkunassa ja ajamalla se painamalla *Enter*. Tämä johtuu siitä, gets-käskyjen lukeminen ajonaikaisesti saatuina syötteinä ei onnistu SciTE-editorin suorittamassa ikkunassa. Jatkossa kaikki ohjelmat, joissa tarvitaan gets-komentoa, suoritetaan ajamalla lähdekoodi suoraan windows-ikkunasta lähdekoodieditorin sijaan. Käytännössä tämä siis tarkoittaa sitä, että lähdekoodi tulee sijoittaa paikkaan, mistä se on helppo löytää, ja valita kaksoisklikkaamalla lähdekooditiedostoa hiirellä. Esimerkiksi hakemisto C:\Rubykoodit\ on helppo löytää käsin koneen tiedostonhallinnasta.



Valittu Ruby-lähdekooditiedosto Windows-ikkunassa.

MUITA PERUSRAKENTEITA

Rubyssa on hieman poikkeavaisuuksia ja erikoisia toimintoja, joita ei monista muista ohjelmointikielistä löydy ja jotka ovat hyödyllistä perustietoa. Jos esimerkiksi muodostuu tarve muodostaa lista jossa on luvut yhdestä kolmeenkymmeneen yhteen, kuten vaikkapa päivämääriä pyydettäessä, voidaan se muodostaa käskyllä

```
> paiva = 1..31
```

Tärkeää tässä on nimenomaan kaksi pistettä. Jos väli määriteltäisi muodossa `1...31`, eli kolmella pisteellä, tarkoitaisi se lukujonoa yhdestä kolmenkymmeneen. Helpointa tämä on todistaa pienellä kokeella. Muodostetaan numeroväli yhdestä kolmeenkymmeneenyhteen ja tutkitaan metodi `include?`:n avulla kuuluuko luku määritellylle välille:

```
> paivat = 1..31
=> 1..31
> paivat.include?(1)
=> true
> paivat.include?(31)
=> true
```

Vaihdetaan tässä vaiheessa määrittelytavaksi `...`-määrittely:

```
> paivat = 1...31
=> 1...31
> paivat.include?(1)
=> true
> paivat.include?(31)
=> false
```

Helpointa asia on mieltää siten, että kahden pisteen määrittelemä väli sisältää kaikki luvut kyseiseltä väliltä, ja kolmen pisteen määritelmä kertoo katkaisukohdat mistä väleistä lukujono katkaistaan. Lisäksi kannattaa muistaa, että `1..31` muodostaa itseasiassa `Range`-nimisestä luokasta tehdyn lukulistan, joka ei sisällä pelkästään kokonaislukuja annetulta väliltä, vaan hyväksyy testattaessa kaikki luvut kyseiseltä väliltä, mukaan lukien desimaaliluvut:

```
> lista = 1...31
=> 1...31
> lista.include?(3.14)
=> true
```

Range-lukulistalla voi myös tehdä muita toimintoja, kuten ottaa listan isoimman tai pienimmän jäsenen, tai poistaa osia:

```
> luvut = 1..10
=> 1..10
> luvut.min
=> 1
> luvut.max
=> 10
> luvut.reject { |i| i ==5}
=> [1, 2, 3, 4, 6, 7, 8, 9, 10]
> luvut.reject{|i| i%2 > 0}
=> [2, 4, 6, 8, 10]
```

Viimeiset kaksi esimerkkiä ovat hieman erilaisia, sillä `reject` tarvitsee itselleen poistoehdon, joka annetaan erillisenä koodilohkona. Esimerkeissä ehdoksi on määritelty ne jäsenet, jotka ovat arvoltaan viisi, tai alemmassa vaihtoehdossa ne, joiden jakojäännös kahdella ei ole nolla, eli parittomat luvut.

Koodilohkot argumentteina

Aiemmissa esimerkeissä näytettiin muutamaan otteeseen esimerkki, jossa suoritettava metodi otti normaalien argumenttien sijaan syötteenä erillisen koodilohkon. Näissä tapauksissa metodia käytetään suorittamaan testi, jossa ensin määritellään muuttujannimi testattaville arvoille, ja toiseksi käsky joka kyseisessä metodissa suoritetaan. Syötettävä lohko on muotoa

```
<olio>.<metodi>{|<muuttuja>| suoritettava koodi}
```

Kun metodi saa syötteenä koodilohkon, liittyy siihen aina jonkin verran erikoisuuksia. Esimerkiksi lukulistaa käsitellessä muuttujaksi määräytyy aina kokonaisluku, ja koodissa määritellään ehto sille, mitä arvon pitää olla, että sitä ei poisteta listalta. Toisaalla, esimerkiksi tiedostoa luettaessa, määritellään mitä tiedostosta luetulle riville tehdään. Tähän tullaan vasta myöhemmin, mutta esimerkiksi koodi

```
luettava.each {|rivi| puts rivi}
```

Lukee aina tiedostosta rivin ja tallentaa sen muuttujaan rivi, sekä tämän jälkeen tulostaa sen ruudulle. Mielenkiintoisena huomiona kannattaa myös huomata, että suoritettavassa koodissa voi olla useampi rivi. Esimerkiksi koodi

```
luettava.each {|rivi| puts rivi; puts ”Kobria!”}
```

On täysin toimiva koodi, jossa jokaisen tiedostosta luetun ja ruudulle tulostetun rivin väliin tulostetaan huudahdus ”Kobria!”. Jatkossa, mikäli metodi ottaa argumenttinaan koodilohkon, selitetään sen toiminta tarkemmin asiayhteydessään.

Merkkijonojen leikkaukset

Merkkijonojen muokkausmetodien lisäksi merkkijonoihin liittyy läheisesti toinen näppärä ominaisuus, eli merkkijonojen leikkaukset. Käytännössä nämä tarkoittavat sitä, että merkkijonosta voidaan ottaa yksittäisiä osakokonaisuuksia määrittelemällä mistä mihin tai mikä osa merkkijonosta otetaan:

```
>teksti = "Totuus ei pala tulessakaan"
=> "Totuus ei pala tulessakaan"
>teksti[14]
=> 32
>teksti[15]
=> 116
>teksti[4, 12]
=> "us ei pala t"
>teksti[5..10]
=> "s ei p"
>teksti[-10..-3]
=> "ulessaka"
```

Leikkaukset toimivat siis seuraavalla tavalla: Jos leikkaukseksi annetaan pelkästään yksi numero, palauttaa Ruby merkkitaulukon muokaisen kokonaisluvun joka vastaa kyseisellä paikalla olevaa merkkiä. tiedon

Satunnaisluku

Useissa ohjelmissa tarvitaan satunnaislukua esimerkiksi tuottamaan satunnainen reaktion pelaajan toimintaan, arpomaan lukuja tai esimerkiksi luomaan salauksessa tarvittavia avaimia. Ruby-kielessä satunnaisluku voidaan arpoa käskyllä `rand`.

```
> rand
=> 0.0970348568209023
> rand
=> 0.246033784118693
```

`rand` palauttaa satunnaisen liukuluvun väliltä 0 ja 1, ja se arvotaan joka kerta uudelleen. Jos luvusta esimerkiksi halutaan tehdä kokonaisluku väliltä 0-100, voidaan se toteuttaa antamalla käskylle argumenttina tieto siitä, miltä väliltä luku pitää ottaa:

```
> rand(100)
=> 34
```

Tässä tapauksessa Ruby arpoo luvun väliltä nollasta sataan, ja palauttaa arvon kokonaislukuna. Eli jos ohjelmassa halutaan vaikkapa mallintaa kuusisivuisen nopan heittoa, voidaan se tehdä käskyllä rand(6):

```
> rand(6)
=> 5
> rand(6)
=> 4
> rand(6)
=> 0
```

Käytettäessä rand-käskyä pitää kuitenkin huomata, että satunnaislukujoukkoon kuuluu aina 0, muttei annettu yläraja, eli käskyllä rand(6) tulos voi olla joko 0, 1, 2, 3, 4 tai 5, mutta ei koskaan 6. Tämä pitääkin muistaa aina satunnaislukua käytettäessä, ja nopan tapauksessa korjata lisäämällä satunnaisesti arvottuun lukuun yksi:

```
> rand(6) + 1
=> 3
> rand(6) + 1
=> 5
> rand(6) + 1
=> 6
```

Nyt noppa näyttäisi toimivan oikein. Jos satunnaisen luvun halutaan olevan desimaali-, eli liukuluku, se joudutaan toteuttamaan ilman määriteltyä ylärajaa, pelkän `rand`-käskyn avulla, tai vaihtoehtoisesti rakentamaan yhdistämällä kaksi `rand`-käskyä keskenään. Esimerkiksi liukuluku väliltä nollasta kahteenkymmeneen voidaan tehdä käyttäen kahta `rand`-käskyä:

```
> rand(20) + rand
=> 16.68486799249411
```

Kommentit

Ruby-kielessä kommentit merkitään lähdekoodiin risuaitamerkillä (#). Merkki tarkoittaa sitä, että kyseisellä rivillä olevaa tekstiä ei huomioida tulkin suorittaessa koodia. Otetaan tarkasteltavaksi seuraava koodi:

```
01  puts "Tekstiä!"
02  #Tämä rivi on kommentoitu
03
04  #Allaoleva rivi on kommentoitu pois käytöstä
05  #puts "Tätä ei tulosteta!"
06  #Kommenttikenttä voi jatkua myös useita rivejä
07
08  puts "Lopetetaan."  #Kommentti voi olla myös rivin lopussa
```

Ohjelma tulostaa suoritettaessa tekstin

```
>ruby kommentit.rb
Tekstiä!
Lopetetaan.
>Exit code: 0
```

Eli lähdekoodiriveistä ainoastaan ne, jotka eivät ala kommenttimerkillä huomioidaan suorituksessa. Esimerkiksi rivillä 5 oleva käsky olisi toimivaa lähdekoodia, mutta se on poistettu käytöstä kommenttimerkillä. Lisäksi rivillä 8 on koodin jälkeen tuleva, ns. *inline-kommentti*, eli käskyn perässä oleva lyhyt selite. Lisäksi Rubyssa on toinenkin kommenttimerkkipari, `=begin` ja `=end`. Nämä käskyt sijoitetaan omille riveilleen, ja niiden väliin jäävä alue jätetään automaattisesti pois ohjelmaa suoritettaessa:

```
01  =begin
02  tulos = -2
03  if tulos == 10
04    puts "Täydellinen suoritus."
05  else
06    puts "No eihän se nyt ihan nappiin mennyt."
07  end
08
09  =end
10
11    puts "Tämä on ainoa suoritettava rivi."
```

Ylläoleva ohjelma ei tulosta siis mitään muuta. kuin rivillä 11 olevan tulostuskäskyn, koska sen yllä olevasta koodista on merkitty pois käytöstä rivien 1 ja 9 väliin jäävä alue.

Lähdekoodin rivittämisestä

Viimeinen luvussa käsiteltävä asia koskee Ruby-kielen tapaa hallita rivinvaihtoja. Tähän asti on kirjan koodiesimerkeissä käytetty pelkkää lähestymistapaa, jossa jokaiselle riville laitetaan aina yksi käsky, eli tyyliin

```
puts "Ensimmäinen tulostus."
puts "Toinen tulostus."
```

jolloin jokainen käsky saa käyttöönsä yhden rivin. Vaikka tämä lähestymistapa tekeekin koodista helposti luettavaa, voi se joskus myös aiheuttaa sen, että koodista kasvaa pitkä, varsinkin jos koodissa on peräkkäin useita lyhyitä käskyjä.

Ruby tarjoaakin mahdollisuuden laittaa useamman käskyn samalle riville käyttämällä puolipistettä erottamaan käskyt toisistaan. Tämä C-kielestäkin tuttu lähestymistapa mahdollistaa siis sen, että käskyjä voidaan laittaa riville useita peräkkäin:

```
  luku = 0; luku = luku + 100
  luku = luku *3.54; puts luku.to_s
```

Ohjelma tulostaa vastaukseksi

```
354.0
```

Ensimmäisellä rivillä määritellään muuttuja luku, ja rivin toisena käskynä lisätään siihen sata. Toisella rivillä kerrotaan arvo 3.54:llä ja toisen rivin toisena käskynä tulostetaan laskutoimituksen vastaus. Puolipisteen avulla mikä tahansa käsky voidaan kirjoittaa seuraavan perään; syntaksisessa mielessä Ruby näkee puolipisteen samanarvoisena rivinkatkaisijana kuin rivinvaihdon. Puolipisteestä onkin ehkä eniten hyötyä silloin, kun sillä määritellään yhden rivin valintalauseita, kuten

```
  if luku > 100; puts "Aika paljon."; end
```

joka on täydellinen ja täysin toimiva valintarakenne, jota voidaan käyttää vaikkapa irtometodin kutsumiseen. Tästä puhutaan lisää seuraavassa luvussa.

Joskus taas vastaavasti voi käydä toisinpäin, eli käskylause ei mahdu yhdelle fyysiselle riville, joten sitä pitää jatkaa seuraavalle riville. Tällöin katkaisukohtaan laitetaan merkki \, joka kertoo tulkille, että allaoleva rivi on itse asiassa jatkoa nyt päättyvälle riville:

```
puts "Todella pitkä tuloste joka jatkuu ja jatkuu ja jatkuu \
ja jatkuu ja jatkuu ja jatkuu..."

puts "Loppui sentään."
```

Tulostaa siis

```
Todella pitkä tuloste joka jatkuu ja jatkuu ja jatkuu ja jatkuu ja jatkuu ja
jatkuu...
Loppui sentään.
```

LUKU 4

Loogiset operaattorit ja valintarakenteet

Loogiset operaattorit
Valintarakenteet, if, case ja unless

Tässä luvussa käsitellään Ruby-kielen keskeisimmät operaattorit, joilla voidaan muodostaa väittämiä ja ehtolauseita. Luvussa esitellään myös valintarakenteet, jotka yhdistettynä loogisiin operaattoreihin antavat mahdollisuuden valita, mikä osa ohjelmasta seuraavaksi suoritetaan.

LOOGISET OPERAATTORIT

Luvun ensimmäinen pääaihe on loogiset operaattorit ja loogisten lausekkeiden kokoaminen. Rubyssa ja muissakin ohjelmointikielissä operaattoreita käytetään kun ohjelmassa halutaan muuttaa tai määritellä muuttujien arvoja. Esimerkiksi jos ohjelmassa halutaan laskea yhteen kaksi kokonaislukua

```
>1+2
=> 3
```

käytetään ohjelmassa yhteenlaskuoperaattoria +, sekä lukuja 1 ja 2, joita sanotaan operandeiksi. Yhteenlaskuoperaattori onkin yksi matemaattisista operaattoreista, joita Rubyssa on sisäänrakennettuna kuusi kappaletta:

Taulukko 4.1 Matemaattiset operaattorit

Operaattori	Selite	Esimerkki
`X + Y`	Yhteenlasku, summa. Laskee operandit yhteen.	`>1+1` `=> 2`
`X – Y`	Vähennyslasku, erotus. Vähentää operandit toisistaan.	`>1–1` `=> 0`
`X * Y`	Kertolasku, tulo. Kertoo operandit keskenään.	`> 10*10` `=> 100`
`X / Y`	Jakolasku, osamäärä. Jakaa ensimmäisen operandin toisella, ei tee pyöristystä.	`> 10 / 5` `=> 2`
`X % Y`	Jakojäännös. Palauttaa jakojäännöksen.	`>10 % 3` `=> 1`
`X ** Y`	Eksponenttilasku, korottaa ensimmäisen operandin toisen potenssiin.	`> 10**3` `=> 1000`

Toinen tässä vaiheessa jo useampaan otteeseen käytetty operaattori on sijoitusoperaattori =. Tämän operaattorin avulla määrätään muuttujille arvoja, ja se onkin yksi toisesta operaattoriperheestä. Rubyssa on perus-sijoitusoperaattorin lisäksi olemassa pikavalinnat matemaattisten operaattorien käyttämiseen:

```
> luku = 47
=> 47
> luku –= 5
=> 42
> luku *=10
=> 420
```

Sijoitusoperaattorien yhteyteen laitettu laskuoperaattori siis lyhentää koodia hieman, samalla muuttaen kohdemuuttujan arvoa. Esimerkin `luku –=5` vastaa käytännössä koodiriviä `luku = luku –5`.

Taulukko 4.2 Sijoitusoperaattorit

Operaattori	Selite
`X = Y`	Normaali sijoitusoperaattori, asettaa X:n arvoksi Y:n.
`X += Y`	Yhteenlaskeva sijoitus, korvaa koodin `X = X + Y`
`X –= Y`	Vähentävä sijoitus, korvaa koodin `X = X – Y`
`X *= Y`	Kertova sijoitus, korvaa koodin `X = X * Y`
`X /= Y`	Jakava sijoitus, korvaa koodin `X = X / Y`

`X %= Y`	Jakojäännössijoitus, korvaa koodin `X = X % Y`
`X **= Y`	Eksponenttisijoitus, korvaa koodin `X = X ** Y`

Varsinaisesti keskeisiä operaattoreita ohjelmointikielessä kuitenkin ovat loogiset operaattorit, joiden avulla voidaan rakentaa vertailulauseita, jotka toimivat valinta- ja toistorakenteiden ehtolauseina. Käytännössä tämä siis tarkoittaa sitä, että meillä on joukko loogisia operaattoreita, joiden avulla voidaan tehdä vertailulause, esimerkiksi `luku > 10`, luku on suurempi kuin kymmenen, tai `teksti.length == 140`, muuttujan teksti sisältämän merkkijonon pituus on tasan 140 merkkiä.

Vertailuoperaattorit toimivat pääsääntöisesti siten, että ne palauttavat joko arvon `true` tai `false` riippuen siitä, onko esitetty väittämä totta vai ei:

```
> 1 == 1
=> true
> 10 < 5
=> false
> arvo = 100
> arvo < 1000
=> true
> 1.eql? 1  #Sama kuin == -operaattori
=> true
```

`true` ja `false` ovat Rubyssa varattuja sanoja, ja esiintyvät aina lähdekoodissa ilman sitaattimerkkejä tai muita erikoismerkkejä. Niitä voidaan myös käyttää osana itse väittämää:

```
> true == true
=> true
> (10 < 5) == false
=> true
```

Kannattaa kuitenkin huomata, että Rubyssa true ja false eivät ole synonyymejä lukuarvoille 1 ja 0 kuten useissa muissa ohjelmointikielissä. Lisäksi `nil` ei tarkoita samaa kuin `false`; nil on kirjaimellisesti ”ei mitään”, kun false tarkoittaa ”epätosi”:

```
> 1 == true
=> false
> nil == false
=> false
```

Ainoan poikkeuksen true/false-käytäntöön tekee operaattori <=>, joka palauttaa arvon 1 jos ensimmäinen operandi on suurempi, arvon 0 jos operandit ovat samanarvoisia ja arvon -1 jos toinen operandi on suurempi:

```
> 42 <=> 13
=> 1
> 42 <=> 42
=> 0
> 42 <=> 313
=> -1
```

Rubyssa onkin vertailulausekkeita varten kahdeksan erilaista operaattoria, joiden avulla väittämiä voidaan rakentaa:

Taulukko 4.3 Loogiset operaattorit

Operaattori	Selite
`X == Y`	Yhtäsuuruustesti, palauttaa true jos X ja Y on samanarvoisia.
`X.eql? Y`	Sama kuin ==, toimii syntaksisesti X-olion metodina.

`X != Y`	Erisuuruustesti, palauttaa true jos X ja Y ovat eri arvoja.
`X < Y`	Pienempi kuin, palauttaa true jos X on pienempi kuin Y
`X <= Y`	Pienempi tai yhtäsuuri, palauttaa true jos X on pienempi tai sama kuin Y.
`X > Y`	Suurempi kuin, palauttaa true jos X on suurempi kuin Y
`X >= Y`	Suurempi tai yhtäsuuri, palauttaa true jos X on suurempi tai sama kuin Y.
`X <=> Y`	Yhdistelmätesti. Palauttaa arvon –1 jos Y > X, 0 jos X == Y ja 1 jos X > Y.

Operaattoreita käytettäessä kannattaa olla jonkin verran tarkkana, sillä luonnollisesti epäloogiset testit, kuten merkkijonon ja lukuarvon kokoerojen mittaaminen, tai tekstin lisääminen lukuarvoon aiheuttavat sen, että ohjelma kaatuu ArgumentError-virheeseen:

```
> "tekstiä" == 3
=> false
> "tekstiä" > 123
ArgumentError: comparison of String with 123 failed
```

Poikkeuksen tähän sääntöön tekee kuitenkin yhtäsuuruustesti, joka yksinkertaisesti palauttaa `false`, jos oliot eivät ole keskenään vertailukelpoisia. Yhtäsuuruustestin ongelma kuitenkin on se, että se ei tunnista samaa arvoa, mutta eri tyyppiä olevia arvoja yhtäsuuriksi. Esimerkiksi merkkijono ”313” ja lukuarvo 313 eivät ole yhtäsuuruustestille sama asia, vaan ensimmäinen on merkkijono ”kolme-yksi-kolme” ja toinen lukuarvo kolmesataakolmetoista:

```
> "313" == 313
=> false
```

Vastaavasti myöskään laskutoimitukset eivät onnistu merkkijonomuodossa oleviin numeroarvoihin:

```
> "300" + 10
TypeError: can't convert Fixnum into String
```

VALINTARAKENTEET IF JA CASE, UNLESS

Toinen luvun pääaihe on valintarakenteet, joita Ruby-kielestä löytyykin useita eri käyttötarkoituksiin. Valintarakenteen avulla voidaan siis päättää valintaehdon perusteella, mikä osa valitarakenteesta suoritetaan. Valintaehto taas muodostetaan esimerkiksi muuttujista ja loogisista operaattoreista.

Yksinkertaisimmillaan valintarakenteet toimivat siten, että valintaehdon avulla päätetään suoritetaanko jokin osa lähdekoodia vai ei riippuen siitä, saako valintaehto arvon true vai false. Useimmiten tämä ei kuitenkaan riitä, vaan valintaan tehdään vaihtoehtoisia rakenteita eri tilanteita varten. Lisäksi monesti määritellään myös else-lohko eli osio, joka suoritetaan siinä tapauksessa, ettei yhtäkään muuta lohkoa voitu valita.

If-elsif-else

If-elsif-else on Rubyn tavallisin valintarakenne, ja setoimii täysin samalla tavoin kuin missä tahansa muussakin ohjelmointikielessä. Koodiin kirjoitetaan avainsana if, ja valintaehto, joka jälkeen tulee koodiosio joka suoritetaan siinä tapauksessa että ehto on totta:

```
if <valintaehto>
   <tämä suoritetaan jos valintaehto on true>
   <tämä suoritetaan jos valintaehto on true>
   <tämä suoritetaan jos valintaehto on true>
end
```

eli esimerkiksi vaikkapa

```
if 100 > 10
  puts "Tottakai 100 on enemmän kuin 10"
end
```

Ruby-kielessä if-valinta päättyy aina yhteen end-käskysanaan, joka kertoo tulkille, että valintarakenne loppuu tähän. Lisäksi Rubyssa on mahdollista käyttää then-syntaksisanaa valintaehdon ja suoritettavan koodiosion erottimena tähän tapaan

```
if <valintaehto> then
   <tämä suoritetaan jos valintaehto on true>
   ...
```

Syntaksisana then ei kuitenkaan ole pakollinen jos suoritettava koodiosio on erillään ja alkaa seuraavalta riviltä, joten sitä harvoin tulee käytettyä poislukien erikoistapaukset joista puhutaan myöhemmin. Toinen asia, joka valintaan voidaan lisätä, on else-osio. Jos if-valintaan lisätään else-osio, joka siis suoritetaan jos if-osiota ei suoriteta, tulee else-osio if-osion ja end-osan väliin:

```
if <valintaehto>
   <tämä suoritetaan jos valintaehto on true>
   <tämä suoritetaan jos valintaehto on true>
   <tämä suoritetaan jos valintaehto on true>
else
   <tämä suoritetaan jos valintaehto on false>
   <tämä suoritetaan jos valintaehto on false>
end
```

Eli vaikkapa

```
if "Oikein" == "Väärin"
  puts "Oletkos nyt ihan varma?"

else
  puts "Tietenkään oikein ei ole väärin."

end
```

Lisäksi if-valintaan voidaan lisätä elsif-osioita if- ja else-lohkojen väliin uusiksi vaihtoehdoiksi:

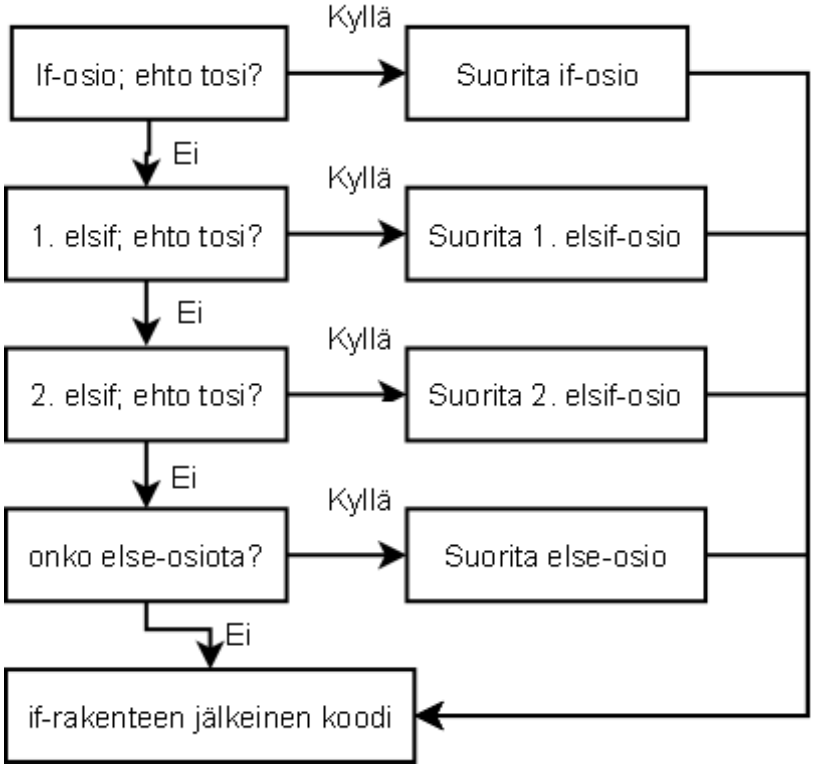
```
if <valintaehto>
   <tämä suoritetaan jos valintaehto on true>
   <tämä suoritetaan jos valintaehto on true>
elsif <2.valintaehto>
   <tämä suoritetaan jos 2. valintaehto on true>
   <tämä suoritetaan jos 2. valintaehto on true>
elsif <3.valintaehto>
   <tämä suoritetaan jos 3. valintaehto on true>
   <tämä suoritetaan jos 3. valintaehto on true>
else
   <tämä suoritetaan jos valintaehto on false>
end
```

Eli vaikkapa näin:

```
vari = "valkoinen"

if vari == "musta"
  puts "Aika synkkää"

elsif vari == "punainen"
  puts "Punainen on joulun väri."

elsif vari == "valkoinen"
  puts "Valkoinen on ihan jees."
```

```
else
  puts "En osaa nyt kyllä arvata."

end
```

Elsif-osioita voidaan laittaa if- ja else-osioiden väliin vapaasti niin monta kuin tarvitaan. If-valintarakennetta suunnitellessa kannattaa kuitenkin muistaa, että ainoastaan yksi osio valinnasta suoritetaan. Jos esimerkiksi if-osio ja ensimmäinen elsif-osio voitaisi valintaehdon puolesta suorittaa, suoritetaan aina se osio, johon ensin päädytään, eli tässä tapauksessa if. Muut osat valintarakenteesta ohitetaan, ja ohjelma jatkuu osion koodin suorituksen jälkeen end-käskyn jälkeiseltä seuraavalta riviltä.



If-valintarakenteen toiminta. Jos valintaehdon lopputulos on true, suoritetaan ensimmäinen tällainen lohko, tai mikäli yksikään ei ole true, suoritetaan else-osio.

Lisäksi if- ja elsif-osion valintaehtoon voidaan määritellä useita osia yhditelemällä useita valintaehtoja and, or ja not-avainsanoilla. And-ehto vaatii sitä, että molemmat – tai kaikki – ehtolauseet saavat arvon true. Vastaavasti or tarkoittaa sitä, että ainakin yhden ehdoista on oltava true. Lisäksi avainsanalla not voidaan ns. ”kääntää vastaus ympäri”; jos valintaehto olisi true, muuttaa not-avainsanan sen arvoksi false ja toisinpäin. Seuraavassa taulukossa on muutamia esimerkkejä ehtolauseiden yhdistämisen tuloksista:

Taulukko 4.4 Usean ehtolauseen käyttäminen, and, or ja not

Ehtolauseiden tulokset	Väittämän tulos
`not true`	`false`
`true and true`	`true`
`true and false`	`false`
`false or true`	`true`
`false or false`	`false`

Periaatteessa useaa ehtolaudetta käyttäessa riittää kun muistaa, että and-yhdistettä käytettäessä yksikin false riittää aiheuttamaan sen ettei koodia suoriteta, ja or-lauseessa yksikin true riittää siihen että koodi suoritetaan. Käytännössä usean ehtolauseen käyttämisessä on vielä toinen ongelma:

```
vari = "valkoinen"
luku = 100

if vari == "musta" and luku == 100
  puts "Nyt ei tainnut mennä oikein?"

elsif vari == "valkoinen" and luku == 100
  puts "Molemmat ehdot on oikein!"

elsif vari == "sininen" or luku == 100
  puts "Myös tämä kelpaisi, mutta ylempi ehto valittiin ensin!"

end
```

Ohjelma tietenkin tulostaa keskimmäisen vaihtoehdon, `Molemmat ehdot on oikein!`. Oheinen esimerkki samalla demonstroi sitä, kuinka viimeinen `elsif` jää nyt suorittamatta, koska ylempi `elsif` valittiin ensin; `and` vaatii että molemmat ehdot täyttyvät, mitä `if`-rivillä ei tapahdu, mutta kolmannessa osiossa `or`-avainsana mahdollistaisi sen, että pelkkä `luku == 100` riittäisi osion suorittamiseen. Useaa ehtolausettä käytettäessä, ja monivaihtoehtoista valintarakennetta kootessa tuleekin kiinnittää huomiota siihen, että ainoastaan yksi koodilohko suoritetaan, ja siksi valitaan aina loogisesti ensimmäinen kaikki ehdot täyttävä rakenne. Jos sopivia lohkoja on useampi ja niistä kaikki olisi tarkoitus suorittaa, pitää if-elsif-rakenne purkaa useaksi peräkkäiseksi if-rakenteeksi. Tässä vielä muutama esimerkki valintarakenteiden käyttämisestä:

Esimerkki 4-1: Valintarakenne, if-else

Lähdekoodi

```
01  #Tehdään yksinkertainen valinta
02  luku = 10
03
04  if luku > 100
05    puts "Luku on suurempi kuin 100"
06
07  else
08  puts "Luku on pienempi kuin 100"
09
10  end
```

Tuloste

```
>ruby 4-1ifelserakenne.rb
Luku on pienempi kuin 100
>Exit code: 0
```

Valintarakenteita käytettäessä tullaan myös ensimmäistä kertaa tilanteeseen, missä ohjelman rakenne alkaa muuttua monimutkaisemmaksi koska ohjelmassa on valintarakenteiden myötä ensimmäistä kertaa vaihtoehtoisia suorituspolkuja ja sisäkkäisiä rakenteita. Luettavuuden parantamiseksi monet ohjelmat tekevätkin automaattisesti pienen sisennyksen erottaakseen eri koodiosiot toisistaan. Tämän ansiosta loogisesti peräkkäin olevien osion hahmottaminen on helppoa, esimerkiksi silloin kun koodissa on useita sisäkkäisiä valintoja:

Esimerkki 4-2: Valintarakenne, sisäkkäiset ehdot

Lähdekoodi

```
01  #Pyydetään nimi
02  puts "Anna nimi: "
03  nimi = gets
04
05  #määritellään sopivat nimet
06  oknimet = ["Jussi", "Mikko","Anna"]
07
```

```
08  #Käyttäjän antamassa syötteessä on rivinvaihto lopussa!
09  if oknimet.include?(nimi.chomp!) == true
10    puts "Anna salasana: "
11    salasana = gets
12    if salasana.chomp! == "salakala"
13      puts "Tervetuloa #{nimi}!"
14    else
15      puts "Taidatkin olla joku muu?"
16    end
17  else
18    puts "En tunne sinua!"
19  end
```

Tuloste

Suoritetaan ohjelma muutaman kerran ja tarkastellaan tuloksia:

```
Anna nimi:
Esa
En tunne sinua!

Anna nimi:
Jussi
Anna salasana:
pallokala?
Taidatkin olla joku muu?

Anna nimi:
Jussi
Anna salasana:
salakala
Tervetuloa Jussi!
```

Ensinnäkin, kun ohjelmassa on useita koodilohkoja, voi niitä SciTEssä ottaa näkyviin ja piilottaa painamalla lohkon aloittavan rivin kohdalla olevaa miinus-merkkiä. Esimerkin koodissa kummankin if-rivin kohdalle ilmestyy merkki, jonka avulla koko tämän if-rivin ja siihen liityvän end-käskyn välisen osan voi piilottaa kerralla. Toisekseen, koodissa on esimerkit siitä, kuinka käyttäjältä luettu syöte sisältää aina rivinvaihtomerkin, joka pitää ottaa erikseen pois chomp-metodilla. Lisäksi esimerkistä on helppo huomata miksi koodi kannattaa rivittää; erityisesti else-osiot menevät helposti keskenään sekaisin.

Loppukommenttina kannattaa vielä huomata, että Rubyssa on mahdollista tehdä yhden rivin mittaisia lyhyitä if-valintarakenteita käyttämällä muotoa

```
if <valintaehto> then <suoritettava käsky> end
```

Tätä voidaan käyttää esimerkiksi silloin, kun ainoastaan halutaan testata nopeasti yksittäistä ehtoa ja vaikkapa kutsua sen seurauksena metodia:

```
01 #Lyhyt versio if-valinnasta
02  sana = "testi"
03
04  if sana == "testi" then puts "Oikein!" end
```

Ohjelma tulostaa vastauksena yksinkertaisesti

```
>ruby 4-2lyhytif.rb
Oikein!
>Exit code: 0
```

Tämä muoto on nopea ja yksinkertainen tapa tehdä testejä, mutta ”lyhyeen if-valintaan” ei voida kytkeä else- tai elsif-osioita. Lyhyelle if-rakenteelle on olemassa myös toinen kirjoitusmuoto;

```
<valintaehot> ? <jos true> : <jos false>
```

Eli vaikkapa

```
> hinta = 100
=> 100
> arvio = hinta < 50 ? "halpa": "kallis"
=> "kallis"
> arvio
=> "kallis"
> uusiarvio = hinta < 200 ? "halpa": "kallis"
=> "halpa"
```

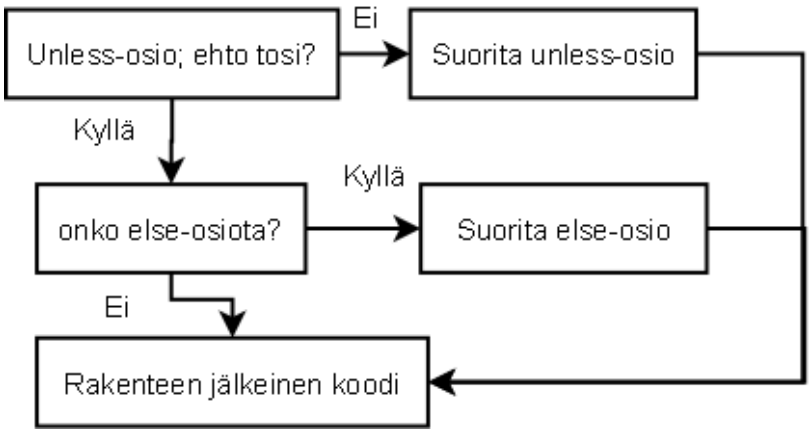
Tämä C-kielestä lainattu kirjoitusmuoto sopii vaikkapa pieniä muuttujan arvoa testaavia valintalauseita varten.

Joskus taas voi olla tarve tehdä ns. käänteinen valinta, eli suorittaa operaatio ainoastaan silloin, jos suoritusehto ei toteudukkaan. Toisin kuin monissa muissa kielissä, Rubyssa on erillinen valintarakenne tätä toimintoa varten.

Käänteinen valinta, unless

Joissain tapauksissa lähdekoodissa tulee tilanne, jossa jokin koodi tulisi suorittaa siinä tapauksessa, että jokin ehto ei toteudu. Useimmissa kielissä tämä on helpointa toteuttaa muotoilemalla valintarakenne muotoon `if not <valintaehto>`, mutta Rubyssa tätä varten on olemassa erillinen valintarakenne `unless`, jolla voidaan korostaa sitä tarkoitusta, että kyseessä on negatiivinen valinta. Käytännössä tämä siis tarkoittaa sitä, että `unless` toimii kuten if-rakenne, mutta koodiosio suoritetaan vain jos valintaehto saa arvon false.

unless-valintarakenne; koodi suoritetaan jos valintaehto saa arvon false. unless-valinnassa ei ole elsif-osioita.



Toiminnaltaan unless on täysin identtinen if-rakenteen kanssa, joskin sillä erolla että unless-valinnassa ei ole mahdollisuutta käyttää elsif-osioita, mutta else-osion määrittely on mahdollista.

Esimerkki 4-3: Käänteinen if-rakenne, unless

Lähdekoodi

```
01  luku = 100
02
03  unless luku > 100
04    luku = luku - 99
05    puts "Luvusta vähenettiin 99"
06  else
07    puts "Luku menisi liian pieneksi"
08  end
09
10  puts "Muuttuja luku on nyt #{luku}"
```

Tuloste

```
>ruby 4-unless.rb
Luvusta vähenettiin 99
```

```
Muuttuja luku on nyt 1
>Exit code: 0
```

Kuten esimerkistä nähdään, toimii `unless` tosiaankin täysin samoin kuin if-else-rakenne, paitsi että siinä suorittamisehtona on se, että annetut ehdot eivät toteudu. Käytännössä `unless`-rakenteen käyttäminen on pitkälti oma valinta, eikä se tarjoa mitään suoranaista helpostusta ohjelmointiin, mutta sen avulla on helpompi ymmärtää myöhemmin mitä koodi tekee.

Case-valinta

Rubyssa on myös toinen valintarakenne, joka poikkeaa jossakinmäärin if- ja unless-valintarakenteesta. Tämä valintarakenne on case, jossa suoritettava koodilohko valitaan ennaltamäärätyistä vaihtoehdoista sen sijaan, että valinta tehtäisi pelkästään valintaehtojen perusteella.

Periaatteessa Case-valinta on hyvin yksinkertainen. Ohjelmaan määritellään case-valintarakenne avainsanalla case ja valinnassa käytettävä muuttuja, vaikkapa

```
case tila
```

tämän jälkeen määritellään kaikki ne vaihtoehdot, joissa case valitaan. Käytännössä casea käytetään niissä tapauksissa, kun ohjelmassa tiedetään kaikki vaihtoehdot, joita valintaehdossa voi olla, ja reagoida sen mukaan. Periaatteessa case-valinnan rakenne on seuraavanlainen:

```
<vastaus> = case <muuttuja>
when <vaihtoehto1> then <tulos1>
when <vaihtoehto2> then <tulos2>
when <vaihtoehto3> then <tulos3>
when <vaihtoehto4> then <tulos4>
else <muu tulos>
end
```

Esimerkissä tietenkin oletetaan, että ratkaisuun on neljä todennäköisintä vaihtoehtoa, sekä `else`:llä määritelty ”muu” tulos. Käytännössä koodi toimii seuraavalla tavalla:

```
pisteet = rand(100)

tulos = case pisteet
  when 0..49 then "hylätty"
  when 50..69 then "tyydyttävä"
  when 70..89 then "hyvä"
  when 90..99 then "kiitettävä"
  else "hämmentävä"
  end

puts "Testin tulos on " + tulos+"."
```

Ohjelma arpoo luvun väliltä nollasta 99:ään, ja tämän jälkeen määrittelee arvosanan pisteiden perusteella käyttäen case-valintaa. Tässä esimerkissä vaihtoehdot määritellään lukualueina, ja tuloksena tulee pelkkä merkkijono.

Käytännössä case-valintaa voidaan kuitenkin käyttää myös toisella tavalla, if-elsif-else-rakenteen korvaajana. Tässä tapauksessa vastausta ei talleteta case-rivillä määriteltyyn muuttujaan, joka aiemmassa esimerkissä oli siis tulos, ja tulos-osiot korvataan koodilohkoilla. Tätä lähestymistapaa case-valintaan käytetäänkin seuraavassa esimerkissä.

Esimerkki 4-4: case-valintarakenne

Lähdekoodi

```
01  vari = "punainen"
02  arvo = 0
03
04  case vari
05    when "sininen" then
06      puts "Väri on sininen"
```

```
07      arvo = 1
08    when "punainen" then
09      puts "Väri on punainen"
10      arvo = 2
11    else
12      puts "Väri on joku muu. "
13      arvo = 0
14    end
15  puts arvo.to_s
```

Tuloste

```
>ruby 4-casevalinta.rb
Väri on punainen
2
>Exit code: 0
```

Kuten tulosteesta voidaan huomata, on case-valinnalla mahdollista korvata if-elsif-else-valintarakenne. Käytännössä kuitenkin case-valinta on parhaimmillaan silloin, kun suoritettavia vaihtoehtoja on hyvin rajallinen määrä, ja valintaehto jolla valinta suoritetaan, on selkeä ja tarkasti määriteltävissä.

If- ja unless-valintarakenteet loppuehtoina

Eräs erikoisuus, joka Rubysta myös löytyy on se, että If- ja unless-ehtoja voidaan käyttää myös käskylauseen tai rakenteen lopussa lisäehtona. Käytännössä tämä tarkoittaa sitä, että lauseen tai rakenteen lopussa oleva if-valinta lopulta päättää siitä, suoritetaanko lause:

```
debug = 1
puts "Debuggausmoodi" if debug != 0
```

Tulostaa vastauksen, kun taas

```
debug = 0
puts "Debuggausmoodi" if debug != 0
```

ei tulosta mitään. Loppuehtona käytettävä valinta toimii myös suuremman kokonaisuuden kanssa:

```
suorita = true

if suorita == true
    puts "Toimiiko?"
else
    puts "Ei ilmeisesti?"
end if suorita == false
```

Tässä valintarakenteessa `end`in jälkeinen `if suorita == false` aiheuttaa sen, että koko valintarakenne ohitetaan, ellei muuttuja `suorita` saa arvoa `false`. Tämä käytännössä tarkoittaa samalla sitä, että yllä oleva esimerkki ei koskaan tulosta lausetta `"Toimiiko?"`, koska loppuehto ja alkuperäinen valintaehto eivät voi olla yhtä aikaa totta. Loppuehtona käytettävä valinta on joissain tapauksissa hyödyllinen, mutta monesti se lähinnä hankaloittaa koodin ymmärrettävyyttä, joten sitä tulisi ainakin suunnitteluvaiheessa välttää.

LUKU 5

Toistorakenteet ja moduulikirjasto

Toistorakenteet, while, for ja until
Kirjastomoduulit

Tässä luvussa puhutaan toistorakenteista, joiden avulla voidaan suorittaa saman ohjelmakoodin useaan kertaan. Lisäksi luvussa puhutaan toistonohjauksesta, jonka avulla voidaan tarkemmin päättää siitä, missä vaiheessa koodiosion toistaminen lopetetaan. Luvun toisessa osassa esitellään Ruby-kielen mukana toimitettava moduulikirjasto, sekä puhutaan siitä, kuinka kirjaston moduuleja pystytään hyödyntämään.

TOISTORAKENTEET WHILE, FOR JA UNTIL

Jos valintarakenteita käytetään siihen, että valitaan lähdekoodista ne osat, jotka halutaan suorittaa, niin toistorakenteiden tarkoituksena on mahdollistaa lähdekoodin käyttäminen useammin kuin kerran. Jos esimerkiksi ohjelmasssa on tarve suorittaa tiedostosta lukeminen tai rivin tulostaminen useampaan kertaan, niin luonnollisesti tarvitaan rakenne, jolla lukukäskyä – tai tulostusta – voidaan toistaa niin kauan kunnes tarpeellista.

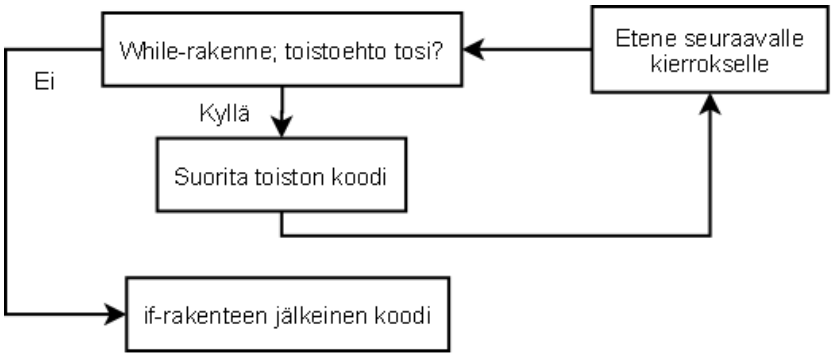
Ruby-kielessä onkin useampi erilainen toistorakenne joita voidaan tarpeen mukaan käyttää niin kuin parhaaksi nähdään. Tavallisempia toistorakenteita edustaa while ja for, jotka toimivat pitkälti kuten samannimiset rakenteet muissa ohjelmointikielissä. Lisäksi Ruby tarjoaa muutaman erikoisemman toistorakenteen kuten until, joka on periaatteessa käänteinen while, tai times, joka on kokonaisluvun metodina käytettävä määräinen toistorakenne.

While-rakenne

Kuten luvun johdannossa jo mainittiin, on while-rakenne ehkä kaikista tavanmukaisin ja perinteisin Ruby-kielen toistorakenteista. While-rakenteeseen määritellään toistoehto, jonka ollessa true, suoritetaan while-rakenteen sisään määritelty koodi uudestaan ja uudestaan. While-rakenne alkaa while-käskyriviltä ja päättyy end-komentoon:

```
while <toistoehto>
  <koodia>
  <koodia>
  <koodia>
end
```

Toistoehto on täysin samanlainen kuin valintaehto valintarakenteissa; se määritellään operandeista ja operaattoreista, jotka on listattu luvussa 4. While-rakenteen koodin suorittamista jatketaan niin kauan, kunnes toistoehto saa arvon false.



While-toistorakenne; toistoa jatketaan kunnes toistoehto saa arvon false tai toistonohjaus katkaisee toiston suorittamisen.

Erityisesti while-rakenteessa kananttaa muistaa, että siinä ei toistoehdolle ole erillistä valvontamekanismia, jolla ohjelma suojautuisi päättymättömältä toistolta, eli tilanteelta jossa toistoehto ei koskaan saa arvoa false. Tämä siis tarkoittaa sitä, että toistoon on rakennettava jonkinlainen kierroslaskuri tai mekanismi, joka keskeyttää toiston kun se on hoitanut tehtävänsä loppuun. Yksinkertaisimmillaan while-rakenne voikin olla seuraavanlainen:

```
01  #määritellään kierrosmäärä
02  kierrosmaara = 5
03
04  while kierrosmaara > 0
05    puts "Nyt on kierros "+kierrosmaara.to_s
06    kierrosmaara = kierrosmaara - 1
07  end
```

Ohjelma tulostaa joka kierroksella kierrosnumeron vastauksena.

```
>ruby 5while.rb
```

```
Nyt on kierros 5
Nyt on kierros 4
Nyt on kierros 3
Nyt on kierros 2
Nyt on kierros 1
>Exit code: 0
```

Toistoehto testataan aina ennen kierroksen aloittamista; tämä tarkoittaa siis sitä, että mikäli toistoehto on jo valmiiksi false, ei toistoa suoriteta kertaakaan, ja suoritus siirtyy toistoa loogisesti seuraavaan rakenteeseen sen kierroksen päätyttyä, jolla toistoehto sai arvon false. Toistoehdon ei kuitenkaan tarvitse olla ennen toiston alkamista määriteltävissä; toistoehdoksi riittää vaikka se, että sille annetaan arvoksi muuttuja, joka on saanut arvon true.

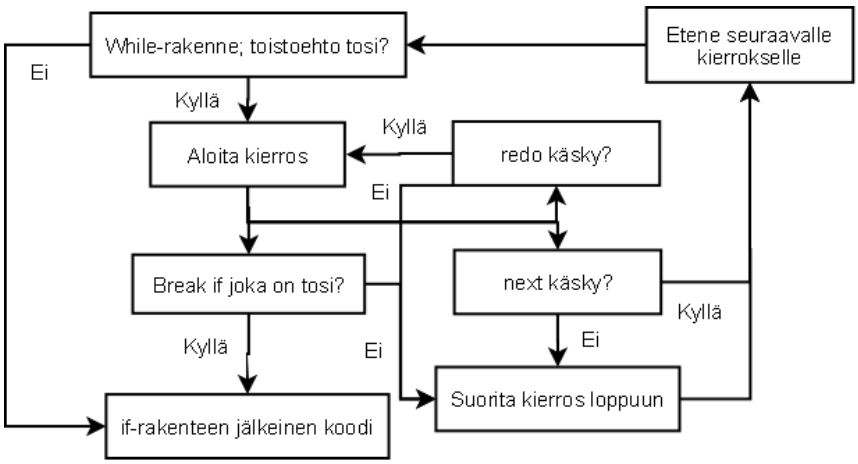
```
01  #Määrätään että toistoa jatketaan toistaiseksi
02  toista = true
03
04  while toista
05    puts "Kirjoita jotain: "
06    luettu = gets
07
08    #jos käyttäjä kirjoittaa lopeta...
09    if luettu.chop! == "lopeta"
10      #...niin ilmoitetaan toiston katkeavan tähän kierrokseen
11      toista = false
12    else
13      puts "Kirjoitit #{luettu}"
14
15    end
16  end
```

Nyt ohjelma jatkaa toimintaansa niin kauan, kunnes käyttäjä antaa sopivan syötteen ja toistoehto saa arvon false:

```
Kirjoita jotain:
testi
Kirjoitit testi
Haloo?
Kirjoita jotain:
lopeta
```

Tämä muoto on kätevä erityisesti silloin, kun etsitään laajasta aineistosta sopivaa syötettä, kuten esimerkiksi tiedostosta oikeaa riviä. Kun haluttu suorite on tapahtunut, voidaan katkaisuehto asettaa käsin arvoon false. Tällätavoin ohjelmoija pystyy tarkemmin määräämään siitä, milloin toistorakenne lopetetaan.

Toistoehdon käsin asettamisen lisäksi Rubyssa on myös kolme erillista toistonohjauskäskyä, joiden avulla voidaan päättää joko koko toiston keskeyttämisestä, käynnissä olevan kierroksen uudelleenyrittämisestä sekä käynnissä olevan kierroksen loppuosan keskeyttämisestä. Nämä käskyt ovat `break if`, `redo` ja `next`.



Break if, redo ja next toistorakenteessa

Toiston katkaisu, break if

Joissain tapauksissa toistorakenne voidaan lopettaa kesken, jos ohjelmassa päästään tilanteeseen jossa toistoa ei enää tarvita. Esimerkiksi luettelon läpikäynti tai oikean tietueen etsiminen tiedostosta voisi olla tällaisia toimintoja; miksi käydä loppuja vaihtoehtoja läpi jos vastaus jo tiedetään? Katkaisu totetetaankin seuraavalla muodolla:

```
break if <katkaisuehto>
```

Jos break if-käskyyn lisätty katkaisuehto saa arvon true, keskeytetään toisto siihen paikkaan ja jatketaan rakennetta seuraavaan käskyyn suorittamatta kierrosta loppuun. Katkaisuehto toimii samalla tavalla kuin aiemmin käsitellyt valinta- tai katkaisuehdot. Käytetään katkaisua vielä toimivassa esimerkissä:

```
01  nuotit = ["do","re","mi","fa","sol","la","si"]
02  i = 0
03  toista = true
04
05  while toista
06    sointu = nuotit[i]
07    break if sointu == "fa"
08    i += 1
09    print sointu+"! "
10  end
11  puts "Loppu!"
```

Tämä ohjelma siis tulostaa seuraavan vastauksen:

```
>ruby breakif.rb
do! re! mi! Loppu!
>Exit code: 0
```

Ohjelma käy läpi taulukkoa nuotit, ja tulostaa ruudulle aina käytettävän nuotin. Toistoon on kuitenkin määrätty katkaisuehto `sointu == "fa"`, eli kun taulukkoa käydään neljänteen alkioon asti, toteutuu katkaisu ja toistorakenne loppuu siihen paikkaan. Tämä voidaan todeta siitä, että ohjelma ei tulostanut enää tekstiä `fa!`, joka olisi tulostettu katkaisuehdon testauksen jälkeen ennen kierroksen päättymistä.

Toiston uudelleensuoritus, redo

Toinen tavallinen toistonohjaukseen liittyvä toiminto on redo, joka aloittaa sisimmän toistorakenteen kierroksen uudelleen. Uudleenaloitukseen ei tarvita muuta kuin käskysana `redo`, eikä siihen liitetä esimerkiksi ehtolausetta tai muutakaan komentoa:

```
for i in 0..10
  if i == 5
```

```
    puts "5!"
    redo
  end
  puts "#{i}!"
end
```

Tämä ohjelma tulostaa seuraavaa:

```
>ruby redoer.rb
0!
1!
2!
3!
4!
5!
5!
5!
5! (ohjelma jää ikuiseen toistoon)
```

Ohjelma siis ei koskaan pysähdy, koska kierroksella `i == 5` ohjelma aina heittää suorituksen kierroksen alkuun. Tässä kannattaa myös huomata, että redo ei testaa toistoehtoa uudelleen, joten uudelleenaloituksen kanssa kannattaa olla tarkkana koska ohjelman saa helposti ikuisen toiston aiheuttamaan lukkoon. Lisäksi esimerkissä ensimmäisen kerran käytettiin `for`-toistoa, johon palataan hetken päästä tarkemmin; `for`-toiston erikoisuus on sen ”sisäänrakennettu” toistomäärän hallinta, mutta sekään ei `redo`:n kanssa toimi. Tätä muotoa ei pidä sotkea komentoon `retry`, joka uudelleenkäynnistää rakenteen edellisen koodiosion alusta ja jonka avulla voidaan esimerkiksi määritellä uudet aloitusarvot. `retry`-käskystä puhutaan lisää virheidenkäsittelyn yhteydessä.

Kierroksen ohitus, next

Kolmas toistorakenteen ohjaamiseen käytettävä avainsana on next, jota voidaan käyttää kierroksen loppuosan ohittamiseen. Kun toistorakenne törmää next-käskysanaan, ohitetaan loput kierroksesta, ja siirrytään suoraan uuteen kierrokseen. Tämä on helpointa havainnollistaa lyhyellä esimerkkiohjelmalla:

```
01  kierros = 0
02
03  while kierros < 4
04      kierros +=1
05      puts "Aloitetaan kierros #{kierros}!"
06      if kierros == 3
07        next
08      end

09      puts "Lopetetaan kierros #{kierros}!"
10    end
```

Tämä ohjelma tulostaa seuraavaa:

```
>ruby next.rb
Aloitetaan kierros 1!
Lopetetaan kierros 1!
Aloitetaan kierros 2!
Lopetetaan kierros 2!
Aloitetaan kierros 3!
Aloitetaan kierros 4!
Lopetetaan kierros 4!
>Exit code: 0
```

Kuten tulosteesta nähdään, ei kierroksella jolla muuttuja `kierros` saa arvon 3 suoriteta jälkimmäistä tulostusta, vaan hypätään suoraan seuraavalle – ja samalla viimeiseksi jäävälle – kierrokselle. Käytännössä next eroaa redo-komennosta kahdella tavalla: ensinnäkin, kierroksen aikana tapahtuneet asiat, kuten kierroslukumuuttujan muuttaminen, pysyvät tallennettuina, sekä toistoehto tarkastetaan aina next-käskyn jälkeen. Tästä seuraakin se, että next ei mene aivan yhtä helposti jumiin kuin redo,

mutta esimerkiksi siirtämällä nyt koodissa rivillä 04 oleva muuttujan vaihto next-käskyn jälkeiseen osaan, saadaan tämäkinn koodi menemään jumiin. Esimerkiksi koodi

```
kierros = 0

while kierros < 4
  puts "Toistolla tehoa!"
  if kierros == 1
    next
  end
  kierros +=1
  end
```

Toistaa tulostettaan `Toistolla tehoa!` äärettömästi, koska toistoehtoa muuttava koodi tulee vasta next-käskyn jälkeen.

For-rakenne

Äskeisissä toistonohjausta koskevissa esimerkeissä jo ohimennen mainittiinkin toinen Ruby-kielestä löytyvä varsin tavallinen toistorakenne joka on nimeltään for. For-toiston tärkein ero while-toistoon on siinä, että for-toisto on rakenteeltaan tavallisesti määräinen, eli toisin sanoen for-toiston alkaessa voidaan tietää kuinka monesti toisto tullaan suorittamaan. Toisaalta taas, for-rakenteen avulla on tehty helpoksi taulukon alkioiden, sekä muiden määräisten rakenteiden läpikäynti. For-rakenne luodaan syntaksilla

```
for <kierrosmuuttuja> in <taulukko, numerojono tai vastaava> do
    <for-toiston suorittaman koodi>
end
```

Yksinkertaisin esimerkki aiheesta voisi olla vaikkapa

```
for i in 1..5 do
  puts i
end
```

joka tulostaa

```
>ruby fortoistot.rb
1
2
3
4
5
>Exit code: 0
```

tai taulukkoa käyttäen

```
taulu = [5,10,15,20]

for alkio in taulu do
  print "#{alkio*100} "
end
```

joka tulostaa alkioiden arvot sadalla kerrottuna: `500 1000 1500 2000`. For-toiston syntaksi kuitenkin vaatinee hieman tarkemman selityksen. Ensinnäkin, kierrosmuuttuja on siis muuttuja, joka kierroksen aikana tallentaa joko kierrosnumeron (jos käytetään lukulistaa) tai käsiteltävän alkion arvon (jos käytetään rakenteista muuttujaa kuten taulukkoa). Tämän ansiosta esimerkiksi taulukkoa läpikäydessä voidaan suoraan viitata alkioon, ja muuttaa tai testata sen arvoa.

For-toistoille tavallinen ominaisuus tosiaan onkin se, että kierrosten määrä on etukäteen tiedossa siinä vaiheessa, kun toiston toteuttaminen aloitetaan. Tietysti for-toiston kierrosmäärää voidaan lisätä myös toiston aikana; Esimerkiksi koodi

```
luvut = [1,2,3,4]
```

```
for i in luvut do
   print i
   luvut.push(i+1)
end
```

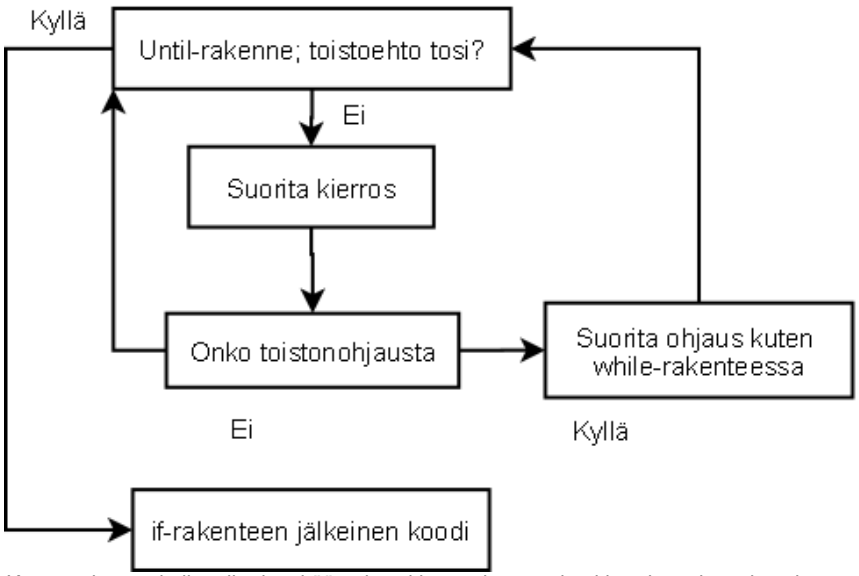
aiheuttaa päättymättön toiston, joka tulostaa

```
1234234534564567567867897891089101191 (jatkuu loputtomasti)
```

koska joka kierros kun for-rakenne etenee `luvut`-taulukossa yhden alkion eteenpäin, lisätään taulukkon loppuun yksi alkio lisää. Täten vaatimus määräisestä toistosta ei teknisesti ajateltuna ole siis ehdoton, mutta sitä voidaan kuitenkin pitää huonona ohjelmointitapana; jos toistosta on tarkoitus tehdä määräämätön, on parempi käyttää while-rakennetta. Toistonohjauskäskyjen käyttämisessä sen sijaan ei ole mitään vikaa; jos kierros pitää aloittaa alusta, voidaan ohittaa tai koko toiston läpikäynti lopettaa, ei ole mitään syytä miksi näin ei voitaisi toimia.

Negatiivinen toisto, Until

Nyt esiteltyjen toistorakenteiden while ja for lisäksi Rubysta löytyy kolmas varsin tavallisesti esiintyvä toistorakenne, joka on negatiivinen toisto `until`. Negatiiviseksi toistorakennetta voidaan sanoa sen takia, että aivan kuten negatiivinen valinta unless, myös until toimii loogisesti päinvastoin kuin muut toistorakenteet.



Kuten unless, toimii until-toisto käänteisesti ja suorittaa uuden kierroksen jos toistoehto on false.

Kun until saa toistoehdon, joka saa arvon `false`, suoritetaan sen sisältämä toistorakenne ja kun se saa arvon `true`, lopetetaan toiminta. Lisäksi until-rakenteen kanssa on mahdollista käyttää toistonohjausta, ja ne noudattavat samoja sääntöjä kuin while-rakenteessa. Until-toiston käyttäminen onkin tavallisinta silloin, kun ohjelman toiminnassa halutaan erityisesti korostaa sitä, että toiston tarkoitus on nimenomaan esimerkiksi pienentää käsiteltävää datajoukkoa tai vähentää jotain arvoa tiettyyn rajaan asti:

Esimerkki 5-1: Negatiivinen toisto, until

Lähdekoodi

```
01  jaljella = 4
02
03  until jaljella < 0
04    puts "Nyt on enää "+jaljella.to_s
05    jaljella -=1
06  end
```

Tuloste

```
>ruby 5until.rb
Nyt on enää 4
Nyt on enää 3
Nyt on enää 2
Nyt on enää 1
Nyt on enää 0
>Exit code: 0
```

Jos tätä esimerkkiä verrataan `while`-toistorakenteeseen, voidaan niiden tosiaan huomata olevan lähes identtiset. Kuitenkin, aivan kuten valintarakenteissa oli rakennepari `if` ja `unless`, on toistoissa käytettävissä `while` ja `until`. Until toimii täysin samalla tavalla kuin while-toistorakenne, mutta edellyttää toiminnaltaan, että toistoehto saa arvon `false` ja lopettaa, kun se saa arvon `true`.

Muita toistorakenteita

Lisäksi Ruby-kielessä on olemassa myös laaja joukko muita erikoisempia tai erikoistapauksiin sopivia toistoja, joista tässä kappaleessa käsitellään muutama. Tavallisin, ja samalla yksinkertaisin käytettävä näistä erikoistapauksista on `times`, joka itse asiassa on kokonaislukuolion metodi. Times-toistorakenne onkin hyvin yksinkertainen. Se käyttää seuraavaa rakennetta:

```
<kokonaisluku>.times do
    <koodi>
end
```

Eli yksinkertaisesti vaikkapa

```
4.times do
    puts "Toistolla tehoa!"
end
```

Tämä ohjelma tulostaa seuraavaa:

```
>ruby timestoisto.rb
Toistolla tehoa!
Toistolla tehoa!
Toistolla tehoa!
Toistolla tehoa!
>
```

Rakenteella onkin helppo tehdä etukäteen määrättyjä toistomääriä. Toisin kuin while- ja for-rakenteiden kanssa, ei times-toiston kierrosten lukumäärää ole mahdollista muuttaa ainakaan vahingossa. Lisäksi kokonaisluku- ja merkkijono-olioilla on metodit `upto` ja `downto`, joita voidaan käyttää toistettavan koodin määrittelyyn:

```
> 5.upto(8) {|i| puts i}
5
6
7
8
=> 5
> 10.downto(7) {|i| puts i}
10
9
8
7
=> 10
> "a".upto("h") do
   print "! "
  end
! ! ! ! ! ! ! ! => "a"
```

```
>
```

Merkkejä käytettäessä toistojen määrä määräytyy merkin ASCII-arvon perusteella. Tämän seurauksena a:sta h:hon suoritettava toisto tapahtuu kahdeksan kertaa, koska kirjaimet ovat taulussa kahdeksan arvon päässä toisistaan.

Kolmas metodi-muotoisesti toimiva toistorakenne on each, jota käytetään tavallisesti tiedostojen tai esimerkiksi verkkosivujen hakemisen yhteydessä. Käytännössä each vaatii toimiakseen rakenteisen tietomuodon tai vaihtoehtoisesti tiedostokahvan, josta lukea tietoa. Each toistaa annetun koodin yhtä monta kertaa kuin rakenteisessa muodossa on alkioita. Taulukon tapauksessa tämä tarkoittaa sitä, kuinka monta arvoa taulukkoon on määritelty; tiedostoissa toisto tapahtuu niin monta kertaa kuin tiedostossa on rivejä.

```
> taulu.each {|arvo| puts arvo.to_s}
1
2
3
4
5
=> [1, 2, 3, 4, 5]
>
```

Otetaan vielä yksi esimerkki erikoisemmista toistorakenteista. Seuraavassa esimerkissä on koottu yhteen `each`-toistorakenteen käyttäminen taulukon kanssa sekä `times`- ja `upto`-toistorakenteet:

Esimerkki 5-2: Erilaisia toistorakenteita

Lähdekoodi

```
01  taulukko = ["eka","toka","kolmas"]
02  taulukko.each {|alkio| puts alkio+" kierros!"}
03
04  3.times do
05      puts "Toistolla tehoa!"
06  end
07
08  "a".upto("d") do
09    puts "Kirjaintoistoja"
10  end
```

Tuloste

```
>ruby timestoisto.rb
eka kierros!
toka kierros!
kolmas kierros!
Toistolla tehoa!
Toistolla tehoa!
Toistolla tehoa!
Kirjaintoistoja
Kirjaintoistoja
Kirjaintoistoja
Kirjaintoistoja
>Exit code: 0
```

Kuten tulosteesta voidaan huomata, rakenteet ovat varsin suoraviivaisia. Taulukon kanssa käytettävä `each` ottaa argumenttina itselleen koodilohkon, jossa määritellään mitä jokaiselle taulun alkiolle tehdään, tai ainakin toistaa annettua lohkoa yhtä monta kertaa kuin taulukossa on alkioita. Kokonaislukujen `times`-metodi on periaatteessa ennaltamäärätty toistorakenne joka suoritetaan aina kokonaisluvun määräämän kertaa. Lisäksi lopussa on `upto`-toisto, joka tällä kertaa on laitettu katkaisuehtonaan laskemaan kirjaimia.

KIRJASTOMODUULIT

Luvun lopuksi puhutaan vielä lyhyesti Ruby-kielen tavasta käyttää moduuleja. Useiden ohjelmointikielten, kuten myös Rubyn, mukana toimitetaan joukko valmiita kirjastoja, joiden avulla on mahdollista käyttöönottaa erilaisia lisätoimintoja. Tästä kirjastosta löytyvät toiminnot mm. päivämäärien, kellonaikojen sekä tietokoneen verkkoyhteyden käyttämiseen ohjelmissa, sekä työkalut rinnakkaisuutta käyttävien ohjelmien tekemiseen.

Moduulikirjaston toiminnot käyttöönotetaan yksinkertaisesti antamalla ohjelman alussa käsky

```
require ”<moduulinnimi>”
```

eli vaikkapa

```
require ”thread”
```

jolloin `thread`-moduulin luokkia voidaan käyttää ohjelman teossa. Huomaa sitaattimerkit moduulinnimessä Esimerkiksi kellonaika saadaan time-nimisestä moduulista, ja siitä voidaan ottaa talteen ohjelman suoritushetken kellonaika helposti:

```
01  require "time"
02
03  kellonaika = Time.new()
04  puts kellonaika
```

Ohjelma tulostaa seuraavaa:

```
>ruby kello.rb
Thu Apr 22 17:51:51 +0300 2010
>Exit code: 0
```

Kellonaika on tallennettuna kiinteänpituiseen tulosteen, joten siitä voidaan parsia halutut tiedot, kuten pelkkä kellonaika yksinkertaisella muutoksella. Lisätään rivit

```
04  kellonaika = kellonaika.to_s
05  puts kellonaika[11..18]
```

joilloin ohjelma tulostaa pelkästään kellonajan:

```
>ruby kello.rb
17:55:56
>Exit code: 0
```

Mielenkiintoisempi kirjastomoduuli kuitenkin on `open-uri`, joka on käyttöliittymämoduuli jolla on mahdollista hakea Internetistä verkkosivuja. Kuten time-moduuli, myös open-uri toimii yksinkertaisesti siten, että moduuli käyttöönotetaan require-komennolla, jonka jälkeen open-käskyllä voidaan tavallisten tiedostojen lisäksi avata myös verkko-osoitteita:

Esimerkki 5-3 Verkkosivun hakeva ohjelma

Lähdekoodi

```
01  require "open-uri"
02
03  koodi = ""
04  sivu = open("http://www.ruby-lang.org/")
05
06  # << lisää muuttujan merkkijonon perään
07  sivu.each {|rivi| koodi << rivi}
08
09  print koodi
```

Tavallisuudesta poiketen tämän esimerkin yhteydessä ei esitetä esimerkkitulostetta, koska muuttujaa koodi on esimerkissä tallentunut koko Ruby-kommuunin verkkosivujen etusivun html-kuvaus. Kirjan sivuilla esitettyinä tämä olisi useita, useita sivuja joten tällä kertaa jätetään asia uskon varaan. Tärkeä huomio kuitenkin on se, että koska verkkosivu saadaan ladattua tavallisen tiedoston tavoin, voidaan moduulin avulla ladata verkosta tietoa, kuten RSS-uutissyötteitä tai muuta tietoa ohjelman käytettäväksi.

Rubyn mukana toimitettavien kirjastomoduulien lista on varsin laaja, joten tässä ei ole mielekästä listata niitä kaikkia. Seuraavaan listaan on kerättynä joukko hyödyllisiä ja mielenkiintoisia kirjastomoduuleja; täydellinen lista ja samalla lisätietoja kyseisistä moduuleista löytyy helpoiten fxri-ohjekirjasta.

Taulukko 5.1 Rubyn keskeisiä kirjastomoduuleja

Nimi	Kuvaus
time	Kellonaika- ja päivämäärämoduuli, sisältää kellonajan ja päivämäärän käsittelyyn liittyviä toimintoja.
thread	Sisältää monisäikeisten (yhtäaikaisesti monia asioita suorittavien) ohjelmien suunnittelun.
open-uri	Sisältää työkaluja verkkoyhteyden, lähinnä http- ja ftp-yhteyksien käyttämiseen.
matrix	Sisältää työkalut matriisi- ja vektorilaskujen tekemiseen.
fileutils	Sisältää työkaluja tiedostojenkäsittelyyn; kopiointi, poisto, uudelleen nimeäminen.
complex	Sisältää työkalut kompleksilukujen käyttämiseen.
socket	Sisältää yleisiä työkaluja verkkoyhteyden käyttämiseen.
date	Lisää työkaluja erityisesti päivämäärien kanssa työskentelyyn.

Luonnollisesti moduuleja voidaan myös tehdä omista luokista ja irtometodeista. Niistä puhutaan enemmän seuraavassa luvussa, jossa läpikäydään omien moduulien ja irtometodien määritteleminen.

LUKU 6

Tiedostot ja metodit

Tiedostojen käyttäminen
Metodien määrittely
Moduulien ja kirjastojen tekeminen

Tässä luvussa käsitellään Ruby-ohjelmointikielen tavat tallentaa tietoa tiedostoihin sekä toimintoja, joita tiedostojenkäsittelyn yhteydessä voidaan toteuttaa. Luvun toisessa osassa puhutaan erillisten metodien, eli eräänlaisten aliohjelmien määrittelystä ja käyttämisestä, sekä niiden kokoamisesta erillisiin moduuleihin.

TIEDOSTOJEN KÄYTTÄMINEN

Luonnollisesti ohjelmia tehdessä tulee jossain vaiheessa myös tarve lukea tietoa erillisestä tiedostosta tai tallentaa esimerkiksi tehdyt asetukset tai muutokset ulkoiseen tiedostoon uudelleenkäyttöä varten. Tällöin Rubyssa voidaan käyttööottaa File-luokka, jonka avulla voidaan tehdä tiedostokahvoja, ja lukea sekä kirjoittaa tiedostoja.

Tiedostokahvan tekeminen

Kuten monessa muussakin ohjelmointikielessä, pitää Rubyssa tiedostonkäsittelyä varten ensin määritellä mihin moodiin tiedosto avataan. Rubyssa tiedostonkäsittelyyn on käytettävissä seuraavat vaihtoehdot:

Taulukko 6.1 Tiedostonavausmoodit

Moodi	Selite
`r`	Pelkkä luku, tiedosto-osoitin laitetaan tiedoston alkuun.
`r+`	Luku ja kirjoitus, tiedosto-osoitin laitetaan tiedoston alkuun.
`w`	Pelkkä kirjoitus, tiedosto-osoitin laitetaan tiedoston alkuun. Jos tiedostoa ei löydy, se luodaan. Aiemmin olemassaollut samanniminen tiedosto tuhotaan.
`w+`	Luku ja kirjoitus, tiedosto-osoitin laitetaan tiedoston alkuun. Jos tiedostoa ei löydy, se luodaan. Aiemmin olemassaollut samanniminen tiedosto tuhotaan.
`a`	Pelkkä kirjoitus, tiedosto-osoitin laitetaan tiedoston loppuun. Jos tiedostoa ei löydy, se luodaan. Mikäli samanniminen tiedosto on jo olemassa, kirjoitusta jatketaan sen loppuun.
`a+`	Luku ja kirjoitus, tiedosto-osoitin laitetaan tiedoston loppuun. Jos tiedostoa ei löydy, se luodaan. Mikäli samanniminen tiedosto on jo olemassa, kirjoitusta jatketaan sen loppuun.

Lisäksi on myös olemassa binääriarvojen kirjoittamseen tarkoitettu moodi `b`, mutta se toimii ainoastaan Windows-käyttöympäristössä joten sen käyttöä kannattaa välttää. Kannattaa myös huomata, että taulukossa puhutaan tiedosto-osoittimesta, joka käytännössä tarkoittaa Rubyn sisäistä kirjanmerkkiä siitä, missä kohtaa tiedostoa ollaan menossa. Tiedosto-osoitin siirtyy aina siihen kohtaan, mihin edellinen tiedostoa käyttänyt käsky loppui. Tällöin kirjoitettaessa tiedostoon kirjoitus jatkuu aina merkistä eteenpäin, jolloin ohjelma ei ylikirjoita aiempaa tekstiä; samoin luettaessa luetaan merkistä eteenpäin, jolloin yhdellä lukukerralla jokainen tiedoston merkki luetaan ainoastaan kerran.

Joka tapauksessa, tiedoston avaaminen ja sulkeminen on Rubyssa helppoa. Oletetaan, että suoritettavan lähdekoodin kanssa samassa hakemistossa sijaitsee tiedosto `teksti.txt`, jossa lukee teksti

```
Tämä teksti on kirjoitettu tiedostoon;
Vesihiisi sihisi hississä.
```

Tiedosto avataan lukemista varten, yksinkertaisesti luomalla uusi File-tyyppinen olio, jolle annetaan argumentteina tiedoston nimi ja käytettävä avausmoodi:

```
tiedosto = File.open("teksti.txt","r")
```

Jolloin luodaan tiedosto-olio `tiedosto`, johon on kytketty kiintolevyllä oleva tiedosto `teksti.txt`. Seuraavaksi voimme lukea tiedoston sisällön, tulostaa sen ruudulle ja sulkea tiedoston käskyillä

```
tiedosto.each {|rivi| print rivi}
tiedosto.close
```

Joka siis tulostaa tiedoston rivit ja lopuksi sulkee tiedoston sulkumetodilla `close`. Lukemiseen ja kirjoittamiseen palataankin hetken päästä, mutta sitä ennen tarkastellaan hieman lähemmin `File`-luokkaa ja sen metodeja.

Tiedostoja käsiteltäessä voidaan siis saada joitain lisätietoja kutsumalla File-luokkaan rakennettuja metodeja. Esimerkiksi metodi `.rename` vaihtaa käytettävän tiedoston nimeä, ja exist? testaa onko kyseisennimistä tiedostoa olemassa. Seuraavaan taulukkoon on lisatattu keskeisimmät File-luokan metodit:

Taulukko 6.2 File-luokan metodit

metodi	Selite
`exists?(<nimi>)`	Testaa onko argumenttina annetun nimistä tiedostoa olemassa. Palauttaa arvon `true` tai `false`.
`file?(<nimi>)`	Testaa, onko annettu nimi tiedosto. Palauttaa `true` tai `false`.
`.readable?(<nimi>)`	Testaa voiko annetunnimistä tiedostoa lukea. Palauttaa `true` tai `false`.
`.writable?(<nimi>)`	Testaa voidaanko tiedostoon kirjoittaa. Palauttaa `true` tai `false`.
`.size(<nimi>)`	Palauttaa annetun tiedoston koon tavuina, eli kertoo kuinka monta merkkiä tiedosto sisältää.
`.zero?(<nimi>)`	Testaa, onko annettu tiedosto tyhjä (kooltaan nolla). Palautta `true` tai `false`.
`.ctime(<nimi>)`	Palauttaa tiedon siitä milloin tiedosto on luotu.
`.mtime(<nimi>)`	Palauttaa tiedon siitä milloin tiedostoa on viimeksi muokattu.
`.rewind`	Siirtää tiedosto-osoittimen takaisin tiedoston alkuun.
`rename(<nimi>,<uusinimi>)`	Muuttaa tiedostonnimen `nimi` pysyvästi arvoon `uusinimi` Palauttaa arvon 0, mikäli nimenvaihto onnistuu.
`delete(<nimi>)`	Tuhoaa nimetyn tiedoston kiintolevyltä. Palauttaa arvon 1 mikäli tuhoaminen onnistuu.

Tiedostojen lukemiseen ja kirjoittamiseen käytettävät tiedostot käsitellään erikseen, koska ne eivät suoranaisesti ole File-luokan, vaan luokasta tehtyjen olioiden metodeja. Niistä puhutaan kuitenkin seuraavaksi.

Tiedoston lukeminen

Tiedoston lukeminen on mahdollista tehdä muutamallakin eri tavalla. Ensinnäkin, lukemista varten tarvitaan tiedosto, jossa on jotain sisältöä. Aiemmin esitellyn esimerkin mukaisesti, avataan siis tiedosto käskyllä

```
tiedosto = File.open("teksti.txt","r")
```

Tiedosto on sama, jota käytettiin aikaisemmassa esimerkissä, ja se siis sisältää kaksi riviä tekstiä. Helpoin tapa lukea tiedostoa on käyttää metodia readline, joka palauttaa tiedostosta yhden rivin, eli tiedoston alusta rivinvaihtoon, rivinvaihdosta rivinvaihtoon tai rivinvaihdosta tiedoston loppuun, yhtenä merkkijonona:

```
>tiedosto.readline
=> Tämä teksti on kirjoitettu tiedostoon;
```

Readlinessa on kuitenkin se ongelma, että se palauttaa kerralla ainoastaan yhden tiedoston. Toinen, vielä ikävämpi ongelma, on lisäksi se, että `readline` palauttaa `EOFError`-virheen, mikäli tiedosto-osoitin on mennyt tiedoston loppuun, joka samalla siis kaataa ohjelman. Tehokkaampi tapa lukea tiedostoja onkin metodi each, joka toistaa sille annetun ohjelmablokin yhtä monta kertaa kuin tiedostossa on rivejä:

```
tiedosto = File.open("teksti.txt","r")
tiedosto.each {|rivi| print rivi}
tiedosto.close
```

Tulostaa siis tekstin

```
Tämä teksti on kirjoitettu tiedostoon;
Vesihiisi sihisi hississä.
```

Ja lopuksi sulkee tiedoston. Metodia each voidaan myös käyttää siihen, että tiedoston sisältö luetaan yhdeksi taulukoksi, jossa jokainen rivi muodostaa oman alkionsa. Tämä onnistuu helposti muotoilemalla argumenttina annettava koodilohko muotoon

```
tiedosto.each {|rivi| lista.push(rivi)}
```

Eli vaikkapa:

```
01  tiedosto = File.open("teksti.txt","r")
02  lista = Array.new
03  tiedosto.each {|rivi| lista.push(rivi)}
04  puts "Luettiin #{lista.length} riviä."
05  puts "Ensimmäinen rivi:" +lista[0]
06  tiedosto.close
```

joka tulostaa vastauksen

```
Luettiin 2 riviä.
Ensimmäinen rivi:Tämä teksti on kirjoitettu tiedostoon;
```

Tiedostoa voidaan myös lukea merkki kerrallaan `getc.chr` –metodilla:

```
tiedosto = File.open("teksti.txt","r")
puts tiedosto.getc.chr
tiedosto.close
```

Joka tulostaisi tiedoston ensimmäisen merkin, eli ison T-kirjaimen. Myös `getc.chr`-metodi kaataa ohjelman, mikä sillä yritetään lukea tiedostoa, jonka tiedosto-osoitin on tiedoston lopussa.

Tiedostoon kirjoittaminen

Tiedostoon kirjoittaminen toteutetaan hieman samankaltaisesti kuin tiedoston lukeaminen. Kirjoittaminen toteutetaan avaamalla tiedosto kirjoittamisen mahdollistavaan moodiin, ja käyttämällä puts-metodia:

```
01  lista = ["eka","toka"]
02  #Kirjoitetaan tiedostoon tavaraa:
03  tiedosto = File.open("tulos.txt","w")
04  tiedosto.puts("Tämä päätyy tiedostoon.")
05  tiedosto.puts(31313)
06  tiedosto.puts(lista)
07  tiedosto.puts("Viimeinen rivi!")
08  tiedosto.close
```

Tuottaa koodin kanssa samaan hakemistoon tiedoston nimeltä `tulos.txt`, joka sisältää tekstin

```
Tämä päätyy tiedostoon.
31313
eka
toka
Viimeinen rivi!
```

Rubyssa tiedostoon voidaan kirjoittaa `puts`-metodilla kutakuinkin mitä tahansa. Esimerkkikoodissa kirjoitettiin tiedostoon merkkijono, taulukko sekä kokonaisluku; voidaan sanoa, että mikä tahansa

tietotyyppi, joka on tulostuva normaalin tulostuskäskyn `to_s`-metodilla, voidaan myös kirjoittaa tiedostoon.

Hakemistossa liikkuminen

Useimmiten ohjelmissa ei kuitenkaan ole mahdollista tai edes järkevää säilyttää kaikkia tiedostoja samassa kansiossa, vaan tallentaa ne esimerkiksi alihakemistoihin. Tällöin on tietenkin tärkeää päästä liikkumaan hakemistosta toiseen luettavaa tiedostoa etsien, tai tarvittaessa pystyä luomaan uusia hakemistoja mikäli kyseessä on ensimmäinen tallennuskerta.

Rubyssa hakemistoissa liikkumiseen käytettävät työkalut löytyvät `Dir`-luokasta. Luokan metodeista löytyy tarpeelliset työkalut kaikkiin hakemistossa liikkumisen ja toimimisen tehtäviin. Esimerkiksi käskyllä `Dir.pwd` Ruby palauttaa sillä hetkellä käytetyn oletushakemiston sijainnin:

```
> Dir.pwd
=> "C:/Ruby/lib/ruby/gems/1.8/gems/fxri-0.3.7"
>
```

Vastaavasti kansion vaihtaminen onnistuu käyttämällä luokan chdir-metodia ja antamalla uuden hakemiston sijainti, esimerkiksi `Dir.chdir("C:/")`. Tämän jälkeen oletuskansio on vaihtanut paikkaansa:

```
> Dir.chdir("C:/")
=> 0
> Dir.pwd
=> "C:/"
```

Tämän lisäksi Dir-luokassa on myös metodi `entries`, joka palauttaa listana kaikki käytössä olevassa oletuskansiossa olevat tiedostot merkkijonotaulukkona:

```
> Dir.entries(".")
=> ["$RECYCLE.BIN", "AMD", "autoexec.bat", "autohook.bat", "BF_Developer", "boot",
"bootmgr", "BOOTSECT.BAK", "cygwin", "Dev-Cpp", "Documents and Settings",
"hiberfil.sys", "hp", "inetpub", "INSTALL.LOG", "Intel", "IO.SYS", "juttu",
"MSDOS.SYS", "MSOCache", "pagefile.sys", "PerfLogs", "Program Files", "ProgramData",
"Railsohjelmat", "Ruby", "swsetup", "System Recovery", "System Volume Information",
"system.sav", "temp", "Users", "Windows"]
>
```

Lisäksi metodilla `mkdir` voidaan tehdä uusia hakemistoja antamalla argumenttina uuden kansion nimi:

```
> Dir.mkdir("C:/testi/")
=> 0
>
```

Tämä luo tässä tapauksessa C-aseman juureen – joka siis oli käytössä ollut hakemisto- alihakemiston nimeltä `testi`. Mihin hakemistotyökaluja sitten voi käyttää? Tiedoston sijainnin haku voidaan esimerkiksi yhdistää näppärästi tiedostojen käsittelyyn. Tiedosto-olio pystyy näet avaamaan minkä tahansa tiedoston johon sillä on lukuoikeus, kunhan sille annetaan täydellinen osoite, eli muodossa

```
<asema>:/<hakemisto>/<alihakemisto>/<tiedostonnimi>
```

Eli vaikkapa

```
tiedosto = File.open("C:/juttu/juttu.txt","r")
```

Jolloin Ruby avaa C-asemalta aseman juuressa olevasta alihakemistosta juttu tiedostoa juttu.txt. Kannattaa tietenkin huomata, että vaikka Windows käyttää luonnollisesti hakemistoerottimena `\`-merkkiä, on sille Rubyssa jo käyttökohde muotoilunohitusmerkkinä, joten `\`-merkki korvataan hakemisto-osoitteessa `/`-merkillä. Vastaavasti, jos tiedetään, että käytettävä tiedosto on lähdekoodin alihakemistossa tiedostot, voidaan siihen viitata muodossa

```
.<alihakemisto>/<tiedostonimi>
```

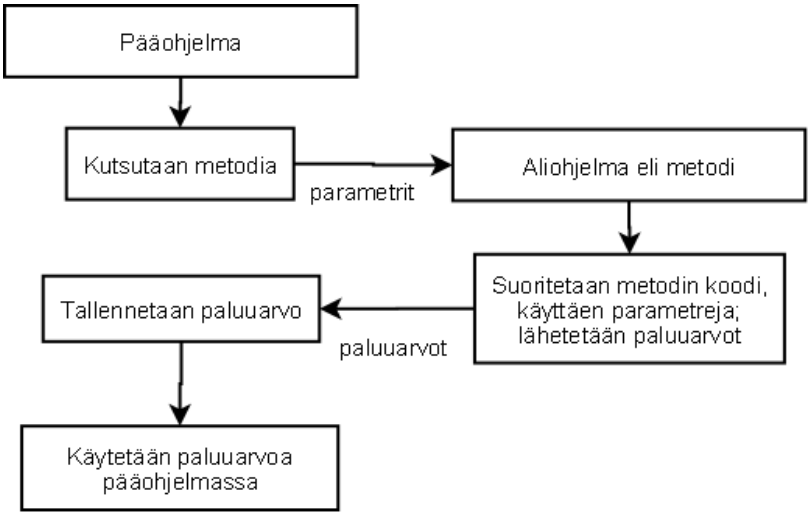
Erityishuomiota kannattaa kiinnittää pisteeseen rivin alussa, joka kertoo käyttöjärjestelmälle ”tämä hakemisto”, ja lisää alkuun automaattisesti suoritettavan lähdekoodin sijainnin:

```
tiedosto = File.open("./tiedostot/juttu.txt","r")
```

Avaa tiedoston normaaliin tapaan, jos oletetaan että suoritettava lähdekoodi-tiedosto on vaikkapa hakemistossa `D:\Rubykoodit\` ja luettava tiedosto `D:\Rubykoodit\tiedostot\`.

METODIEN MÄÄRITTELY

Jos ohjelmassa on joskus käskysarjoja, jotka halutaan tai joudutaan suorittamaan uudestaan ja uudestaan, voidaan Ruby-kielessä määritellä näitä toimintoja varten irtonaisia metodeja. Nämä



toiminnot ovat irrallaan normaaleista olio-rakenteista, eli ne toimivat enemminkin normaalien funktioiden, kuin aitojen metodien tavoin. Näille irtometodeille voidaan antaa argumenttejä, eli syötteitä niitä kutsuttaessa, ja ne palauttavat toiminnan päätyttyä paluuarvon.

Metodin toimintaperiaate

Yleisesti ottaen Ruby-kielessä käytettävä metodi on toiminnallisesti samanlainen kuin useissa muissa ohjelmointikielissä käytetty funktio. Ruby-kielessä puhutaan kuitenkin metodeista, koska metodit ovat toinen nimi olioiden jäsenfunktioille, ja Ruby-kielessä kaikki toiminnot ovat olioita. Myös irtometodit, vaikka ne käytännössä ovatkin irti oliorakenteisesta muodosta verrattuna olioiden sisäisiin metodeihin, ovat Rubyssa `NilClass`-luokan olioita.

Irtometodin määrittely on hyvin yksinkertaista. Ruby-tulkille kerrotaan metodin nimi, sekä se, minkä nimisiä argumentteja metodi ottaa vastaan:

```
def <metodinnimi>(<argumentit>)
```

Eli yksinkertaisimmillaan joko

```
def tulostus
```

tai

```
def laskuri(luku1, luku2)
```

Jos metodissa ei ole argumenttejä, ei sitä varten tarvita edes sulkeita. Lisäksi kannattaa huomata, että `def`-lohko päättyy toisto- ja valintalohkojen tapaan `end`-riviin. Seuraavassa esimerkissä tehdäänkin yksinkertainen metodi:

Esimerkki 6-1: Tulostusmetodi

Lähdekoodi

```
01  #Määritellään metodi
02
03  def tulosta_moi
04    puts "Terve vaan!"
05    end
06
07  #Kutsutaan luotua metodia
08  tulosta_moi
```

Tuloste

```
>ruby 6-metodi.rb
Terve vaan!
>Exit code: 0
```

Kuten esimerkistä voidaan huomata, ei metodin määrittelyä varten tarvita paljoakaan koodia. Lisäksi, kun metodi halutaan suorittaa, sitä voidaan kutsua muualta koodista. Ainoa varsinainen ero irtometodin kutsumisessa on se, että sitä kutsutaan suoraan

```
<metodinnimi>(argumentit)
```

Eikä kuten Rubyssa tavallisemmin

```
<olionnimi>.<metodinnimi>(argumentit)
```

Lisäksi metodeille voidaan tietenkin antaa argumentteja, joiden avulla metodin toimintaa voidaan ohjata. Tästä puhutaankin seuraavaksi.

Metodin argumentit

Ilman kykyä syöttää tietoa metodille niiden käyttömahdollisuudet ovat kuitenkin hyvin rajalliset. Jotta metodeista olisi oikeasti hyötyä, vodiaan niille antaa lisätietoa suoritusta varten argumentteinä. Rubyssa argumentit annetaan metodin määrittelyn yhteydessä, missä kerrotaan minkä nimisiä ja minkä verran argumentteja annetaan. Kuten normaaleiden muuttujien yhteydessä, määrittelyssä ei tarvitse ottaa kantaa siihen mitä argumentteihin itse asiassa tallennetaan. Argumentit määritelään metodien kutsussa muodossa

```
def <metodinnimi>( <argumentinnimi>, <argumentinnimi> ...)
```

Eli vaikkapa

```
def sensuroija(teksti, sanalista)
```

tai

```
def kerrokeskenaan(eka, toka, kolmas)
```

Ensimmäisessä esimerkissä metodi sensuroija saa kaksi argumenttia, teksti ja sanalista; vastaavasti kerrokeskenaan saa kolme argumenttia. Argumenttejä voidaankin määritellä metodiin niin monta kuin katsotaan tarpeellisesti. Lisäksi metodin saamat argumentit toimivat metodin sisällä argumentin nimellä kuten muuttujat. Esimerkiksi kerrokeskenaan-esimerkkimetodin argumentit eka, toka ja kolmas toimisivat metodin sisällä suoraan muuttujina:

```
def kerrokeskenaan(eka,toka,kolmas)
  tulos = eka*toka*kolmas
  puts tulos.to_s
  end
```

Tämän vuoksi tulkki myös varmistaa aina sen, että jokaiselle argumentille löytyy jokin kelvollinen arvo. Kannattaa tosin pitää mielessä, että johtuen siitä ettei Ruby-tulkki ota mitään kantaa argumenttien tyyppeihin, on metodin itse tarkastettava, että argumentit saavat kelvollisen arvon. Tämän vuoksi esimerkiksi yllä oleva koodi kannattaisi kirjoittaa enemmin muotoon

```
def kerrokeskenaan(eka,toka,kolmas)
  # merkkijonon muuttaminen to_f-metodilla tuottaa arvon 0.0
  tulos = eka.to_f * toka.to_f * kolmas.to_f
  puts tulos.to_s
  end
```

siltä varalta, että metodille päätyy argumenttina. Käytännössä kaikki mitä argumenttien tekemiseen vaaditaan onkin lyhyesti seuraavassa esimerkissä:

Esimerkki 6-2: Metodi jossa on argumentti

Lähdekoodi

```
01  #Määritellään metodi jossa on parametri
02  def sanonimi(nimi)
03    puts "Terve #{nimi}!"
04   end
05
06   #Kutsutaan luotua metodia muutaman kerran
07   sanonimi("Jussi")
08   sanonimi("Jani")
```

Tuloste

```
>ruby 6-parametrillinen.rb
Terve Jussi!
Terve Jani!
>Exit code: 0
```

Koodi ei muutu paljoakaan ensimmäisestä esimerkistä, vaan metodin määrittelyyn ainaostaan lisätään tieto siitä, kuinka monta ja minkä nimisiä parametrejä kutsussa pitää määritellä. Jos esimerkkikoodin metodille `sanonimi` olisi jätetty parametri `nimi` antamatta, olisi kutsu oolut virheellinen, ja aiheuttanut Rubyn virheilmoituksen ArgumentError, joka näyttää kutakuinkin tältä:

```
6-parametrillinen.rb:7:in `sanonimi': wrong number of arguments (0 for 1)
(ArgumentError)
            from 6-parametrillinen.rb:7
```

Jossa kerrotaan missä kohtaa ohjelma suoritti virheen, ja mikä ohjelmassa oli pielessä. Tällä kertaa kyseessä olisi siis ollut rivillä 7 ollut kutsu `sanonimi,` jolle ei annettu argumenttia.

Argumentteja ei kuitenkaan tarvitse aina määritellä jokaista erikseen. Jossain tapauksissa metodissa voi olla ylimääräisiä argumentteja, joiden määrittely voi olla tarpeellista ainoastaan poikkeustapauksissa.

```
def laskuri(luku1 = 1, luku2 = 2)
  tulos = luku1 + luku2
  puts "Laskun summa on #{tulos}"
end
```

Kutsutaan luotua metodia kolmesti seuraavilla argumenteilla:

```
laskuri
laskuri(10)
laskuri(10,1024)
```

Ruby antaa seuraavanlaisen vastauksen:

```
Laskun summa on 3
Laskun summa on 12
Laskun summa on 1034
```

Koska metodissa on kaikille argumenteille määritelty oletusarvo, tarkoittaa se sitä että ne voidaan määritellä metodia kutsuttaessa vapaasti. Ensimmäisellä kerralla ei anneta yhtäkään argumenttia, ja

laskutoimitus suoritetaan tällöin oletusarvoilla. Toisella kerralla määritellään ensimmäinen, ja kolmannella kerralla molemmat argumentit, koodin toimiessa joka tapauksessa normaalisti. Erikoislaatuisena lisäominaisuutena kannattaa myös huomata, että metodikutsussa sulkumerkit ovat pitkälti luetunymmärtämisen tukena. Metodikutsu

```
laskuri 25, 25
```

tuottaa täysin samankaltaisen vastauksen kuin sulkujen kanssa kirjoitettu:

```
Laskun summa on 50
```

kuin kutsu, johon sulkumerkit on laitettu. Argumenttien lisäksi metodit tarvitsevat myös tavan antaa vastauksia. Tähän metodit voivat käyttää paluuarvoa, joka on tämän osion kolmas ja viimeinen irallisia metodeja käsittelevä osio.

Metodin paluuarvo

Jotta metodeista saadaan oikeasti hyödyllisiä, on niille annettava sisääntuloarvojen, eli argumenttien, lisäksi myös lopetusarvo, jota kutsutaan nimellä paluuarvo. Käytännössä tämä tarkoittaa sitä, että metodi kertoo toimintansa pääteeksi sitä kutsuneelle ohjelmalle lopputuloksen, oli se sitten laskutoimituksen tulos, tieto siitä onko pyydetty tehtävä onnistunut tai mitä suoritettu testi antoi tuloksena.

Tavallisin esimerkki paluuarvosta on nimenomaan metodien testauksessa käytetty true ja false. Esimerkiksi kaikissa muuttuja-olioissa on metodi `.empty?`, joka palauttaa paluuarvon true, jos muuttuja on tyhjä, ja false, jos muuttujalle on määritelty jokin arvo. Paluuarvo määritellään koodilla

```
return <palautettava muuttuja>
```

Eli yhdistettynä laajempaan metodikokonaisuuteen vaikkapa seuravalla tavalla:

```
def pienempikuinsata?(luku = 0)
  if luku < 100
    tulos = true
  else
    tulos = false
  end
  return tulos
end
```

Kutsutaan metodia muutamalla eri argumentilla, ja tulostetaan saamamme vastaus:

```
puts pienempikuinsata?(115)
puts pienempikuinsata?(75)
```

tulostaa vastauksen

```
false
true
```

Eli metodi pienempikuinsata? kertoo sitä kutsuneelle ohjelmalle minkä tuloksen se sille annetun parametrin avulla saa aikaiseksi. Tietenkin, jotta ohjelmasta on jatkossa mitään hyötyä, voidaan metodin tuottama vastaus ottaa talteen muuttujaan syntaksilla

```
<muuttuja> = <metodinnimi>(<argumentit>)
```

Eli tässä tapauksessa vaikkapa

```
vastaus = pienempikuinsata?(1024)
```

jolloin ohjelman ajon jälkeen muuttujaan vastaus on tallentunut metodin pienempikuinsata? paluuarvona antama vastaus:

```
puts vastaus
```

```
=> false
```

Metodeja ja paluuarvoja käyttäessä kannattaa kuitenkin huomata eräs Ruby-kieleen liittyvä erikoisuus; metodilla on aina paluuarvo, vaikka sitä ei erikseen määriteltäisi. Tämä voi joskus johtaa epämääräiseltä vaikuttaviin ongelmiin, mutta käytännössä paluuarvon määräytyminen on varsin simppeliä. Määritellään seuraavaksi metodi, jolla ei ole erikseen määritettyä paluuarvoa:

```
01  def kasittelija(sana = "Inkvisitio", luku = 0)
02    if luku < 0
03      tapahtuma = sana
04    else
05      tapahtuma = luku
06    end
07  end
```

Kuten koodista voidaan huomata, ei siinä tällä kertaa ole itsestään selvää paluuarvoa. Kutsutaan metodia seuraavaksi muutamaan otteeseen erilaisilla parametreilla ja tallennetaan saatu vastaus:

```
tulos1 = kasittelija("Boa", -10)
tulos2 = kasittelija("Kobra", 3)
```

Toisin kuin voisi olettaa, ei muuttujien `tulos1` ja `tulos2` tulostuskäsky aiheuta virhettä, vaan niille on määritelty selkeät arvot:

```
puts tulos1.to_s
=>Boa
puts tulos2.to_s
=>3
```

Miksi ohejelma toimii näin? Syy siihen on siinä, että Ruby antaa aina paluuarvoksi viimeisimmäksi määritellyn muuttujan arvon, mikäli metodille ei muuten ole määritelty oikeaa paluuarvoa. Koska ensimmäisessä esimerkissä muuttujaan `tapahtuma` tallennetaan argumentti `sana`, jää se viimeiseksi määritellyksi muuttujan arvoksi. Vastaavasti alemmassa esimerkissä suoritetaan luvun tallennus muuttujaan `tapahtuma`, jolloin automaattisesti luoduksi paluuarvoksi jää argumentti `luku`, jota käytettiin viimeisessä muuttujan arvon määrittelyssä. Jos metodia muutetaan siten, että rivit 06-09 menevät seuraavasti

```
06    end
07  juttu = ”kahvin maukuinen ihovoide”
08  end
```

jää kyseinen käsky metodin viimeiseksi muuttujamäärittelyksi, ja antaa kyseisen tuloksen aina automaattisena paluuarvona:

```
tulos1 = kasittelija("Boa", -10)
tulos2 = kasittelija("Kobra", 3)
puts tulos1.to_s, tulos2.to_s
```

Tulostaa nyt vastauksen

```
>ruby metodeja.rb
kahvin makuinen ihovoide
kahvin makuinen ihovoide
>Exit code: 0
```

Ei liene ole vaikea hahmottaa, että automaattinen paluuarvon generointi on hyvin epämääräinen tapa toimia, ja sen vuoksi paluuarvo tulee aina määritellä erikseen return-määritteellä, jos on olemassa minkäänlaista mahdollisuutta että metodin paluuarvoa voidaan käyttää jossain.

MODUULIEN JA KIRJASTOJEN TEKEMINEN

Tähän asti on luvussa puhuttu siitä, kuinka käskyjä voidaan niputtaa erillisiksi irtometodeiksi. Seuraava luonnollinen siirtymä onkin se, kuinka joukko irtometodeja voidaan koota kirjastoksi, joka voidaan siirtää ohjelmasta toiseen ja ottaa käyttöön samaan tapaan kuin Ruby-ohjelmointikielen omat kirjastot, joista puhuttiin edellisessä luvussa.

Yksinkertaisin tapa koota irtometodit yhdeksi kirjastoksi on koota ne kaikki yhteen erilliseen lähdekooditiedostoon, ja tämän jälkeen `require`-komennolla käyttöönottaa tämä lähdekooditiedosto. Oletetaan, että pääohjelman kanssa samassa hakemistossa on tiedosto ”omatmetodit.rb”, jonka sisältö on seuraavanlainen:

```
01  def tulostaja (teksti = "", toistot = 1)
02    puts teksti * toistot
03  end
04
05  def testaaja (arvo)
06    if arvo < 10
07      return true
08    else
09      return false
10    end
11  end
```

Tiedosto ei siis sisällä mitään muuta kuin joukon irtometodeja. Jotta nämä metodit saadaan käyttöön pääohjelmassa, riittää kun Rubylle annetaan määräys käyttöönottaa tiedosto käskyllä `require ”<tiedostonnimi>”`, tiedoston nimi siis ilman loppuosaa `.rb` ja sitaattimerkeissä:.

```
01  require "omatmetodit"
02
03  tulostaja("Kvaak!",4)
04  vastaus = testaaja(11)
05  puts vastaus
```

Kun ohjelma suoritetaan, toimivat omatmetodit-tiedostosta tuodut metodit kuin ne olisi määritelty pääohjelmassa itsessään:

```
>ruby kutsuja.rb
Kvaak!Kvaak!Kvaak!Kvaak!
false
>Exit code: 0
```

Samaan tapaan esimerkiksi luokkien määrittelyt voidaan viedä erilliseen tiedostoon, joskin tästä puhutaan vasta myöhemmin luvussa 8. Menetelmä käyttää require-komentoa tuomaan metodit ohjelmaan ei kuitenkaan vielä ole kovin turvallinen tai toimintavarma mekanismi. Entä jos tiedoistoissa onkin päällekkäisiä nimiä, esimerkiksi kahdessa eri tiedostossa on samanniminen metodi `testaasyote`? Tätä vastaan voidaan varautua määrittelemällä tiedostoihin moduuleja, eräänlaisia kokoelmia, jotka käyttöönotetaan erikseen komennolla `include`.

Moduuli on uusi koodilohkon muodostava kokonaisuus, joka luodaan seuraavasti:

```
module <nimi>
  <koodia>
  <koodia>
  <koodia>
end
```

Moduulin erikoisuus on siinä, että moduulin sisälle voidaan määritellä irtometodeja, sekä esimerkiksi muuttujia, joita on tarkoitus käyttää osana koodia. Otetaan esimerkiksi moduuli, jossa on esimerkkimetodi ja muutama muuttuja:

```
01  module Sisusarvot
02    def sanonimi
03      puts "Olen Sisusarvot!"
04    end
05    Omaarvo = "sisusarvo"
06  end
```

Huomaa, että nimeämissäännöt vaativat, että moduulin muuttujien nimen pitää alkaa isolla alkukirjaimella jotta ne toimivat oikein moduulin ulkopuolella.

Jotta moduulin muuttujat ja metodit voidaan ottaa käyttöön, pitää koodiin määritellä include <moduulinimi>, eli tässä tapauksessa `include Sisusarvot`. Tehdään nyt hieman lisää koodia, jolla testataan moduulin toimintaa:

```
01  include Sisusarvot
02  Sisusarvot::sanonimi
03  Omaarvo = "pääkoodi"
04  puts Omaarvo  #tämä on pääohjelman omaarvo
05  puts Sisusarvot::Omaarvo #tämä tulee moduulista
```

Tämä koodi tulostaa vastauksen

```
>ruby moduulit.rb
Olen Sisusarvot!
pääkoodi
sisusarvo
>Exit code: 0
```

Ensinnäkin, moduulin sisällä oleviin muuttujiin sekä metodeihin viitataan muodossa

```
<moduulinnimi>::<moduulinsisäinen nimi>
```

Eli vaikkapa koodin `Sisusarvot::Omaarvo`. Lisäksi, kuten esimerkistä voidaan huomata, ei tässä yhteydessä kumpikaan muuttuja Omaarvo mene sekaisin, vaikka pääohjelmassa onkin samanniminen muuttuja, voidaan moduulien sisään huoletta määritellä samanniminen muuttuja, koska siihen viitataan muodossa `<moduulinnimi>::<muuttujannimi>`.

Muiden tiedostojen kanssa työskennellessä asia ei muutu paljoakaan, ainoastaan siinä, että ensin require-käskyllä tuodaan tiedosto ohjelmaan mukaan, ja sitten include-käskyllä käyttöönotetaan moduuli. Ensimmäisen esimerkin tapauksessa se tarkoittaisi sitä, että tiedosto `omatmetodit.rb` muuttuu seuraavaan muotoon:

```
01  module OmaModuuli
02    def tulostaja (teksti = "", toistot = 1)
03      puts teksti * toistot
04    end
05
06    def testaaja (arvo)
07      if arvo < 10
08        return true
09        else
10          return false
11        end
12     end
13  end
```

ja sitä käyttävä tiedosto kutsuja.rb muotoon

```
01  require "omatmetodit"
02  include OmaModuuli
03  OmaModuuli::tulostaja("Vuh!",4)
04  vastaus = OmaModuuli::testaaja(11)
05  puts vastaus
```

jolloin ohjelma toimii edelleen samalla tavalla kuin aikaisemmin:

```
>ruby kutsuja.rb
Vuh!Vuh!Vuh!Vuh!
false
>Exit code: 0
```

Tärkein ominaisuus ja etu moduuleissa onkin se, että ne luovat uuden nimiavaruuden, eli mahdollistavat sen, että paikallisella tasolla ei tarvitse huolehtia siitä, onko muuttujan, luokan tai metodin nimi käytetty jo jossain muualla. Moduuleihin pakatut nimet onkin erittäin suosittuja mm. Ruby on Rails-sovellusten puolella.

LUKU 7

Virheidenkäsittely ja poikkeukset

Virheiden kiinniotto, begin ja rescue
Jälkien siistiminen ja uudelleenyrittäminen

Tässä luvussa esitellään erilaisia keinoja varautua virhetilanteisiin sekä virheen sattuessa keinoja toipua niistä. Luvussa esitellään myös omien virheilmoitusten määrittelykeinoja, sekä tapoja joilla virheiden tapahduttua yritetään toipua ongelmatilanteesta tai voidaan pakottaa ohjelma lopettamaan toimintansa.

VIRHEIDEN KIINNIOTTO, BEGIN JA RESCUE

Ohjelmoidessa on tavallista tehdä virheitä, tai päätyä tilanteisiin joissa ohjelman suoritusta ei voida jatkaa. Joskus ongelma ei välttämättä edes johdu omasta ohjelmasta, vaan jokin toinen palvelu estää jonkin resurssin, kuten tiedoston, käyttämisen. Joskus voi myös käydä niin, että esimerkiksi heikkolaatuinen verkkoyhteys tai rikkinäinen muistikortti aiheuttaa virhetilanteen. Virheiden kiinniotto ja käsittely ei itse asiassa ole kovin epätavallista; jos voidaan sanoa, että huippuohjelmoija tuottaa 99.9% virheetöntä koodia, tarkoittaa se siitäkin huolimatta sitä, että ohjelman joka tuhannes rivi sisältää virheen. Kun ohjelmassa on miljoona riviä koodia, on ohjelmassa tällöin tuhat virhettä, huolimatta siitä että ohjelman teki ohjelmointiammattilainen.

Täysin virheetöntä koodia ei enää pienen mittakaavan (alle 500 riviä) yli päästyä ole oikein mahdollista, tai edes mielekästä, toteuttaa koska kyseisen koodin testaaminen ja kaikkien toiminnallisten vaihtoehtojen tarkastaminen veisi kohtuuttomasti aikaa ja rahaa. Lisäksi kuten aiemmin mainittu, ei ongelma aina edes aiheudu nimenomaisesti omasta lähdekoodista vaan ympäristöstä, missä ohjelma operoi. Tämän vuoksi virhetilainteita varten ohjelmassa tarvitaan poikeukksien käsittelyyn työkalut virheiden ennakointiin ja virhetilojen ohittamiseen.

Virheiden käsittely

Ruby-kielessä virheiden ennakointiin käytetään kaksi-osaista begin-rescue-rakennetta. Käytännössä rakenne toimii siten, että begin-osioon kirjoitetaan se lähdekoodi, jonka arvellaan aiheuttavan virheen, ja rescue-osioon se koodi, joka suoritetaan jos virhe tapahtuu:

```
begin
  <suoritettava koodi>
  <suoritettava koodi>
rescue
  <koodi joka ajetaan jos tulee virhe>
  <koodi joka ajetaan jos tulee virhe>
end
```

Käytännössä rakenne toimii siis siten, että jos begin-osiossa tapahtuu virhe, suoritetaan rescue-osio johon on kirjoitettu ns. pakoreitti virhetilanteesta pois. Yksinkertaisin, ja varmasti tavallisin, paikka missä rakennetta on luonnollista käyttää on esimerkiksi tiedoston avaaminen lukemista varten:

Esimerkki 7-1: Virheen kiinniottaminen

Lähdekoodi

```
01  puts "Anna luettavan tiedoston nimi:"
02  nimi = gets
03  nimi.chomp!
04
05  begin
06    tiedosto = File.new(nimi,"r")
07    tiedosto.each_line{ |rivi| puts rivi}
08    tiedosto.close
09
10  rescue
11    puts "Jotain meni pieleen!"
12  end
```

Tuloste

Koska meillä on jälleen ohjelmassa syötteen pyyntö `gets`, suoritetaan ohjelma ajamalla se Windowsin ikkunassa:

```
Anna luettavan tiedoston nimi:
teksti.txt
Tämä teksti on luettu tiedostosta!
Vesihiisi sihisi hississä.
```

Suoritetaan ohjelma uudestaan, mutta annetaan tällä kertaa virheellinen tiedostonnimi:

```
Anna luettavan tiedoston nimi!
apina?
Jotain meni pieleen!
```

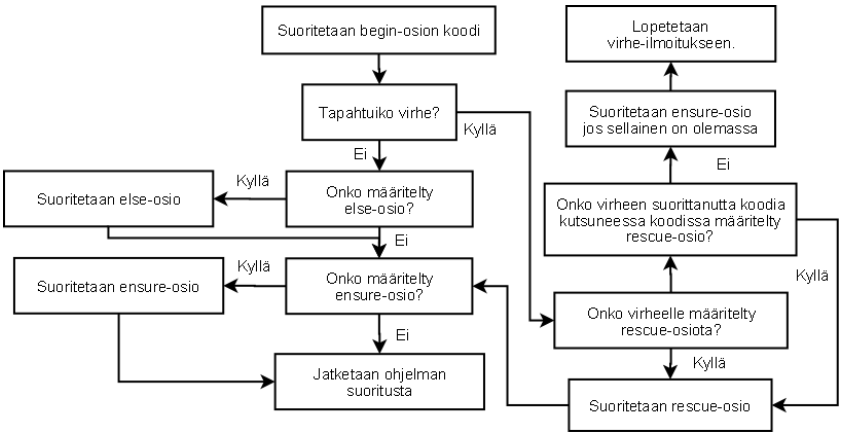
Kuten tulosteesta on mahdollista huomata, siirtyy Ruby begin-osiosta rescue-osioon, mikäli ohjelmassa tapahtuu virhe begin-osion aikana. Rescue-osioita voidaan lisäksi määritellä useita; useimmiten tilanne vaatii sitä, että erilaisiin virheisiin suhtaudutaan eri tavoilla. Jos esimerkiksi tehdään ohjelma, jossa tiedostosta luetaan kaksi arvoa, jaetaan ensimmäinen toisella ja tulostetaan vastaus, voidaan koodista löytää ainakin kolme kohtaa, joihin tulee puuttua. Ensinnäkin, tiedosto, joka suoritusta varten avataan, voi olla käytöltään estetty tai puuttua kokonaan. Toiseksi, tiedostosta luettu arvo ei välttämättä ole numero, jolloin laskutoimitus epäonnistuu, ja kolmanneksi, jos toinen luku on nolla, suorittaa laskutoimitus virheen. periaatteessa ongelma voidaan ratkaista seuraavanlaisella rakenteella:

```
begin
  <tähän ohjelmakoodi>
rescue IOError
  <tähän koodi joka ajetaan jos tiedosto aiheuttaa virheen.>
rescue TypeError
  <tähän koodi joka ajetaan jos arvot on väärät.>
rescue ZeroDivisionError
  <tähän koodi joka ajetaan jos toinen luku oli 0.>
rescue
  <tähän tulisi koodi joka ajetaan muiden virheiden sattuessa>
end
```

Käytännössä rescue-osioita voidaan laittaa peräkkäin niin monta kuin ohjelman toiminnan kannalta on tarpeellista. Lisäksi, mikäli virheen aiheuttava koodi on aliohjelmassa joka ei sisällä omaa virheenkäsittelijää, etsitään sopivaa käsittelijää ohjelmaa kutsuneesta osasta:

```
01  def jakaanollalla
02    tulos = 10 / 0
03  end
04  #virheenkäsittelijä:
05  begin
06    jakaanollalla
07  rescue ZeroDivisionError
08    puts "Virhe napattu!"
09  end
```

Esimerkiksi tässä ohjelmassa virheenkorjaus onnistuu, koska tulkki löytää `ZeroDivisionError`-käsittelijän virheen aiheuttanutta metodia kutsuneesta koodista. Jos tulkki ei kuitenkaan löydä virheenkäsittelijää, annetaan normaali virheilmoitus ja ohjelma lopetetaan. Lisäksi virheenkäsittelijään voidaan lisätä else- ja ensure-osioilla muitakin työvaiheita:



Rescue-rakenne kokonaisuudessaan; virheenkäsittelyssä on monta vaihetta joissa voidaan antaa erilaisia käskyjä.

Seuraavassa on lisäksi lista kaikista Ruby-kielen tavallisista, ohjelman toiminnan seurauksena syntyvistä virhetilanteista.

Taulukko 7.1 Rubyn tavalliset virhetyypit	
Poikkeuksen nimi	Selite
StandardError	Ohjelman toiminnan aiheuttamien virheiden perusluokka, rescue-osion oletusarvo joka käsittää kaikki muut listan virheluokat.
ArgumentError	Ohjelma kutsui metodia virheellisellä määrällä argumenttejä jättäen jonkin arvon määrittelemättä tai lisäten mukaan ylimääräisiä argumentteja.
IndexError	Ohjelma viittasi taulukon alkioon jota ei ole olemassa tai yritti leikkausta joka ei ole mahdollinen.
IOError (EOFError)	Ohjelma yrittää avata tiedoston, jota ei ole olemassa.
LocalJumpError	Ohjelma yrittää suorittaa virheellisen siirtymän koodilohkosta toiseen.
NameError (NoMethodError)	Ohjelma yrittää kutsua sen nimistä muuttujaa tai metodia jollaista ei ole olemassa.
RangeError	Ohjelma yrittää käyttää lukulistaa, jossa on virheellisiä arvoja.
RegexpError	Ohjelmassa on säännöllinen lauseke (regular expression) joka on virheellinen.
RuntimeError	Ohjelmassa on päädytty itse laukaistuun virhetilaan raise-käskyllä.
SystemCallError	Ohjelman yrittää kutsua käyttöjärjestelmän toimintoa tai palvelua, joka palauttaa virheen.
ThreadError	Ohjelman rinnakkaisprosessien kanssa on tapahtunut virhe, kuten esimerkiksi yritys yhdistää jokin osaprosessi itseensä.
TypeError	Ohjelma yrittää käyttää muuttujaa, jonka sisältö ei sovellu suoritettavaan tehtävään, kuten kertoa merkkijonoja keskenään.
ZeroDivisionError	Ohjelma yrittää jakaa nollalla
SystemExit	Ohjelma lopetti toimintansa exit()-komennolla.
SystemStackError	Ohjelma yrittää esimerkiksi rekursiivista operaatiota tehdessään luoda liian pitkän kutsupinon.

Kannattaa huomata, että StandardError poikkeaa muista virhetyypeistä siitä, että se on virheiden ”oletustyyppi”, ja kaikki muut virheilmoitukset ovat sen erikoistapauksia. Jos Ruby-ohjelmaan kirjoitetaan rakenne

```
rescue
```

on se itse asiassa täysin sama asia kuin

```
rescue StandardError
```

eli `StandardError`-tyyppiä ei koskaan tarvitse erikseen nimetä vaan se otetaan kiinni oletuksena jos rescue-käskylle ei anneta mitään muuta arvoa.

Tavallisimpina poikkeuksina, joita Rubyssa tulee vastaan, voidaan sanoa olevan `RuntimeError`, joka tulee siinä tapauksessa että ohjelma itse käynnistää poikkeuksen `raise`-käskyllä, `NoMethodError`, `NameError`, `IOError`, `TypeError` ja `ArgumentError`. Koska kaikki näistä virheluokista kuitenkin ovat `StandardError`-luokan aliluokkia, tarkoittaa se sitä, että pelkkä `rescue` ilman tarkempaa määrittelyä pystyy käsittelemään ne kaikki oletusarvoisesti.

StandardError ei kuitenkaan pysty ottamaan kiinni kaikkia virhetyyppejä, vaikkakin se sisältää kaikki tavallisimmat. Oheisessa taulukossa on vielä listattuna ne virhetyypit, joiden käsittelijä pitää määritellä aina erikseen:

Taulukko 7.2 Rubyn erikoisvirheluokat

Poikkeuksen nimi	Selite
fatal	Rubyn sisäinen virhe; Ruby-tulkki tai sitä käyttävä järjestelmä on vioittunut ja vaatii luultavasti uudelleenasennuksen.
NoMemoryError	Ohjelma epäonnistui muistinvarauksessa.
ScriptError	Ohjelma yrittää suorittaa virheellisen komentosarjan.
LoadError	Ohjelma yrittää ladata moduulin tai tiedoston jota ei löydy.
NotImplementedError	Ohjelma yrittää käyttää toimintoa tai ominaisuutta, joka on suunniteltu osaksi järjestelmää mutta ei vielä toteutettu.
SyntaxError	Tulkki ei ymmärrä koodissa olevaa rakennetta.
SecurityError	Ohjelma yrittää suorittaa toiminnon, jota Ruby pitää nykyisillä turvallisuusasetuksilla vaarallisena.

Tässä taulukossa olevat virheluokat ovat virheitä, joita eivät perustu StandardError-luokkaan, joten niihin ei pelkkä rescue-osio ilman tarkempaa määrittelyä pysty vaikuttamaan. Virheluokista erityisesti `LoadError`, joka tulee kun Ruby yrittää ladata noduulin jota ei löydy, on kohtuullisen tavallinen, joskin tavallisesti korjattavissa tarkastamalla että moduulin nimi on kirjoitettu oikein. Muut virhetyypit ovat enempi tai vähempi epätavallisia, ja useimmiten merkki siitä, että käytetty ohjelmointialusta toimii virheellisesti.

Viimeinen perus-virheenkäsittelijöihin liittyvä asia on else-osion. Virheenkäsittelijään voidaan liittää else-osio, joka suoritetaan siinä tapauksessa, että yhtäkään virhettä ei havaittu:

```
begin
  <suoritettava koodi>
rescue
  <koodi joka ajetaan jos tulee virhe>
else
  <koodi joka ajetaan jos virheitä ei tule>
end
```

Else-osio on rakenteellisesti täysin samanlainen kuin esimerkiksi if- tai while-rakenteen else-osiot. Else siis suoritetaan siinä tapauksessa, ettei yhtäkään rescue-toimintoa tarvittu. Toinen vastaava rakenne on ensure, joka suoritetaan aina, aiheutti ohjelma virheen tai ei, ja siitä puhutaankin seuraavaksi.

Pakotettu lopetus, exit

Jos ohjelma suorittaa virheen, on täysin luonnollista että ohjelma lopetetaan. Toisaalta taas, joskus voidaan myös tulla tilanteeseen jossa ohjelma ei välttämättä toimi väärin, mutta toiminnan kannalta on hyödyntöntä jatkaa työskentelyä. Tavallisin esimerkki tästä voisi esimerkiksi olla komentorivityökalu, jolle annetaan virheellinen määrä parametrejä tai arvot joissa ei ole mitään mieltä; tällöin helpointa on antaa lopetuskäsky `exit`.

Kaikessa lyhykäisyydessään exit() lopettaa ohjelman toiminnan siihen riviin, jolla käsky annetaan. Komennolle voidaan lisäksi antaa argumenttina lopetusarvo, joka on eräänlainen tieto siitä mihin ohjelma päättyi. Komentona exit() käyttäytyy samoin kuin tulostuskäsky puts, eli se ei ole olio metodi vaan erillinen komento:

```
01  puts "Tähän pysähtyy!"
02  exit(100)
03  puts "Tätä ei enää suoriteta..."
```

Koodinpätkä tulostaa siis vastauksen

```
>ruby exitkoe.rb
Tähän pysähtyy!
>Exit code: 100
```

Kuten esimerkistä nähdään, pysähtyy ohjelma tähän paikkaan ilman sen erikoisempia seuraamuksia kuten virheilmoituksia. Teknisesti `exit()` kuitenkin nostaa `SystemExit`-tyyppisen virheilmoituksen, joten jos ohjelmaan tulee ehdoton tarve tehdä vielä jälkien siistimistä tai vastaavaa exit-komennon jälkeen, on se mahdollista toteuttaa tällätavoin:

```
01  begin
02    puts "Tähän pysähtyy!"
03    exit(100)
04  rescue SystemExit
05    puts "Ohitetaan exit()"
06  end
```

Tulostaa siis

```
>ruby exitkoe.rb
Tähän pysähtyy!
Ohitetaan exit()
>Exit code: 0
```

Tästä ominaisuudesta on hyötyä esimerkiksi silloin, jos käytetään ulkoisesta moduulista otettua metodia, johon on kirjoitettu `exit`-komento, vaikka pääohjelma voisi jatkaa toimintaansa virheellisista arvoista huolimatta.

JÄLKIEN SIISTIMINEN JA UUDELLEENYRITTÄMINEN

Virheidenkäsittelyn yhteydessä on siis tilanteita, joissa ohjelman voidaan haluta suorittavan jonkin toiminnon huolimatta siitä, tapahtuiko ohjelman ajossa virhettä vai ei. Käytännössä tämä tarkoittaa tilanteita, joissa ohjelman käyttämä toiminto halutaan lopettaa joka tapauksessa. Esimerkkinä voidaan vaikka ajatella tiedoston sulkemista tapahtumalokin kirjoittamista varten; huolimatta siitä, löytyykö lokitietoja tai ei, voidaan tiedosto laittaa kiinni siinä vaiheessa kun kirjoitusta on ainakin yritetty.

Tapahtuman varmistaminen, ensure

Ensure-osio siis suoritetaan aina, huolimatta siitä onko siihen kytketty virheenkäsittely tapahtunut tai ei. Tämä koskee myös tapauksia, joissa virheenkäsittelyssä on else-rakenne; ensure-rakenne tulee aina viimeiseksi virheenkäsittelijän rakenteeksi:

```
begin
  <suoritettava koodi>
rescue
  <koodi joka ajetaan jos tulee virhe>
else
  <koodi joka ajetaan jos virheitä ei tule>
ensure
  <koodi joka suoritetaan joka tapauksessa>
end
```

Uudelleenyrittäminen

Ensure on hyödyllinen siinä tapauksessa, että ohjelmassa halutaan toteuttaa jokin toiminto huolimatta siitä, tapahtuiko virhe tai ei. Mutta entäpä siinä tapauksessa, että ohjelma pystyy korjaamaan ongelman ja halutaan yrittää toimintoa uudelleen? Yksi ratkaisu olisi tietenkin tehdä virheenkäsittelyn sisältävästä toiminnosta metodi, ja paluuarvoa tarkastelemalla päättää suoritetaanko koodi uudelleen vai ei, mutta Ruby tarjoaa myös helpomman tavan toteuttaa uudelleenyritys, ja se on käsky retry.

Toistorakenteiden yhteydessä puhuttiin käskystä redo, joka suoritti saman kierroksen uudelleen aloittaen kierroksen alusta. Virheenkäsittelijöiden yhteydessä vastaava komento on retry, joka ei redon tapaan ota argumentteja tai muitakaan operaattoreita. Jos ohjelman suoritus tulee retry-komentoon, palaa suoritus takaisin virheenkäsittelijän alkuun.

Mihin tällaista rakennetta sitten kannattaa käyttää? Periaatteessa rakenteissa, joissa on mahdollisuus useampaan erilaiseen virheeseen joihin voidaan puuttua suorituksen yhteydessä. Esimerkiksi nollalla jako tai avattavan tiedoston nimen pyytäminen on varsin yksinkertaisia tapauksia;

pyydetään käyttäjältä syötettä virheenkäsittelijän sisällä, ja mikäli tapahtuma ei onnistu, palautetaan suoritus takaisin alkuun:

Esimerkki 7-2 Retry virheenkäsittelijässä

Lähdekoodi

```
01  begin
02    puts "Anna avattavan tiedoston nimi:"
03    nimi = gets
04    #poistetaan rivinvaihtomerkki
05    nimi.chomp!
06
07    #Itse tiedoston lukeminen
08    sisus = ""
09    tiedosto = File.open(nimi,"r")
10    tiedosto.each {|rivi| sisus << rivi }
11
12  rescue
13    puts "Tiedostoa ei löydy!"
14    retry
15
16  else
17    puts sisus
18    tiedosto.close
19  end
```

Tuloste

```
Anna avattavan tiedoston nimi:
apina?
Tiedostoa ei löydy!
Anna avattavan tiedoston nimi:
teksti.tct
Tiedostoa ei löydy!
Anna avattavan tiedoston nimi:
testi.txt
Tiedoston eka rivi!
Toka rivi!
Viimeinen rivi!
```

Esimerkkitulosteessa käyttäjän kirjoittamat syötteet on merkitty lihavoituina. Kyseisessä esimerkkikoodissa ohjelma pyytää retry-käskyn avulla tiedoston nimeä uudelleen niin kauan, että saa nimen joka on oikeasti olemassa. Kun tiedoston nimi vihdoin menee oikein, lukee ohjelma tiedoston sisällön muuttujaan, tulostaa sisällön ruudulle ja sulkee käytetyn tiedoston.

Virheen aiheuttaminen käsin

Viimeinen virheidenkäsittelyyn liittyvä konsepti on virheen aiheuttaminen käsin. Jos ohjelmassa päädytään tilanteeseen, jossa ohjelma ei suoranaisesti ole kaatunut, mutta esimerkiksi päätynyt tilanteeseen mistä se ei voi enää edetä, on myös mahdollista toteuttaa ohjelman lopettaminen käsin. Tämä tehdään raise-komennolla, joka toimii seuraavalla tavalla:

```
raise <virhetyyppi>, ”<virheteksti>”
```

Eli vaikkapa

```
raise ArgumentError, ”Virheelliset argumentit”
```

joka tuottaa normaalin Rubyn virheilmoituksen:

```
>raise ArgumentError, "Virheelliset argumentit"
```

```
ArgumentError: Virheelliset argumentit
            from (irb):6
            from :0
>
```

raise-komennosta voidaan myös jättää virhetyyppi pois, jolloin virhetyyppinä käytetänä RuntimeError-tyyppiä. Jos raise-käsky annetaan esimerkiksi kutsutussa irtometodissa, voidaan sitä kutsuneeseen koodiin rakentaa sopiva virheenkäsittelytoiminto, eli kutsutussa metodissa annettu raise jää kiinni ylemmän tason virheenkäsittelijään: Virheenkäsittelijöiden näkökulmasta `raise`-komennolla tuotetut virheet ovat täysin samassa asemassa kuin ”normalisti” tapahtuneet virheet.

LUKU 8

Luokat ja omien olioiden luominen

Luokka ja olio
Luokkien perintä
Luokkamuuttujat

Tässä luvussa puhutaan lisää jo aiemmin käsitellystä asiasta, olioista. Luvussa tutustutaan tapoihin määritellä omia olioita luomalla ensin luokkia, josta olio pystytään tekemään. Luokkien yhteydessä puhutaan niiden tavallisimmista ominaisuuksista, attribuuteista, metodeista ja luokan perinnästä, sekä luokan metodien näkyvyyksistä eri olioihin.

LUOKKA JA OLIO

Tähän mennessä on kirjassa puhuttu useaan otteeseen luokista ja luokista muodostettavista olioista, mutta emme vielä toistaiseksi ole paljoakaan käsitelleet sitä, kuinka luokkia itseasiassa määritellään. Luokkahan on periaatteessa eräänlainen tietorakenne, joka sisältää itsenäisiä muuttujia sekä irtometodeja, ja niistä luodaan olioita, jotka toteuttavat luokissa määriteltyjä toimintoja. Uuden olion muodostaminen on tehty ennenkin esimerkiksi taulukon määrittelyn yhteydessä metodilla new, mutta kuinka itse luokka sitten tehdään? Tämä tapahtuu Ruby-ohjelmointikielessä `class`-rakenteen avulla. Yksinkertaisimmillaan luokka voidaan määritellä' ilmoittamalla luotavan luokan nimi, ja lopettamalla rakenne `end`-käskyllä:

```
class UusiLuokka
  end
```

Tietenkään tämä määritelty luokka ei tee mitään hyödyllistä, koska se ei sisällä mitään ominaisuuksia tai toimintoja, eli jäsenmuuttujia tai jäsenfuktioita, eli metodeja.

Tavallisin asia, mitä luokkaan kirjoitetaan, on rakentaja (*constructor*), jonka avulla oliolle voidaan joko asettaa alkuarvoja, tai esimerkiksi tehdä ilmoitus siitä, että olio on luotu. Tämä tehdään lisäämällä olioon metodi initialize, johon kirjoitettu koodiosio suoritetaan aina, kun luokasta tehdään uusi olio. Tehdään uusi esimerkkiluokka:

```
class Robotti
  def initialize
    puts "Valmiina toimimaan."
    end
  end
```

Kirjoitetaan luokan lisäksi komento, joka luo uuden olion nyt määritellystä luokasta:

```
palvelija = Robotti.new()
```

Suoritettaessa käskyn tulkki suorittaa Robotti-luokan metodin initialize, ja tulostaa tekstin

```
>ruby 8-luokka.rb
Valmiina toimimaan.
>Exit code: 0
```

Tästä luokasta käyttäjä voi jo päätellä, että jotain todellakin tapahtui, kun ohjelmassa määriteltiin uusi olio. Normaalisti määrittelyvaiheessa voidaan alustaa olion jäsenmuuttujat, ladata olion käyttämät tiedot tiedostot, tarkastaa esimerkiksi riippuvuussuhteet muihin olioihin tai tarvittaessa luoda tarvittavat lisäoliot.

Joissain olio-kielissä on rakentajan lisäksi purkaja (*destructor*), jota kutsutaan automaattisesti silloin kun olio tuhotaan. Tämä ei ole Rubyssa tarpeellista, koska kielessä on tehokas automaattinen muistinhallinta, joka hoitaa purkutehtävät itsenäisesti.

Luokan jäsenmuuttujat, attribuutit

Lisäksi olisi tietenkin hyödyllistä, mikäli oliolle voisi antaa tietoja. Tätä varten luokkiin voidaankin määritellä jäsenmuuttujia, jotka ovat käytössä ainoastaan olion itsensä sisällä. Nämä luodaan aivan kuten normaalitkin muuttujat, joskin sillä erotuksella, että niiden nimi aloitetaan @-etumerkillä.

```
01  class Robotti
02    def initialize(nimi, ammatti)
03      @omanimi = nimi
04      @omaammatti = ammatti
05      puts "Valmiina toimintaan!"
06    end
07  end
```

Koska muuttujat `@omanimi` ja `@omaammatti` ovat olion sisäisiä, omia muuttujia, ei niitä pystytä käyttämään muualta kuin olion sisältä. Ylläolevassa esimerkissä luokkaan on määritelty rakentaja, joka vastaanottaa oliota luotaessa annetut argumentit.

Rakentajassa saadut argumentit sijoitetaan olion omiin sisäisiin muuttujiin, jolloin niitä voidaan käyttää olion myöhemmissä toiminnoissa. Ruby-kielessä ei olioiden jäsenmuuttujia, attribuutteja, ole mahdollista muuttaa mistään muualta kuin olion itsensä sisältä, joten muuttamiseen tarvitaan aina metodi. Tähän voidaan käyttää kahta pienoismetodia, joita voidaan kutsua vaikka palauttaja- ja asettajametodeiksi (*getter* ja *setter* joissain muissa kirjoissa). Otetaan vaikka luokka, jossa on jäsenmuuttuja muisti, joka alustetaan oliota luotaessa arvoon `nil`:

```
class Varasto
  def initialize(alkuarvo)
    @muisti = alkuarvo
  end
end
```

Seuraavaksi tehdään luokkaan asetus- ja palautusmetodit, joiden avulla voidaan muokata olion jäsenmuuttujaa sen ulkopuolelta. Periaatteessa palauttaja-metodin ainoa tehtävä palauttaa luokan jäsenmuuttujan arvo, ja se on voidaan siis määritellä vaikkapa

```
def muisti
  return @muisti
end
```

Asettajametodi voidaan määritellä itse asiassa samannimiseksi kuin palauttaja. mutta siihen tehdään pieni muutos; määritellään metodin nimen loppuun yhtäsuuruusmerkki, joka kertoo kyseessä olevan sijoitusmetodi:

```
def muisti= (arvo)
  @muisti = arvo
end
```

Nyt luokkamääritelmä on siis muodossa

```
class Varasto
  def initialize(alkuarvo = "nil")
    @muisti = alkuarvo
  end
  def muisti
    return @muisti
  end
  def muisti= ( arvo )
    @muisti = arvo
  end
end
```

Seuraavaksi luokasta voidaan määritellä uusi olio, ja kokeilla määriteltyjen sijoittaja- ja astettajametodien toimintaa:

```
siilo = Varasto.new()
siilo.muisti = "Piilossa"
arvo = siilo.muisti
puts arvo
```

Tulostaa vastauksen

```
>ruby luokat.rb
Piilossa
>Exit code: 0
```

Eli sijoitus- ja palautusmetodien avulla jäsenmuuttuja saadaan kutakuinkin toimimaan normaalin muuttujan kaltaisesti. Tätä toimintoa varten on Rubyssa kuitenkin olemassa nopeampi tapa totetuttaa

samat ominaisuudet. Käskyllä `attr_accessor` voidaan nopeasti ja helposti määrätä, että luokan jäsenmuuttujalla on asettaja- ja palauttajametodit. Esimerkiksi luokka

```
class Testaus
  def initialize(eka = nil, toka = nil)
    @eka = eka
    @toka = toka
  end
  attr_accessor :eka
  attr_accessor :toka
end
```

määrittelee, että kummallakin jäsenmuuttujalla `eka` ja `toka` on asettaja- ja palauttajametodit:

```
kokeilu = Testaus.new
kokeilu.eka = 10
kokeilu.toka = kokeilu.eka + 15
puts kokeilu.toka
```

tuottaa vastauksen

```
25
```

Jos metodille halutaan antaa pelkkä asettaja, on siihen käsky `attr_writer`, ja pelkälle palauttajalle `attr_reader`. Lisäksi luokan jäsenmuuttujia ei ole rajattu pelkästään niihin, jotka annetaan argumentteina:

```
01  class Sailio
02    def initialize
03      @sisus = "tyhjä!"
04    end
05
06    def tutki
07      puts "Säiliö on #{@sisus}"
08    end
09  end
10
11  varasto = Sailio.new()
12  varasto.tutki
```

tulostaa tekstin

```
>ruby luokkarakenne.rb
Säiliö on tyhjä!
>
```

Esimerkin luokassa ei rakentaja vastaanota argumentteja, vaan ainoastaan määrittelee jäsenmuuttujan sisus, jonka arvoksi laitetaan `tyhjä!`. Luokan jäsenmuuttujia voidaan tehdä niin monta kuin tarpeelliseksi katsotaan, ja niille annetaan oletusarvot rakentajassa. Luokassa myös määriteltiin metodi `tutki`, joka tulostaa jäsenmuuttujan sisällön, mutta ei suoranaisesti tapaa millä arvoa voidaan muuttaa.

Luokan metodit

Vaikka jäsenmuuttujat itsessään ovat jo tehokas työkalu, on niiden käytössä hieman rajoitteita, johtuen siitä ettei jäsenmuuttujiin päästä käsiksi ilman metodeja. Aikaisemmin käsiteltiin tähän ongelmaan auttavat asetus- ja palautusmetodit, mutta suoranaisesti luokan metodien käyttötarkoitus on huomattavasti laajempi, niiden olisi tarkoitus itse asiassa toteuttaa oliolta odotetut toiminnallisuudet. Luokan metodit määritellään samalla syntaksilla kuin irtometodit, ainoana eronaan se, että metodi kirjoitetaan luokkamääritelmän sisälle. Jos esimerkiksi aiemmin käytetty luokan määrittely muutetaan seuraavaan muotoon

```
01  class Robotti
02    def initialize(nimi, ammatti)
```

```
03      @omanimi = nimi
04      @omaammatti = ammatti
05      puts "Valmiina toimintaan!"
06    end
07
08    def kerroitsestasi
09      puts "Olen #{@omanimi} ja ammatiltani #{@omaammatti}."
10    end
11  end
```

ja koodiin lisätään seuraavat rivit

```
palvelija = Robotti.new("Kipupiste", "Teurastaja")
palvelija.kerroitsestasi()
```

tulostaa Ruby seuraavanlaisen tuloksen:

```
>
Valmiina toimintaan!
Olen Kipupiste ja ammatiltani Teurastaja.
>
```

Tässä esimerkissä luokalla Robotti on siis jo ensimmäinen oma normaali metodi, kerroitsestasi, joka tulostaa luokan jäsenmuuttujien arvot. Luokan sisällä metodit pääsevät vapaasti käyttämään olionsisäisiä jäsenmuuttujia ilman ylimääräisiä asettaja- ja palauttaja metodeja. Seuraavassa esimerkissä tarkastellaan luokan määrittelyä ja käyttämistä vielä lähemmin.

Esimerkki 8-1: Luokan määritteleminen ja käyttäminen

Lähdekoodi

```
01  class Arvioija
02    def initialize()
03       @tallenne = ""
04       end
05    attr_accessor :tallenne
06
07    def tyhja?
08       if tallenne == ""
09         puts true
10       else
11         puts false
12       end
13    end
14
15    def poista
16       puts "Poistetaan: #{@tallenne}"
17       tallenne = ""
18    end
19  end
20  tagi = Arvioija.new
21  tagi.tallenne = "Sata salamaa"
22  tagi.tyhja?
23  tagi.poista
```

Tuloste

```
>ruby luokkametodeilla.rb
false
Poistetaan: Sata salamaa
>Exit code: 0
```

LUOKAN PERINTÄ

Olioiden hyödyllisyyden kannalta tärkeä ominaisuus on lisäksi mahdollisuus periä jokin aiempi luokka, ja määritellä ainoastaan uuden luokan tarvitseman muuttuneet arvot. Otetaan vaikkapa aiemmin käytetty `Sailio`-luokka, jossa on jäsenmuuttuja ja muutama metodi, sekä uusi toiminto `laita`, jolla säiliöön voidaan sijoittaa tavaraa:

```
01  class Sailio
02
03    def initialize
04      @sisus = "tyhjää!"
05    end
06
07    def tutki
08      puts "Säiliössä on #{@sisus}"
09    end
10
11    def laita(arvo)
12      @sisus = arvo
13    end
14  end
```

Oletetaan, että haluamme laajentaa luokan toimintaa siten, että jokaiselle luodulle oliolle annetaan myös omistajan nimi, joka tulostetaan metodilla `omistaja`. Kuitenkin johtuen siitä, että luokkaa Sailio on voitu käyttää jossain muualla koodissa, ei luokan määrittelyä kuitenkaan voida muutaa, varsinkaan rakentajan osalta. Siksi luokka voidaankin periä luokalle Parempisailio:

```
01  class Parempisailio < Sailio
02    def initialize (omistajannimi)
03      @sisus = "tyhjää"
04      @omistaa = omistajannimi
05    end
06
07    def omistaja
08      puts "Säiliön omistaa #{@omistaa}."
09    end
10  end
```

Kun tätä luokkaa käytetään ohjelmakoodissa, on siihen sisältynyt uudet, Parempisailio-luokan ominaisuudet sekä alkuperäisen Sailio-luokan ominaisuudet. Tämä voidaan todeta kirjoittamalla seuraavanlainen ohjelmakoodi:

```
varasto = Parempisailio.new("Schröndinger")
varasto.laita("kissa.")
varasto.tutki
varasto.omistaja
```

Huolimatta siitä, että uuden luokan määrittelyssä ei ole mainittu sanallakaan metodeja `laita` tai `tutki`, toimii ohjelma edelleen oikein:

```
>ruby luokkarakenne.rb
Säiliössä on kissa.
Säiliön omistaa Schröndinger.
>
```

Olioiden perintä toimii hyvin suoraviivaisesti; uutta luokkaa määritellessä laitetaan uuden luokan perään <-merkki osoittamaan sitä, että luokkaan tuodaan ominaisuuksia toisesta luokasta, sekä luonnollisesti perittävän luokan nimi. Luokasta periytyy kaikki luokan metodit ja ominaisuudet, poislukien ne jotka ovat ylikirjoitettu samannimisellä metodilla uudessa luokassa.

Tässä esimerkissä tietenkin oletettiin, että luokkien `Sailio` ja `ParempiSailio` määritelmät on kirjoitettu samaan tiedostoon. Koska tämä ei aina ole mahdollista, tai edes järkevää, voidaan

luokkamääritelmät tuoda ulkoisesta moduulista, aivan kuten irtometodien tapauksessa. Jos oletetaan, että luokka `Sailio` on määritelty tiedostossa *omialuokkia.rb*, riittää kun ohjelman alkuun laitetaan koodi

```
require 'omialuokkia'
```

tai mikäli halutaan olla varma että mitään ei mene pieleen, niin varmennetaan tiedoston tuonti virheenkäsittelijällä:

```
begin
  require 'omialuokkia'
rescue LoadError
  raise "Moduulia omialuokkia ei ole!"
end
```

Metodien näkyvyys

Joissain tapauksissa olioita luotaessa tulee kuitenkin tilanteita, joissa oli hyödyllistä pystyä rajoittamaan sitä, mistä kaikkialta olion metodeja voidaan kutsua. Tähän käytetään näkyvyysmäärittelyjä `public`, `protected` ja `private`. Kun jokin näistä avainsanoista esiintyy luokan määrittelyssä, tarkoittaa se sitä, että kyseisestä kohdasta eteenpäin kaikki määritellyt metodit saavat avainsanan määräämän näkyvyyden..

Public

Public on näkyvyysluokista tavallisin, eikä sitä tarvitse määritellä erikseen, vaan on oletuksena käytössä aina kun mitään muuta näkyvyyttä ei ole määritelty. Tähän asti kirjan esimerkeissä on siis toisinsanoen käytetty poikkeuksetta public-näkyvyyteen määriteltyjä metodeja. Public-metodeja voidaan kutsua mistä tahansa muualta suoritettavasta ohjelmasta, ja ne voidaan luokkamäärittelyssä erikseen määritellä public-näkyvyyteen avainsanalla `public`:

```
class Omaluokka
  public
  def metodi
    puts "Public toimii!"
  end
end¨
```

Tässä yhteydessä avainsana ei siis tee mitään erikoista, mutta jos esimerkiksi jonkin luokan metodin näkyvyyttä on myöhemmin päätetty rajoittaa, voidaan avainsanalla `public` palauttaa loput metodit julkisiksi.

Private

Private on kaikista rajoittavin näkyvyysluokka, ja se estää metodin käyttämisen olion ulkopuolelta. Käytännössä tämä siis tarkoittaa sitä, että private-metodit ovat suojattu samalla tapaan kuin ilman asettajaa tai palauttajaa olevat jäsenmuuttujat. Private-metodien käyttäminen onkin älykästä esimerkiksi silloin, kun halutaan varmistaa, että jotain tiettyä toimintoa käytetään ainoastaan haluttua reittiä. Otetaan esimerkiksi seuraavanlainen luokka:

```
01  class Sekaluokka
02    #Oletuksena metodit on aina public
03    def toimi
04      puts "Julkinen metodi"
05    end
06    private  #Tästä eteenpäin metodit on privateja
07    def yksityinen
08      puts "Yksityinen!"
09    end
10  end
```

Tässä luokassa on kaksi metodia, julkinen `toimi`, sekä yksityinen `yksityinen` Jos luokasta tehdään olio, ja kutsutaan sen metodeja, voidaan huomata näkyvyyksien toiminta:

```
Olio = Sekaluokka.new
```

```
Olio.toimi
Olio.yksityinen
```

Ohjelma tulostaa virheen:

```
>ruby nakyvyys.rb
nakyvyys.rb:25: private method `yksityinen' called for #<Sekaluokka:0x35eeb1c>
(NoMethodError)
Julkinen metodi
>Exit code: 1
```

Jos taas luokkaan lisätään public-metodi kutsu, joka on muotoa

```
def kutsu
   yksityinen
end
```

Ja käytetään ohjelmaa uudestaan:

```
Olio = Sekaluokka.new
Olio.toimi
Olio.kutsu
```

Saadaan tällä kertaa tulos

```
>ruby nakyvyys.rb
Julkinen metodi
Yksityinen!
>Exit code: 0
```

Koska private-metodin kutsu tuli olion itsensä sisältä, joka on ainoa sallittu reitti käyttää private-näkyvyydelle määriteltyjä metodeja. Private-näkyvyyden asettamiseen on lisäksi olemassa erillinen käsky `private :<metodinnimi>`, samaan tapaan kuin asettaja- ja palauttajametodien määrittelyssä. Tätä muotoa käyttämällä ainoastaan nimetyn metodin näkyvyydeksi asetetaan private. Tätä tapaa on hyvä käyttää esimerkiksi silloin, kun halutaan muuttaa yksi metodi private:ksi suurikokoisen oliomäärittelyn keskeltä.

Protected

Kolmas näkyvyysluokka Rubyssa on protected. Tämä näkyvyysluokka on hieman vaikeampi esitellä, koska sen toiminta on pitkälti ohjelman rakenteesta riippuvainen. Lyhyesti selitettynä, protected-metodeja voi käyttää keskenään ne oliot, jotka ovat muodostettu samasta luokasta tai luokan perineestä luokasta. Määritellään kokeilumielessä kolme luokkaa, Suojattu, Oma ja Uusisuojattu, joka perii luokan Suojattu:

```
01  class Suojattu
02    def tervehdi
03      suojattu_moi
04    end
05    protected
06    def suojattu_moi
07      puts "Suojattu tervehdys!"
08    end
09  end
10  class Uusisuojattu < Suojattu
11    def koita(olio)
12      olio.suojattu_moi
13    end
14  end
15  class Oma
16    def koita(olio)
17      olio.suojattu_moi
```

```
18    end
19  end
```

Luodaan näistä luokista olioita, ja tarkastellaan niiden toimintaa:

```
suojaolio = Suojattu.new
uusiolio = Uusisuojattu.new
omaolio = Oma.new
```

Nyt ohjelmassa on kolme oliota. `suojaolio` sisältää protected-metodin `suojattu_moi`, joka tulostaa ruudulle tervehdyksen, sekä `uusiolio`, joka on samasta luokasta perityn luokan `Uusisuojattu` olio. Lisäksi kolmantena on täysin erillinen olio `omaolio`, joka on peritty luokasta `Oma`. Luokissa `Oma` ja `Uusisuojattu` on molemmissa identtinen metodi `koita`, joka vastaanottaa argumenttina olion, ja yrittää käyttää sen `suojattu_moi`-metodia. Käytetään olioita seuraavilla kutsuilla:

```
suojaolio.tervehdi
uusiolio.koita(suojaolio)
omaolio.koita(suojaolio)
```

Tuloksena on se, että tavoista ensimmäinen ja toinen toimivat, mutta kolmas ei. Olio `suojaolio` voi tietenkin kutsua itsensä sisällä tervehtimismetodia. Samoin `uusiolio` voi käyttää `suojaolio`:n protected-metodeja, koska se on samaa ”sukua”. Viimeisessä kohdassa ohjelma ajautuu virheeseen; koska `omaolio` ei ole samaa, eikä edes perittyä luokkaa `suojaolio`:sta, ei se pysty millään menetelmällä hyödyntämään sen protected-metodeja.

LUOKKIEN MUUTTUJAT

Viimeinen olioihin liittyvä aihe tässä luvussa on luokkien omien muuttujien tekeminen. Aikaisemmin kirjassa on puhuttu siitä, kuinka oliot määritellään luokista; jokainen olio saa omat muuttujat ja arvot. Tämä on tietenkin olio-mallien perusta, mutta olioiden saamien omien muuttujien lisäksi voidaan myös itse luokalle tehdä omia muuttujia. Näitä luokkamuuttujiksi sanottuja muuttujia voidaankin käyttää esimerkiksi silloin, kun halutaan tallentaa jotain itse luokkaan liittyvää tietoa, kuten viimeisimmän luokasta luodun olion tietoja, tai luokan kaikkien olioiden yhteiseen käyttöön tarkoitettua tietoa. Periaatteessa luokkamuuttujia voidaankin ajatella eräänlaisina olioiden yhteisinä muuttujina.

Luokkamuuttuja määritellään käyttämällä nimen edessä kahda @-merkki, eli vaikkapa `@@maara` tai `@@nimi`. Nämä muuttujat määritellään luokan määrittelyssä, ja niitä käytetään samalla tavalla kuin olioiden omia muuttujia. Otetaan esimerkiksi seuraava luokkamääritelmä:

```
01  class Asiakas
02    @@raja = 150
03    @@nimi = ""
04    def initialize(nimi)
05      @nimi = nimi
06      @@nimi = nimi
07    end
08    def saldoraja
09      puts "Asiakkaan #{@nimi} saldoraja on #{@@raja}."
10    end
11    def korota
12      puts "Nostetaan kaikkien saldorajaa."
13      @@raja += 100
14    end
15    def viimeisin
16      puts "Viimeisin lisätty asiakas on #{@@nimi}."
17    end
18  end
```

Luokassa on siis kaksi luokkamuuttujaa, raja ja nimi, joista ensimmäiseen tallennetaan kaikille Asiakas-olioille yhteinen saldorajan arvo. Toiseen luokkaolioon taas tallennetaan aina uuden olion luomisen yhteydessä viimeisimpänä luodun olion nimi. Lisäksi luokkaan on määritelty kaikille olioille yhteinen

korota-metodi, joka nostaa @@raja-muuttujan arvoa, sekä metodit viimeisimpänä luodun olion nimen ja olion yleistietojen tulostamiseen. Luodaan nyt luokasta kaksi oliota ja käytetään näitä metodeja:

```
mikko = Asiakas.new("Mikko")
maija = Asiakas.new("Maija")
mikko.saldoraja
mikko.korota
maija.saldoraja
mikko.viimeisin
```

Ohjelma tulostaa seuraavaa:

```
>ruby luokkamuuttujat.rb
Asiakkaan Mikko saldoraja on 150.
Nostetaan kaikkien saldorajaa.
Asiakkaan Maija saldoraja on 250.
Viimeisin lisätty asiakas on Maija.
>Exit code: 0
```

Käytännössä siis luokkamuuttuja on muuttuja, jonka arvo on yhteinen kaikille luokasta tehdyille olioille. Tämä tarkoittaa samalla myös sitä, että luokkamuuttujaa voidaan myös muuttaa mistä tahansa saman luokan oliosta. Esimerkiksi yllä olevassa koodissa `mikko`-nimisen olion `korota`-metodi muutti arvoa myös `maija`-oliolle; lisäksi `mikko`-oliosta kutsuttu `viimeisin`-metodi tulosti viimeisimpänä luodun olion nimen, koska se tallennetaan olion luomisen yhteydessä luokamuuttujaan `@@nimi`, joka on kaikille yhteinen. Erityisesti kannattaa huomata, että luokanmuuttuja `@@nimi` onkin täysin eri asia kuin samanniminen jäsenmuuttuja `@nimi`, joka määräytyy jokaiselle oliolle erikseen.

HUOMIOITA RUBY-KIELESTÄ

Kirjan alkuosan ensimmäisissä kahdeksassa luvussa on nyt tutustuttu Ruby-ohjelmointikieleen ja sen syntaksin perusrakenteisiin. Tässä vaiheessa voidaan hetki arvioida sitä, millainen ohjelmointikieli Ruby oikeastaan onkaan.

Yleisesti ottaen Rubya voidaan pitää hyvin tavanomaisena modernina olio-pohjaisena tulkattavana kielenä. Siitä löytyy kaikki normaalit osat ja ohjelmointirakenteet, kuten toisto- ja valintarakenteet, työkalut tiedostojen käyttämiseen sekä virheenkäsittelytoiminnot. Kuitenkin Ruby-kielen vapaamuotoinen syntaksi sekä mahdollisuus toteuttaa asioita useilla eri tavoilla viehättää varmasti monia kieleen tutustuvia. Lisäksi Rubyn hieman epätavallinen ”kaikki on olioita”-lähestymistapa on saatu toimimaan siten, että se ei perusasioita läpikäytäessä muodostu esteeksi, ja myöhemmässä vaiheessa siitä on jopa hyötyä, esimerkiksi nimiavaruuksien muodossa moduuleja tehtäessä. Kielen kehitys onkin ollut syntaksiin avoimen suhtautumisen ansiota nopeaa. Itse asiassa niin nopeaa, että se on jopa muodustunut ongelmaksi: uudet versiot sisältävät uusia toiminnallisuuksia ja muutoksia siinä määrin, että edes Internet-lähteet eivät ole pysyneet kunnolla kielen kehityksen vauhdissa. Tämän ongelman voi kuitenkin olettaa helpottavan ajan mittaan, kun enimmät muutos- ja laajennusideat on toteutettu.

On myös pidettävä mielessä, että Ruby-ohjelmointikieli ei myöskään olisi lähelläkään nykyistä suosiotaan ilman siihen tarjolla olevia laajennusosia, erityisesti Ruby on Rails-verkko-ohjelmakehystä. Ruby-yhteisön omien tilastojen mukaan Ruby-asennuspaketti on ladattu 4,4 miljoonaa kertaa, kun jokin Rails-projektin asennuspaketti (tavallisimmin Rails tai Instant Rails) on kerännyt yhteensä 2,1 miljoonaa latausta. Tietysti Ruby-asennuspaketteihin pitää huomioida myös Linux- ja Mac-asennukset, joita varovaisesti arvioiden voidaan sanoa olevan ehkä kaksi-kolme miljoonaa, mutta tosiasia on, että Ruby on Rails on hyvin, hyvin keskeinen Ruby-kielen osa ja numerot puhuvat tämän tosiseikan puolesta. Siksi kirja, jossa esitellää Ruby-ohjelmointia, mutta ei puhuta mitään on Rails-kehyksestä, olisi parhaimmillaankin keskeneräinen, ja tämän vuoksi Rails onkin mukana myös tässä kirjassa.

Seuraavassa kirjan osassa vaihdetaankin aihetta perus-Rubysta Ruby on Rails-kehykseen. Rails-kehys on kuitenkin hyvin, hyvin isokokoinen ja monipuolinen kehys. Jo yhtään laajamittaisempiin Rails-projekteihin liittyy useita erilaisia verkkoteknologiaan liittyviä toimintoja kuten tietokantakytkennät, varmennusjärjestelmät sekä tietysti sivujen määritteleminen HTML/CSS-kuvauksilla. Tämän kaiken läpikäyntiä varten tarvittaisi kutakuinkin kirjasarja tai vähintäänkin erillinen aihetta käsittelevä teos, joten lähestymistavaksi on tässä kirjassa valittu esimerkkiprojektit. Kirjan toisessa osassa käydäänkin lyhyen johdannon jälkeen suoraan käsiksi Rails-kehitykseen, johon tutustutaan tekemällä kolme esimerkkiprojektia, joiden avulla on mahdollista saada perustason ymmärrys siitä, kuinka Ruby ja Rails liittyvät toisiinsa, ja kuinka Rails-sivuston toteuttaminen tapahtuu. Lisäksi osion lopussa annetaan vielä lisäneuvoja Rails-kehyksen hyödyntämiseen ja ohjataan oikeaan suuntaan itsenäisen opiskelun jatkamisessa.

OSA II

Ruby on Rails

Ruby-kielen laajennukset, Mikä on Ruby on Rails?

Tutustuminen ympäristöön

Yksinkertaisen sivuston luominen

Tiedon määrittely ja käsittely, blogin tekeminen

Muita Rails-toimintoja

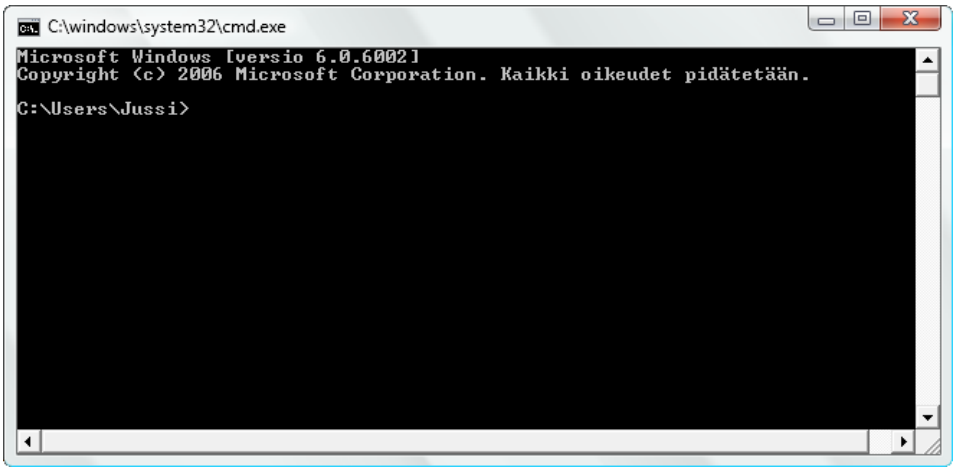
LUKU 9

Ruby-kielen laajennukset, Mikä on Ruby on Rails?

Tässä luvussa esitellään kirjan toinen pääaihe, Ruby on Rails–ohjelmistokehys ja sen kehitysympäristö, jonka avulla on mahdollista luoda verkkopalveluita Ruby-ohjelmointikieltä apunakäyttäen. Luvussa ei vielä tehdä ohjelmointitehtäviä, vaan keskitytään teoriaan siitä, kuinka Ruby on Rails oikeastaan toimii.

RUBY-KIELEN LAAJENNUKSET

Pelkkä Ruby on tehokas ja pienikokoinen ohjelmointikieli, jolla on mahdollisuus nopeasti ja helposti luoda työkaluja, mutta pelkällä peruspaketilla pystyy toteuttamaan ainoastaan rajallisen määrän tehtäviä. Lisäksi Ruby on kohtuullisen nopealla vauhdilla päivittyvä ohjelmointikieli, ja kielestä on jatkuvasti useita eri versioita. Esimerkiksi Rubyn omien sivujen (*http://www.ruby-lang.org*) kautta pelkästään Windows-käyttöjärjestelmille on kolme esikäännettyä versiota. Tämän vuoksi esimerkiksi



Rubyn osien päivittäminen voi olla ongelmallista. Kielen laajennusten asentamista ja hakemista, sekä kielen ohjelmamoduulien ajan tasalla pitämistä varten Rubysta löytyykin perusasennuspaketin mukana tuleva työkalu *Rubygems*.

Rubygems ja sen käyttäminen

Rubygems on siis Ruby-ohjelmointikielen asennuspakettien ja laajennusten hallintaan käytettävä työkalu. Toisin kuin useimmat Windows-ympäristössä toimivat ohjelmat, käytetään Rubygems-työkalua komentokehotteessa, mikä on tavallisempaa erityisesti Unix-käyttöjärjestelmien yhteydessä.

Helpoin tapa päästä komentokehotteeseen Windows-käyttöjärjestelmissä on avata Käynnistä-valikko, ja kirjoittaa alakulmassa olevaan ”Aloita haku” (”*Search programs and files*”)-ikkunaan `cmd` sekä painaa Enter. Toinen helppo tapa on käyttää pikanäppäintä (*Windows+R*) ja kirjoittaa tähän ikkunaan `cmd` sekää painaa Enter. Nyt ruudulla pitäisi olla seuraavanlainen ikkuna:

Komentokehotteen ikkuna Windows-käyttöjärjestelmässä

Jos Ruby-tulkki on asennettu oikein, pitäisi komennon `gem` tuottaa seuraavanlainen tuloste:

```
C:\Users\Jussi>gem
RubyGems is a sophisticated package manager for Ruby.  This is a
basic help message containing pointers to more information.

  Usage:
---leikattu tästä---
```

Tällöin Rubygems on asennettu oikein ja sitä voidaan käyttää laajennusmoduulien asentamiseen ja asennettujen moduulien päivittämiseen. Itse pakettien asentaminen onnistuu yksinkertaisesti käskyllä

```
gem install [paketinnimi]
```

Eli vaikkapa Ruby on Rails-laajennuksen yhteydessä

```
gem install rails
```

Toinen hyödyllinen käsky on yleiskomento jolla Rubygems tarkastaa kaikkien asentamiensa pakettien versionumeron ja tarvittaessa päivittää niitä. Tämä tapahtuu yksinkertaisesti komennolla

```
gem update
```

Joka tapauksessa Rubygems-työkalun avulla voidaan asentaa laajennuspaketteja Ruby-ympäristöön. Rubygems onkin varsin hyödyllinen, koska jotkin paketit tarvitsevat myös muita paketteja toimiakseen tehokkaasti, eli niillä on keskinäisiä riippuvuussuhteita (dependencies). Rubygems huolehtiikin myös näistä tarpeista; se asentaa kaikki tarvittavat paketit siinä tapauksessa, että käyttäjä pyytää asennusta jollekin paketille jolla on riippuvuussuhde pakettiin, jota ei vielä ole asennettu. Parhaiten tämän huomaa asentaessa Rails-kehyksen; sen asennuksen yhteydessä asennetaan myös muita paketteja, kuten rake, activerecord ja actionpack. Listaa asennetuista gem-paketeista on mahdollista tarkastella komennolla

```
gem list –local
```

Tässä vaiheessa asennetaan Rubygems-työkalun avulla Ruby on Rails-ohjelmakehys, ja siirrytään puhumaan kirjan toisesta pääaiheesta. Ohjeet paketin asennukseen löytyy liitteestä 1.

RUBY ON RAILS LYHYESTI

Ruby on Rails (lyhyesti Rails) on kokoelma työkaluja ja laajennusmoduuleja, jotka mahdollistavat dynaamisten webbisovellusten kehittämisen Ruby-ohjelmointikielen avulla. Vaikka Rails voi alkuun vaikuttaa hyvin erilaiselta työkalulta kuin Ruby, käyttää se samaa ohjelmointikieltä ja ymmärtämällä Ruby-ohjelmia on mahdollista saada helposti selvää siitä, mitä Rails-ohjelmissa tehdään.

Rails-ympäristöä on monesti verrattu esimerkiksi PHP-kieleen tai ASP.NET-kehykseen, koska sen avulla voidaan määritellä ja generoida verkkopalveluja Internet-sivustoille. Rails-työkalut on myös monesti ensimmäinen kerta, kun ohjelmoijat tutustuvat Ruby-ohjelmointikieleen, joten eräässä mielessä voidaan sanoa, että Rails ei ole pelkästään sovelluskehys tai yksittäin laajennus Ruby-ohjelmointikieleen, vaan se on kokonainen verkko-ohjelmointipuoli Ruby-kielestä. Lisäksi Rails-ympäristön eduksi voidaan laskea sen laajuus; Ruby on Rails on hyvin laaja sovelluskehys, sisältäen työkalut esimerkiksi testiympäristön, suojattujen toimintojen toteuttamiseen tai tietokantojen rakentamiseen. Rails-ympäristössä ei ylipäänsä ole mitään ilmeisiä puutteita verrattuna mihinkään muihin verkkosovelluksien tekemiseen tarkoitettuun työkaluun. Lisäksi Rails tarjoaa peruskomponentit, joilla on mahdollista saada rakennettua toimivia palveluita hyvinkin nopeasti; esimerkiksi verkkoblogin pystyttäminen onnistuu Rails-ympäristöä tuntevalta alle viidessätoista minuutissa.

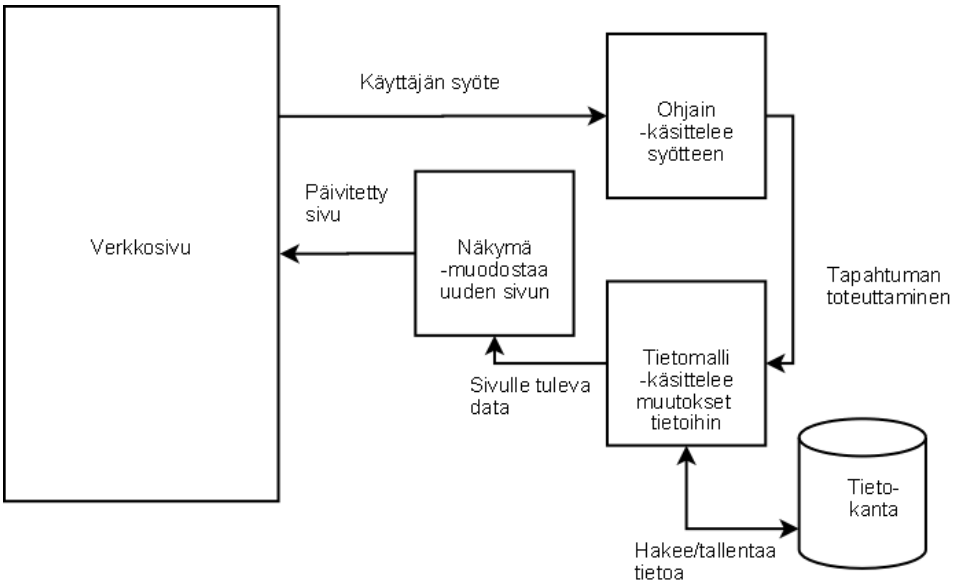
Perusperiaatteet

Rails-kehystä voidaan kuvailla hieman epätavallisesti ketteräksi verkkosovellusten kehykseksi. Useimmiten verkkoon tarkoitettujen palvelujen kehittäminen ja testaaminen on hidasta ja aikaavievää puuhaa, eikä työkalujen ole ollut tarkoituskaan tuottaa järjestelmiä nopeasti vaan ennemminkin virheettömästi. Luonnollisesti virheettömyys on ohjelmoitaessa hyve, eikä Rails tietenkään pyri tätä asiaa muuttamaan, mutta kehitystyön helpottamiseen ja nopeuttamisene Rails pyrkii vaikuttamaan aktiivisesti.

Rails-kehyksen kotisivuilla (*http://rubyonrails.org*) Rails-työkalujen toimintaa ja käyttötarkoitusta kuvaillaan ehkä kaikista parhaiten sillä, että Rails vaatii käyttäjää ainoastaan määrittelemään ne asiat, jotka eivät toimi niin kuin yleisimmin voitaisi olettaa. Käytännössä tämä tarkoittaa sitä, että Rails määrittelee jonkin toimintatavan oletustavaksi, ja käyttää sitä kaikissa muissa tapauksissa paitsi silloin, jos ohjelmaan on erikseen määrätty jokin sen korvaava toimintamalli. Rails-järjestelmän toiminta voidaankin tiivistää kolmeen johtoajatukseen:

- *DRY-Don't Repeat Yourself*, eli ”Älä toista itseäsi”. Tämä siis tarkoittaa sitä, että kerran määriteltyä toimintoa tulisi ennemmin pystyä käyttämään monessa eri paikassa sen sijaan, että se joka kerta joudutaan määrittelemään uudelleen. Tällä myös pyritään vähentämään järjestelmään päätyvien virheiden, bugien, määrää poistamalla mahdollisuus ristiriitaisiin toimintaohjeisiin saman toiminnon eri määritelmissä.
- *Convetion over Configuration,* eli ”Tavanmukaisuus ennen virittelyä”, tarkoittaen sitä että Rails määrittelee asiat oma-aloitteisesti sen mukaan mitä käyttäjä on aikeissa tehdä sen sijaan, että käyttäjän pitää määritellä ja säätää jokainen yksityiskohta paikalleen käsin. Tämä tarkoittaa samalla myös sitä, että jos jokin toimintatapa on määritelty, ei sitä pitäisi muuttaa ilman hyvää syytä, vaan pyrkiä aina noudattamaan kehyksen määrittelemiä ”hyviä tapoja”.
- *REST is the best pattern for web applications,* tarkoittaaen sitä, että RESTful-malli on paras toimintatapa verkkosovelluksille. RESTful-malli on periaatteessa http-tiedonsiirron perusmalliin pohjautuva tapa siirtyä tilasta toiseen; asiat tapahtuvat vain käyttäjän niitä pyytäessä ja järjestelmä sisältää erikseen määriteltyjä oletusresursseja näiden asioiden tekemiseen. Rails-sovellusten ohjaimet tulisikin mallintaa RESTfull-mallin mukaisesti, tukeutuen CRUD (C*reate, Read, Update, Delete*)-toimintoihin.

Rails-ympäristön tavallisin kritiikki onkin siinä, että se ei ole erityisen vapaamielinen oletustapojensa suhteen. Jos Rails-ohjelmoija ei ole vielä tottunut johonkin toiseen toimintamalliin, on Rails-ympäristön automatiikka ja työkalut varmasti näppäriä ja mukavia. Tässä suhteessa Rails-ympäristöllä työskentely muistuttaakin pitkäli rakennuspalikoilla työskentelyä; otetaan tarvittavat palikat ja laitetaan niitä kiinni toisiinsa kunnes lopputulos miellyttää silmää. Kolikon kääntöpuolena kuitenkin on se, että toiseen lähestymistapaan tottuneille Rails on rajoittava ympäristö joka yrittää kokoajan pakottaa toimimaan opittujen, toimivaksi todettujen tapojen vastaisesti. Tässä mielessä Rails-ympäristöön tutustuminen ilman aiempaa kokemusta ASP.NETistä tai PHPstä voi itse asiassa olla jopa hyödyllistä.



MVC-arkkitehtuuri

Teknisesti Rails-kehys on rakennettu MVC-arkkitehtuurin (*Model, View, Controller*) ympärille. Käytännössä tämä tarkoittaa siis sitä, että Rails-sovelluksissa käyttöliittymän komponentit eli näkymät, toiminnalliset osat eli ohjaimet sekä käytettävä data eli mallit, ovat määritelty toisistaan erillään.

Kuinka MVC-arkkitehtuuri toimii

Periaatteessa MVC-arkkitehtuurilla rakennetut ohjelmat toistavat uudestaan ja uudestaan seuraavanlaista tomintakierrosta:

1. Käyttäjä antaa syötteen näkymään ja käynnistää jonkin toiminnon, kuten esimerkiksi kirjoittaa uuden viestin verkkoblogiinsa.
2. Ohjain saa näkymään annetun syötteen, eli tässä tapauksessa kirjoitetun viestin.
3. Ohjain käyttää mallia pohjana luodessaan ja tallentaessaan syötteestä uuden viestin.
4. Ohjain antaa näkymälle käskyn päivittää itsensä, ja näkymä laittaa uuden viestin näkymään verkkoblogin uusimpana viestinä.
5. Käyttöliittymä jää odottamaan käyttäjältä uutta syötettä.

Tämä kiertokulku tapahtuu siis uudestaan ja uudestaan niin kauan, kunnes käyttäjä poistuu palvelusta. Käytännössä mallien, näkymien ja ohjaimien työnjako onkin seuraavanlainen:

- Mallit (*Models*) määrittelevät millaista tietoa ohjelma käyttää, ja miten annettuja tietoja voidaan muutella. Malleihin määritellään myös eri toiminnot, joita tiedolta vaaditaan ja miten eri mallit liittyvät toisiinsa.
- Näkymät (*Views*) määrittelevät palvelun käyttöliittymän. Rails-ohjelmissa nämä ovat useimmiten HTML-tiedostoja joiden sekaan on kirjoitettu Ruby-koodia millä ohjelma tuottaa toivotunlaisen lopputuloksen.
- Ohjaimet (*Controllers*) vastaanottavat tulevat syötteet, ja käyttävät malleja tehdäkseen niistä käytettävää dataa. Lisäksi ohjaimet vastaavat mallien mukaisen tiedon toimittamisesta näkymille sekä yleisesti erilaisten parametrien käsittelystä.

Nämä kolme rakennetta toimivat päällekäin ja ovat liittyneinä toisiinsa hyvin läheisellä tavalla. Se, mikä toiminto kuuluu millekin osalle voi olla esimerkiksi PHP-ympäristöön tottuneelle hieman epämääräistä, mutta käyttämällä oppiminen on helpoin tapa tulla tutuksi järjestelmän toimintaan. Kaikenkaikkiaan Rails-kehyksen toiminta voi vielä tässä vaiheessa vaikuttaa hieman epämääräiseltä, mutta seuraavassa luvussa asennetaan Ruby on Rails-ympäristö ja luodaan ensimmäinen projekti. Tässä yhteydessä pitäisi erilaisten komponenttien ja toimintamallin periaatteiden alkaa hahmottua paremmin.

LUKU 10

Tutustuminen ympäristöön

Rails-työkalut
Uuden projektin luominen
Rails-komponentit

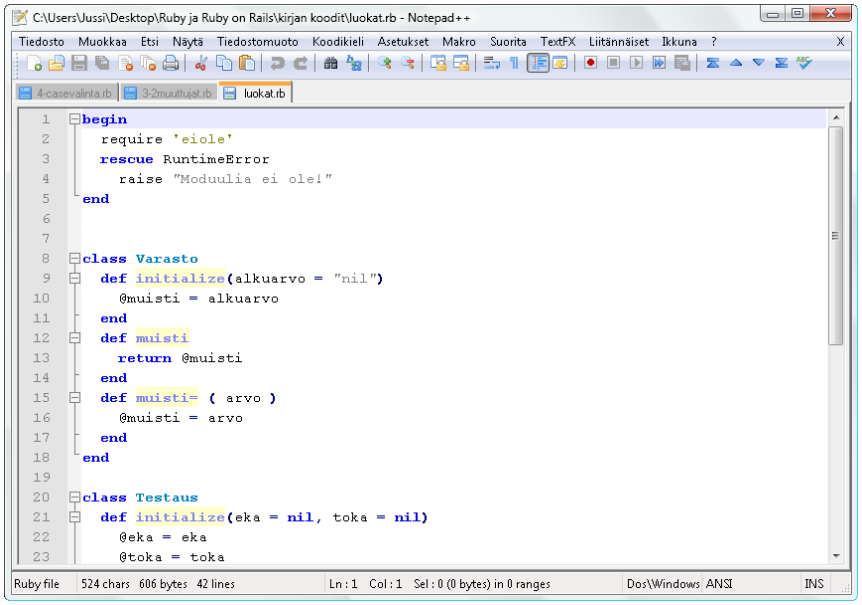
Tässä luvussa aloitetaan Ruby on Rails-ohjelmistokehyksen käyttäminen luomalla yksinkertainen HelloWorld-projekti sekä tekemällä ensimmäiset omat näkymät ja ohjaimet. Projektissa tutustutaan yleisellä tasolla siihen, mitä Rails-kehys tuo mukanaan ja kuinka siihen voidaan luoda eri komponentteja ja miten nämä komponentit toimivat yhdessä.

RAILS-TYÖKALUT

Tähän asti olemme käyttäneet ohjelmointiin SciTE editoria, joka onkin yksinkertaisuutensa takia sopiva ympäristö Ruby kielen perusteiden opetteluun. Rails-ympäristössä työskentelyä varten tarvitsemme kuitenkin myös muita työkaluja, sillä pelkkä SciTE ei enää riitä ohjelmointitarpeita varten. Ensinnäkin, Rails tarvitsee toimiakseen tietokantajärjestelmän, sekä jonkinlaisen virtuaalipalvelimen, jonka avulla Rails-sovelluksia voidaan testata ilman että tiedostot pitää aina siirtää verkkopalvelimelle.

Tässä vaiheessa tulee viimeistään tehdä Rails-kehyksen sekä SQLite-tietokannan asennukset. Ohjeet tähän löytyy Liitteestä 1.

Helpoin ongelma ratkaistavaksi on lähdekoodieditorin valinta. Koska Rails-projekteissa voi monesti olla useita yhtäaikaisia tiedostoja, on editoriohjelma syytä olla sellainen, joka tukee usean tiedoston yhtäaikaista aukioloa. Pelkkä SciTE riittää varmasti hyvin, mutta esimerkiksi Notepad++ (*http://notepad-plus.sourceforge.net*) on ilmainen open source-editori, joka tukee useita erilaisia ohjelmointikieliä, sekä asentaa hyödyllisen ”Edit with Notepad++”-valinnan vasemman hiirennapin valikkoon. Tietysti muutkin editori- ja kehitystyökalut (esimerkiksi Netbeans, Eclipse) ovat hyviä, joten valinta perustuu pitkälti siihen mistä pitää. Monesti kuitenkin editori- ja kehitystyökalut ovat varsin raskaita ja monimutkaisia käyttää johtuen siitä, että ne on tarkoitettu ensisijaisesti ammattilaiskäyttöön. Aloittelijoille yksinkertaisemmat työkalut voivat olla hyödyllisempiä, koska ne ovat suoraviivaisempia eivätkä sisällä ylimääräisiä, perusharjoitusten kannalta tarpeettomia, toimintoja. Esimerkiksi tämän kirjan esimerkkiprojektit on koodattu Notepad++-editorilla, joka näyttää tältä:



Notepad ++- editori ja Ruby-koodia.

Tietokantaratkaisun kanssa on käytännössä kolme vaihtoehtoa; SQLite3, MySQL ja PostgreSQL. Näistä MySQL ja PostgreSQL ovat isokokoisia, täydellisiä tietokantajärjestelmiä, joiden avulla pystytään toteuttamaan mitä tahansa tietokantoihin liittyvää. SQLite on kevyt tietokantasovellus, joka pakkaa alle megatavun kokoiseen asennuspakettiin suurimman osan SQL-tietokantojen toiminnoista, ja tuottaa kevyitä, yhden tiedoston kokoisia tietokantarakenteita. Periaatteessa valinnaksi käy mikä tahansa, mutta kirjan esimerkkeihin riittää mainiosti SQLite3-tietokanta, joka samalla on Ruby on Rails-ympäristön oletustietokantapalvelin. Toinen hyvä vaihtoehto voisi olla vapaana lähdekoodina saatavilla oleva MySQL, mutta sen asentaminen ja käyttöönottaminen on esimerkkiprojekteja ajatellen tarpeettoman suurikokoinen tehtävä.

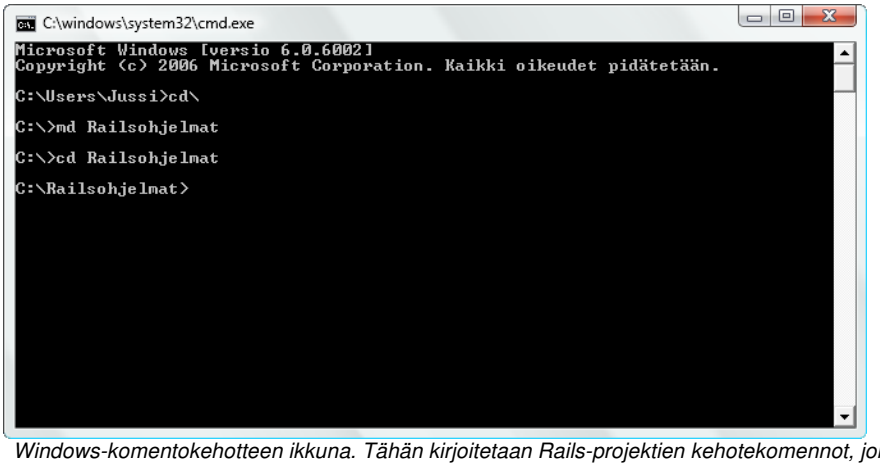
UUDEN PROJEKTIN LUOMINEN

Ruby on Rails-projektin luominen ei itse asiassa ole kovinkaan vaikeaa sen jälkeen kun perustyökalut on asennettu. Itseasiassa, yksinkertaisin toimiva projekti voidaan luoda ja käynnistää kolmellä

komentokehoitteen käskyllä. Kuitenkin, koska Rails-kehityksessä käytetään hyvin paljon komentokehotetta johtuen kehyksen Unix-pohjaisesta rakenteesta, on ennen projektin luomista syytä käydä nopeasti läpi se, miten Windows-komentokehotetta käytetään.

Komentokehotteen käyttäminen

Komentokehoite siis käynnistetään joko Käynnistä-valikosta käskyllä cmd, tai pikanäppäimellä Win+R ja kirjoittamalla tähän cmd. Tämä ikkuna kannattaa pitää Rails-sovellusta tehdessä jatkuvasti auki, sillä komentokehotteen kautta pystytään helpoiten käyttämään Rails-skriptejä, joilla voidaan helposti luoda ohjaimia, malleja ja näkymiä Rails-projekteihin, sekä esimerkiksi käynnistää virtuaalipalvelin sekä hallita projektiin liitettyä tietokantaa.



Windows-komentokehotteen ikkuna. Tähän kirjoitetaan Rails-projektien kehotekomennot, joilla esimerkiksi luodaan projekti tai käynnistetään virtuaalipalvelin.

Ensimmäinen asia, mitä Rails-projekteja varten kannattaa tehdä, on luoda helposti saatavilla oleva hakemisto, johon uudet projektit luodaan. Esimerkiksi `C`-aseman juuressa oleva hakemisto `Railsohjelmat` on hyvä vaihtoehto, koska siihen löytää nopeasti mikäli vahingossa sulkee kehotteen ikkunan. Tämä voidaan luoda komentokehotteessa käskyillä

```
cd\
md Railsohjelmat
```

jonka jälkeen kyseisen kansion pitäisi löytyä C-aseman juuresta. Seuraavaksi siirrytään kansioon käskyllä

```
cd Railsohjelmat
```

jolloin kehoitteessa kohdehakemisto vaihtuu luonnollisesti nimeen *C:\Railsohjelmat*. Tähän kansioon jatkossa luodaan kaikki Rails-projektihakemistot. Oheisessa taulukossa on vielä jotain käskyjä, jotka kannattaa ainakin muistaa kehoitetta käytettäessä.

Taulukko 10.1 Joitan komentokehotteen käskyjä

Käsky	Tapahtuma
`cd <nimi>`	Mene alihakemistoon <nimi>
`md <nimi>`	Luo alihakemisto nimeltä <nimi>
`cd ..`	Mene ylempään hakemistoon.
`rd <nimi>`	Poista alihakemisto <nimi>, toimii ainoastaan tyhjään alihakemistoon.
`del <nimi>`	Poistaa tiedoston <nimi> tai tyhjentää hakemiston <nimi>
`dir`	Listaa nykyisen hakemiston tiedostot

`cd \`	Mene levyn juureen (C:\, D:\)
`C:`	Siirry C-asemalle, normaalisti aseman juureen.

Kannattaa myös huomata, että joissain Windows-versioissa toimii kehotteessa osoitteen automaattinen täydennys. Jos halutaan esimerkiksi siirtyä hakemistoon *todellapitkanimijokaolihuonoajatus*\, voi riittää pelkästään se, että kirjoitetaan *cd todellapi* (käsky ja muutama kansion nimen ensimmäinen kirjain) ja painetaan tabulaattoria. Joissain Windows-versioissa kehoite täydentää loppuosan.

Rails-projektin alustaminen ja luotavat tiedostot

Seuraavaksi voidaan siirtyä itse Rails-ohjelman luomiseen ja alustamiseen. Kirjoitetaan Rails-hakemistossa (*C:\Railsohjelmat*) seuraava käsky

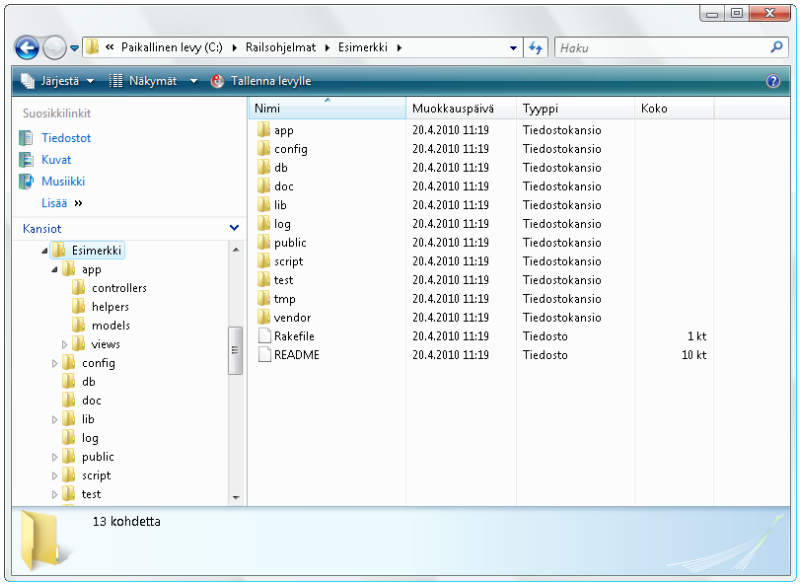
```
rails EkaOhjelma
```

Joka antaa Ruby on Rails-kehykselle käskyn luoda projektikansio ja perustiedostot `EkaOhjelma`-nimiselle Rails-sovellukselle. Ohjelma miettii hetken, ja ryhtyy tämän jälkeen luomaan useita tiedostoja ja alihakemistoja, ja saatuaan tehtävän valmiiksi palaa takaisin kehotteeseen.

Jos tässä vaiheessa tulee ilmoitus ”rails ei ole tunnistettu sisäiseksi tai ulkoiseksi komennoksi”, on jokin osa Rails-kehyksestä asennettu väärin. Syy tähän on tavallisimmin se, että osia ei ole asennettu järjestelmänvalvojan oikeuksilla tai että konetta ei ole käynnistetty uudelleen komponenttien asentamisen jälkeen.

Kun Rails on saanut projektin luomisen valmiiksi, siirrytään projektin juurihakemistoon (tässä tapauksessa *C:\Railsohjelmat\EkaOhjelma*). Tämä on jatkossa tämän projektin päähakemisto, jossa kaikki projektiin liittyvät käskyt (palvelimen käynnistäminen, komponenttien luominen...) toteutetaan.

Projektitiedostojen luomisen yhteydessä Rails loi joukon hakemistoja ja tiedostoja. Jos kaikki meni oikein, pitäisi käytettävään projektikansioon olla nyt generoitu 11 kansiota sekä kaksi juurihakemistossa olevaa tiedostoa:



Projektin oletuspuu Esimerkki-nimiselle Rails-projektille

Mitä nämä kansiot ja tiedostot sitten tarkoittavat? Käytännössä näihin kansioihin kirjoitetaan ja tallennetaan kaikki tieto, mitä Rails-sovellukseen tullaan tekemään. Kansioiden käyttökohteet ovat kutakuinkin seuraavanlaiset:

Taulukko 10.1 Rails-projektin oletuskansioiden käyttötarkoitukset	
Kansio/Tiedosto	Käyttötarkoitus

app/	Tähän kansioon ja sen alikansioihin määritellään Rails-sovelluksen ohjaimet, mallit ja näkymät, ja se sisältää suurimman osan projektiin kirjoitettavasta koodista alikansioissaan **controllers**, **models** ja **views**.
config/	Tämä kansio sisältää sovelluksen ajoympäristöä koskevat asetukset ja määritykset sekä reititystiedot, joilla kerrotaan mikä resurssi löytyy mistäkin.
db/	Sisältää määrittelyt ja asetukset sovelluksen käyttämillä tietokantarajapinnoille.
doc/	Tarkoitettu Rails-tiedostojen dokumentaation säilytyspaikaksi
lib/	Rails-laajennusmoduulien asennuskansio
log/	Rails-sovelluksen lokitiedost tallennetaan tänne.
public/	Tähän kansioon tallennetaan kaikki sovelluksen ulkomaailmaan näkyvät tiedostot, kuten verkkosivun kuvat tai CSS-tyyliohjeet sekä ladattavaksi tarkoitetut saattiset tiedostot sekä html-virhesivut.
script/	Sisältää Rails-sovelluksen käyttämät skriptitiedostot joilla voidaan luoda ohjaimia, tehdä scaffold-peruskehys tai käynnistää virtuaalipalvelin.
test/	Tähän kansioon voidaan määritellä Rails-sovelluksen testausympäristö ja siinä käytetyt testicaset.
tmp/	Tähän kansioon tallennetaan väliaikaistiedostot.
vendor/	Tähän kansioon laitetaan kaikki kolmannen osapuolen koodi.
README	Tiedosto, johon on mahdollista kirjoittaa lyhyt esittely tai ohje Rails-sovelluksesta.
Rakefile	Sisältää rake-skriptit jotka voidaan ajaa komentokehotteesta.

Käytännössä suurin osa tämän kirjan Rails-koodista tullaan kirjoittamaan `app`-kansioon. Seuraavaksi testataan, että SQLite-tietokantaohjain on laitettu oikeaan järjestelmäkansioon. Annetaan projektin päähakemistossa käsky

```
rake db:create
```

Jos tietokantaohjain on asennettu oikein, ei käsky anna virheilmoitusta, ja projektin db-kansioon ilmestyi tiedosto `development.sglite3`. Lopuksi voidaan vielä kokeilla, että virtuaalipalvelin toimii oikein, ja antaa Railsille määräys käynnistää projekti virtuaalipalvelimelle, joka toimitettiin Rails-kehyksen mukana. Kirjoitetaan projektin juuressa kehotteen ikkunaan komento

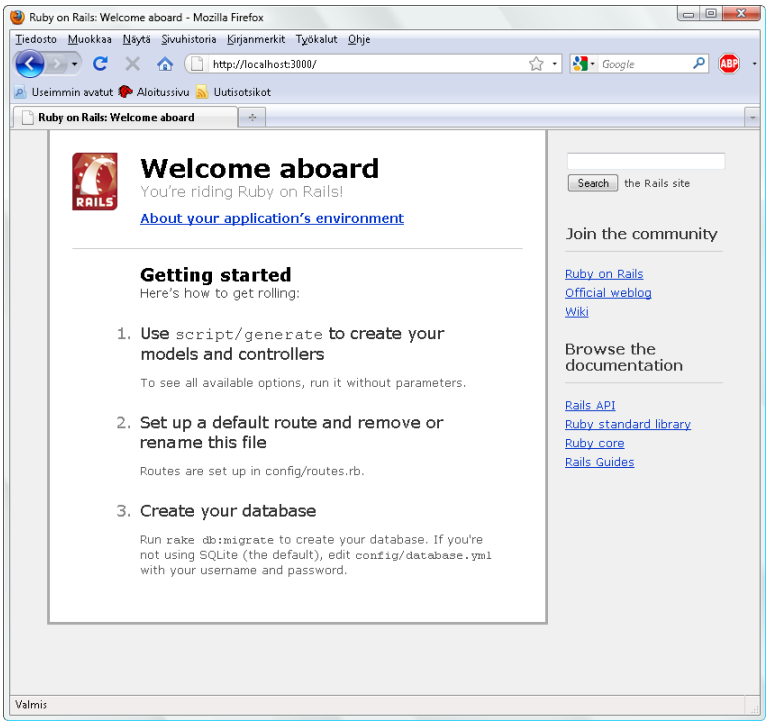
```
ruby script/server
```

Käynnistysvaiheessa tietokoneen palomuuri voi antaa varoituksia tai pyytää lupaa ohjelman, WEBrick-virtuaalipalvelimen käynnistämiseen. Vaikka palvelimen ollessa käynnissä ohjelma näyttää verkkosivua, ei se näy ulkomaailmaan eikä tietokoneelle ole yhteyttä muualta.

Nyt Rails- käynnistää WEBrick-virtuaalipalvelimen kehotteen ikkunaan. Ohjelma lataa hetkenaikaa, ja ilmoittaa olevansa toimintavalmis kun ikkunaan tulee tämänkaltainen teksti:

```
C:\Railsohjelmat\EkaRails>ruby script/server
=> Booting WEBrick
=> Rails 2.3.5 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2010-02-23 15:40:30] INFO  WEBrick 1.3.1
[2010-02-23 15:40:30] INFO  ruby 1.8.6 (2008-08-11) [i386-mswin32]
[2010-02-23 15:40:30] INFO  WEBrick::HTTPServer#start: pid=2940 port=3000
```

Nyt kun virtuaalipalvelin on käynnissä, käynnistetään myös Internet-selain ja kirjoitetaan osoitteeksi *localhost:3000*, johon pitäisi latautua seuraavanlainen sivusto:



Rails-oletusikkuna; Rails-ympäristö on asennettu oikein ja kaikki toimii.

Jos ohjelma Rails-sovellus käynnistyi oikein ja muutkin asiat ovat kunnossa, voidaan virtuaalipalvelin sammuttaa valitsemalla komentokehote ja painamalla *Ctrl-C*.

RAILS-KOMPONENTIT

Nyt tiedetään siis se, että Rails-ympäristö on asennettu oikein, ja varsinainen Rails-sovellusten tekeminen voidaan aloittaa. Tehdään siis vielä luvun loppuun esimerkkisivusto, jonka avulla voidaan hieman tarkastella sitä, miten Rails-komponentteja tehdään. Ensiksi luodaan uusi projektikansio nimeltä `tervehtija`. Tämä toteutetaan menemällä komentokehotteessa projektikansioon *C:\Railsohjelmat* ja antamalla käsky

```
rails tervehtija
```

Nyt Rails-systeemi luo projektitiedostot ja alustaa sivuston valmiiksi samaan tapaan kuin aikaisemmin. Tällä kertaa kuitenkin luodaan myös jotain uutta; tämän vuoksi tässä projektissa luodaan uusi sivu, johon tervehtimisikkuna toteutetaan.

Ohjaimen tekeminen

Toisen osan ensimmäisessä luvussa puhuttiin MVC-arkkitehtuurista, jossa toiminnot siis toteutetaan malleina, ohjaimina ja näkyminä. Jotta Rails-sovelluksessa saadaan jotain toimimaan, luodaan ensimmäisenä ohjain. Tämä tehdään menemällä Rails-projektin kansioon ja antamalla käsky

```
ruby script/generate controller <nimi>
```

Joka luo projektiin uuden kontrollerin nimeltä `<nimi>`. Tehdään nyt kontrolleri nimeltään `tervehtija`, jolloin ohjelman pitäisi tulostaa seuraavaa:

```
C:\Railsohjelmat\tervehtija>ruby script/generate controller Tervehdi
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/tervehdi
      exists  test/functional/
      create  test/unit/helpers/
      create  app/controllers/tervehdi_controller.rb
```

```
      create  test/functional/tervehdi_controller_test.rb
      create  app/helpers/tervehdi_helper.rb
      create  test/unit/helpers/tervehdi_helper_test.rb

C:\Railsohjelmat\tervehtija>
```

Siirrytään seuraavaksi projektikansiossa kansioon *app/controllers/* ja avataan sieltä tiedosto `tervehdi_controller.rb` lähdekoodin editointiohjelmalla. Kuten tiedostopäätteestä voidaan päätellä, ovat Rails-kontrollerien tiedostot itse asiassa normaaleja Ruby-lähdekooditiedostoja. Tiedoston sisältö näyttää nyt tältä:

```
class TervehdiController < ApplicationController
end
```

Eli oikeastaan ei muuta kuin luokkamääritelmä luokalle `TervehdiController`, joka on peritty `ApplicationController`-äitiluokasta, joka samalla on Rails-ympäristön ohjainten perusluokka. Lisätään luokkaan uusina toimintoina metodit nimeltä `terve` ja `nakemiin`, jolla ei ole muita toiminnallisuuksia kuin määritelty viesti, joka taas on tallennettu sisäiseen muuttujaan `viesti`:

```
01  class TervehdiController < ApplicationController
02  def terve
03    @viesti = "Moi maailma!"
04  end
05
06  def nakemiin
07    @viesti = "Näkemiin maailma."
08  end
09
10  end
```

Nyt ohjelmaan on luotu kontrolleri, johon on määritelty kaksi funktioita `terve` ja `nakemiin`, joihin on lisäksi määritelty arvoksi muuttujaan tallennettu viesti. Jotta saamme tämän viestin oikeasti näkyviin, tarvitaan vielä erillinen näkymä sen esittämistä varten. Seuraavaksi toteutetaan siis sellainen.

Näkymän tekeminen

Ensinnäkin, siirrytään projektihakemistossa kansioon *app/views/* johon Rails on luonut uusia kansioita. Siirrytään nyt kansioon `tervehdi`, koska tarkoituksena on tehdä äsken määritellyille ohjaimen `tervehdi` toiminnoille näkymät. Luodaan kansioon tiedosto nimeltä `terve.html.erb` (nimen tunniste on tosiaan kaksiosainen `.html.erb`, ja se tarkoittaa Rails-komentoja sisältävää HTML-kuvausta, *HTML with Embedded Ruby*). Joissain yheyksissä saatetaan myös mainita pääte `.rhtml`, joka on Railsin aiempien versioiden käyttämä tiedostopääte.

Rails-ohjelmat yhdistää näkymät ja ohjaimet automaattisesti siten, että ohjaimen nimeä vastaava kansio luodaan ohjainta luodessa views-kansioon, ja tähän kansioon luotu metodin nimeä vastaava tiedoston nimi (esimerkki terve.html.erb) on ohjaimen samannimisen metodin näkymä.

Tässä esimerkissä halutaan siis ainoastaan saada kontrollerissa tervehtija metodissa terve määritelty teksti näkyviin. Tätä varten voidaan kirjoittaa tiedostoon `terve.html.erb` seuraavanlainen yksinkertainen html-määritelmä:

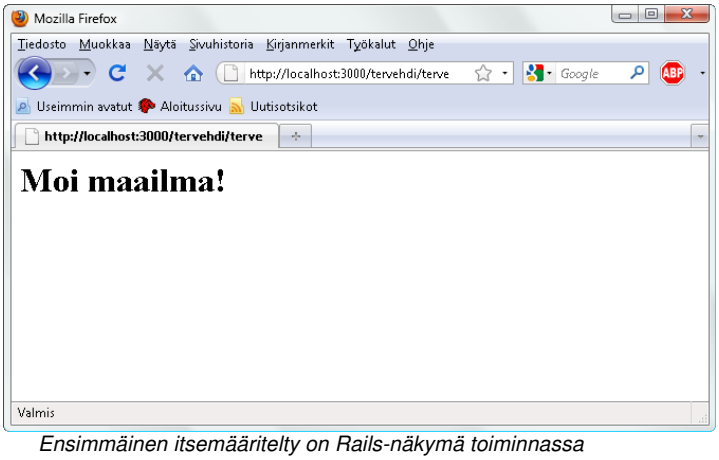
```
01  <html>
02    <body>
03      <h1> <%= @viesti %> </h1>
04    </body>
05  </html>
```

Rails-koodia käyttävät käskyt sijoitetaan html-kuvauksessa **<%** ja **%>**-merkkien väliin; Muu osa näkymästä on normaalia html-kuvauskieltä. Siitä puhutaan enemmän liitteessä 2.

Nyt näkymä on saatu valmiiksi, joten käynnistetään Rails-sovellus kuten aikaisemmin käskyllä `ruby script/server` projektin kansiossa. Kun palvelin on käynnistynyt, voidaan luotua näkymää käydä tarkastelemassa osoitteessa

http://localhost:3000/tervehdi/terve

Nyt tervehtimisviesti on saatu näkyviin! Jos kaikki meni oikein, on selaimen ikkunassa seuraavanlainen näkymä:



Ensimmäinen itsemääritelty on Rails-näkymä toiminnassa

Jos näkymän määrittelevästä tiedostosta otetaan projektihakemistossa kopio ja tallennetaan se samaan kansioon nimellä `nakemiin.html.erb`, nähdään toisessa näkymässä osoitteessa

http://localhost:3000/tervehdi/nakemiin

myös toinen ohjaimeen määritelty viesti. Kannattaa huomata, että tämä voidaan tehdä ilman että virtuaalipalvelin käynnistetään uudelleen; on Rails-sivut parsitaan kokoon dynaamisesti siinä vaiheessa kun niitä kutsutaan, joten projektin näkymien muokkaaminen onnistuu myös sitä käyttävän virtuaalipalvelimen ollessa käynnissä. Ainoastaan jotkut tiedostot, kuten sivujen sijainnin määrittelevän *config\routes.rb* muokkaaminen vaativat palvelimen käynnistämistä uudelleen.

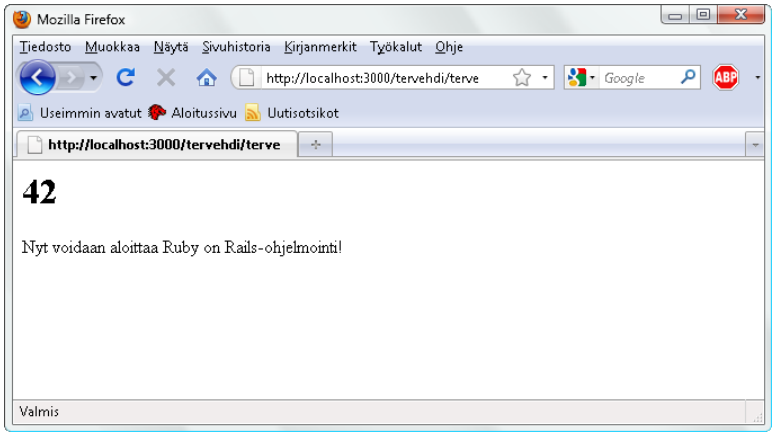
Lisäksi kannattaa myös huomata, että näkymän ikkunassa voidaan Ruby-osia ja HTML-kuvausta sekoittaa vapaasti keskenään. Jos `terve.html.erb`-kuvaukseen lisätään riville viisi seuraava teksti

```
<p>Nyt voidaan aloittaa Ruby on Rails-ohjelmointi!</p>
```

Ilmestyy kyseinen teksti myös sivuston ikkunaan. Vastaavasti jos taas tervehdi-ohjaimeen muutetaan metodin `terve` koodi muotoon

```
@viesti = (4000 + 200) / 100
```

Ilmestyy näkymään Ruby-tulkin laskema laskutehtävän vastaus:



Muutettu on Rails-sivu; näkymään voidaan kirjoittaa vapaasti HTML-kuvausta ja ohjaimessa käyttää Ruby-koodia sisällön luomiseen.

Käytännössä tällä on siis tarkoitus kuvata sitä, että ohjaimeen voidaan kirjoittaa vapaasti Ruby-lähdekoodia, jonka on Rails-kehys suorittaa luodessaan sivustoa näkymän pohjalta. Vastaavasti näkymään taas voidaan kirjoittaa vapaasti HTML-elementtejä, Rubyn generoiman sisällön tullessa `<%` ja `%>`-merkkien väliin jäävälle alueelle. Notaation avulla voidaan tehdä html-sivun kokoamiseen jopa valintarakenteita yhdistelemällä `<% <rubykoodia> %>` ja `<% end %>` -merkkejä mutta niistä puhutaan lisää myöhemmin..

<% %> ja <%= %> ovat molemmat toimivia *Embedded Ruby* –merkkitageja. <% -alkuiset lohkot ainoastaan suorittavat sen sisään kirjoitetun Ruby-koodin html-sivun parsimisvaiheessa, <%= -alkuiset tulostavat koodin tuottaman vastauksen html-kuvaukseen.

Tietenkin on Rails-ympäristö ei käytännössä ole ihan näin suoraviivainen tai yksinkertainen, mutta periaatteessa tämä on se, mistä on Rails-ohjelmoinnissa on kyse: Ruby-koodi kirjoitetaan ohjaimeen ja html-kuvaus, johon merkitään Rubylla luodun materiaalin sijoituspaikat, näkymään. Tietenkin valmiiden elementtien löytäminen ja toimintojen tekeminen vaatii enemmän vaivaa kuin yhden rivin muutokset..

Seuraavassa luvussa tehdään ensimmäinen oikea ohjelma, missä tutustutaan on Rails-kehyksen tarjoamiin varsinaisiin elementteihin ja tehdään verkossa toimiva muistikirja.

LUKU 11

Yksinkertaisen sivuston luominen

Projektin määrittely
Omien toimintojen lisääminen
Yhteenveto projektista

Tässä luvussa muodostetaan ensimmäinen oikea Rails-sivusto. Luvun projektissa luodaan pohja verkkosivuilla toimivalle muistikirjalle, määritellään Rails-komponentteja ja lisätään sivustolle uusia näkymiä hyödyntäen Rails-kehyksen automaattisia toimintoja.

PROJEKTIN MÄÄRITTELY

Ensimmäisessä Ruby on Rails-projektissa tutustutaan Rails-kielen perusasioihin luomalla sivuston johon voidaan jättää ylös muistilappuja tai ylipäänsä viestejä. Kuten ennekin, aloitetaan projektin toteuttaminen luomalla uusi projektihakemisto, tällä kertaa nimeltään muistio komennolla

```
C:\Railsohjelmat\ rails muistio
```

Joka totuttuun tapaan luo uuden kansion ja tuottaa sinne perusrungon uudelle projektille.

Uuden aloitussivun määrittely

Ensimmäinen asia, mitä uutta, oikeaa Rails-ohjelmaa, varten tarvitaan on tietenkin aloitussivun määritteleminen joksikin muuksi kuin Rails-projektien käyttämäksi HelloWorld-etusivuksi, joka nähtiin aikaisemmin edellisessä esimerkissä. Luonnollisesti tätä varten tarvitaan siis ohjain ja näkymä, joksi aloitussivu määritellään. Ensimmäinen tehtävä on siis luoda uusi ohjain, joka tehdään samaan tapaan kuin aikaisemmin käyttämällä script/generate –komentoa. Tällä Annetaan nyt luotavalle ohjaimelle nimeksi `koti`, sekä samalla ylimääräisenä argumenttina `index`, joka automaattisesti luo ohjaimeen koti metodin nimeltä `index`. Käsky on siis muodossa

```
ruby script/generate controller koti index
```

Jos luotua ohjainta tarkastellaan lähdekoodieditorissa, voidaan sen nähdä olevan seuraavanlainen:

```
class KotiController < ApplicationController
  def index
  end

end
```

Käytännössä `script/generate controller`-komennolle voidaan siis antaa luotavan ohjaimen nimen lisäksi automaattisesti luotavien metodien nimiä seuraavalla tavalla:

```
ruby script/generate controller <ohjaimennimi> <metodinnimet>
```

Metodien nimiä voidaan antaa kerralla useita, ja ne erotellaan toisistaan välilyönnillä. Esimerkiksi käsky

```
ruby script/generate controller testi eka toka
```

Käsky loisi `testi`-nimisen ohjaimen, johon on valmiiksi luotu metodit nimeltä `eka` ja `toka`, joissa tosin ei vielä tässä vaiheessa ole mitään määriteltyä toimintoa. Samalla lisäksi luodaan *app\views*-hakemistoon metodeja vastaavat `<nimi>.html.erb`-tiedostot.

Kannattaa huomata, nimi index on hieman poikkeuksellinen verrattuna esimerkiksi metodinnimiin eka tai toka; index on aina oletusnimi ”pääsivulle”, joka ladataan oletuksena, joten index-niminen näkymä on se, jota käytetään kun siirrytään pelkän ohjaimen nimen perusteella sivulta toiselle.

Nyt Rails-sivustolle on siis luotu uusi ohjain `koti`, jossa on metodi, ja samalla näkymä, nimeltä `index`. Koska ohjelmaa varten on olemassa uusi kotisivu, voidaan tässä vaiheessa poistaa vanha, alkuperäinen Rails-ohjelman oletussivu projektihakemistosta. Tämä sivu löytyy hakemistosta *public* nimellä `index.html`. Koska sivua ei tarvita enää sen jälkeen kun uusi aloitussivu on tehty valmiiksi, voidaan tämä tiedosto poistaa kokonaan antamalla projektihakemistossa käsky

```
del public\index.html
```

Seuraavaksi sovellukseen tulee vielä määritellä uuden aloitussivun sijainti reititystietoihin, jotta projekti itsekin löytää uuden aloitussivun ohjainten ja näkymien joukosta. Tämä voidaan tehdä projektikansion tiedostossa *config\routes.rb*. Kun tämä tiedosto avataan editoriin, pitäsi sen alkaa seuraavankaltaisilla riveillä ja sisältää useita rivejä kommentoitua tekstiä:

```
ActionController::Routing::Routes.draw do |map|
```

```
  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
```

Tekstissä esitellään erilaisia esimerkkitapoja määritellä Rails-sovellukselle kytkentöjä näkymien ja ohjaimien välille. Tällä kertaa ei kuitenkaan keskitytä vielä niihin vaan etsitään kyseisen tiedoston lopusta seuraavat kaksi riviä:

```
map.connect ':controller/:action/:id'
map.connect ':controller/:action/:id.:format'
```

Nämä rivit määrittelevät tavat, jolla on Rails-projekti yrittää löytää ohjaimelle tehdyn näkymän ja yhdistää annetun osoitteen näkymiin. Esimerkiksi

```
/merkinta/view/1
```

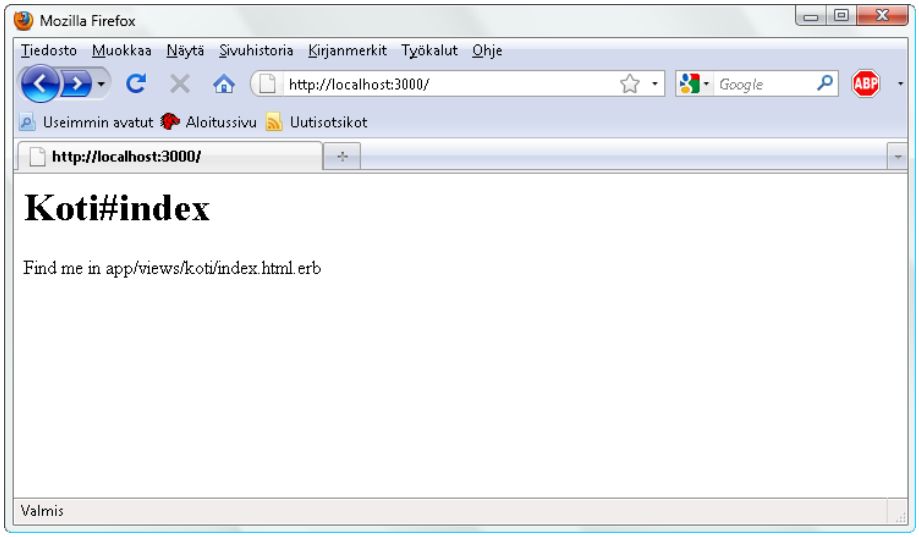
hakisi `merkinta`-ohjaimen `view`-toimintoa merkinnälle `1` tai

```
/sihijuoma/osta
```

joka taas hakisi ohjaimen `sihijuoma` toimintoa `osta`, huolimatta siitä onko sen nimistä ohjainta edes olemassa. Koska tällä kertaa ohjelmaan tarvitaan erikseen määritelty aloitussivu, lisätään tähän tiedostoon sielä valmiiksi olevien määrittelyrivien yläpuolelle rivi

```
  map.root :controller => "koti"
```

joka määrittelee sen, että ohjelman – tai tässä tapauksessa sivuston - juurisivu (root) on määritelty ohjaimeen nimeltä `koti`. Nyt Rails-sovelluksen pitäisi oletussivuna sijaan käyttää oletussivuna ohjainta



`koti`, ja sieltä näkymää `index`. Kun virtuaalipalvelin käynnistetään uudelleen, on etusivu muuttunut seuraavaan muotoon:

Ruby on Rails-aloitussivu vaihdettuna näkymän oletusruutuun

Jos Rails antaa tässä vaiheessa virheilmoituksen, johtuu se siitä että tietokantaa ei vielä ole luotu: tällöin sammutetaan palvelin ja ajetaan komento **rake db:create** sekä käynnistetään virtuaalipalvelin uudelleen.

Kuten kuvasta on mahdollista nähdä, on aloitussivu nyt korvautunut html-sivulla, joka löytyy siis projektihakemistosta osoitteesta *app/views/koti/index.html.erb* ja sisältää seuraavan tekstin:

```
<h1>Koti#index</h1>
<p>Find me in app/views/koti/index.html.erb</p>
```

Muutetaan tämän sivun teksti seuraavaan muotoon, tallennetaan ja päivitetään selaimen ikkuna:

```
<html>
<head></head>
<body>

<h1>Muistio</h1>
<p>Muistioon voi tallentaa tekstiä ja jättää huomioita niihin liittyen.</p>

</body>
</html>
```

Nyt etusivun tekstien pitäisi muuttuja äsken kirjattuun muotoon, ja muodostaa ainakin alkeellisen aloitussivun Rails-ohjelmalle.

HTML-kuvauskielen käyttämisestä on lisää tämän kirjan lopussa Liitteessä 2, jossa käydään lyhyesti läpi kuvauksien tekeminen ja esimerkiksi kuvien määrittely html-sivuille.

Tietokannan määrittely ja luominen

Tässä vaiheessa ohlemassa on siis aloitussivu, joka minimaalisesta lähestymistavastaan huolimatta antaa käyttäjälle jonkinlaisen idean siitä, mitä ohjelmassa olisi tarkoitus myöhemmin tehdä. Seuraavaksi sovellukseen tarvitaan vielä kaksi asiaa; ensinnäkin tietokanta, johon viestit tallennetaan ja toimintaympäristö, jolla viestejä voidaan kirjoittaa ja tallentaa muistioon. Näistä ensimmäisenä on luonnollisesti kannattavaa tehdä tietokanta, koska ilman sitä ei viestien tallentaminen muistikirjaan onnistu kuitenkaan.

Tietokannoista yleisesti

Ensimmäisessä esimerkissä luotiin tietokanta Rails-sivuston suorittamisen yhteydessä, mutta ei varsinaisesti puhuttu siitä, missä kyseinen tietokanta ylipäänsä määritellään. Tietokantoja koskevan tiedon varsinainen tallennuspaikka on projektihakemistossa *db*, mutta varsinainen määrittely siitä, millainen tietokanta projektissa on käytössä, on tallennettuna hakemistoon *config*, tiedostoon `database.yml`, joka oletuksena on seuraavanlainen:

```
# SQLite version 3.x
#   gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

Kuten tekstiä tulkitsemalla voidaan nähdä, määritellään tässä tiedostossa se, minkälainen tietokanta Rails-projektissa on käytössä, millä asetuksilla se toimii ja mihin tietokantatiedosto luodaan. Rails-järjestelmä osaakin käyttää useita erilaisia tietokantatyyppejä, kuten esimerkiksi SQLite, MySQL tai PostgreSQL, joilla jokaisella on omat hyöty- ja haittapuolensa. Oletuksena Rails kuitenkin luo SQLite-pohjaisen järjestelmän, joka on vaihtoehdoista yksinkertaisin ja samalla riittävä tämän kirjan projekteihin.

Lisäksi tässä tiedostossa määritellään kolme erilaista tietokantaa, `development`, `test` ja `production`. Kun aiemmassa esimerkissä Railsia pyydettiin luomaan tietokanta, se loi development-tietokannan, joka onkin tarkoitettu käytettäväksi siinä vaiheessa, kun Rails-ohjelmaa tehdään. Test-tietokanta taas on tarkoitettu käytettäväksi Rails-sivuston tekniikan testaamiseen, ja production-tietokanta siihen vaiheeseen, kun Rails-sivusto on asennettu verkkopalvelimelle ja on ns. tuotantokäytössä.

YML-tiedostopääte tarkoittaa YAML-tyyppistä tietorakennetiedostoa. YAML-tiedostot on suunniteltu erityisesti ihmisten ymmärtämäksi tietorakenneformaatiksi, ja YML-tiedostoja voikin ilman ongelmia editoida käsin esimerkiksi lähdekoodieditorilla. Lyhenne YAML tulee rekursiivisesta lyhennenimestä (YAML Ain't Markup Language).

SQLite-tietokannan määritteleminen

Rails-sivusto osaa siis käyttää erilaisia tietokantatyyppejä, ja käyttää SQLite-tietokantaa oletusarvonaan. Tämä sopii myös tähän projektiin erinomaisesti, joten tietokantamäärittelyjä ei tarvitse tässä vaiheessa ryhtyä muuttamaan. Lisäksi, koska SQLite-tietokanta luodaan yhteen yksittäiseen tiedostoon jota Rails-järjestelmä käyttää suoraan itse, ei asetuksiin tarvitse tehdä muutoksia tai määritellä tunnuksia saatikka sitten käytettävän tietokannan nimeä. Luodaan siis development-tietokanta antamalla komento `rake db:create`, joka tulostaa kutakuinkin seuraavan vastauksen:

```
C:\Railsohjelmat\muistio>rake db:create
(in C:/Railsohjelmat/muistio)

C:\Railsohjelmat\muistio>
```

Jos kaikki toimii oikein ei rake-käsky tulosta mitään ilmoitusta. Lisäksi, jos nyt tarkastellaan projektikansiota *db*\, on sinne luotu tiedosto nimeltä `development.sqlite3`, jossa ohjelman kehityksenaikainen tietokanta nyt fyysisesti sijaitsee. Tämän tiedoston tuhoaminen poistaa koko tietokannan kerralla, ja pakottaa siis luomaan tietokannan uudelleen, mutta on samalla myös näppärä toimenpide jos ohjelmaan tehdyt muutokset tekevät siitä yhteensopimattoman tietokannan kanssa tai aiheuttavat virheitä.

Muut tietokantatyypit

SQLite on näppärä pieni tietokantatyyppi, mutta sen ongelmana on juuri se, että tietokanta on yksi tiedosto, johon kirjoitaminen ja josta lukeminen onnistuu yhtäaikaisesti onnistuu ainoastaan muutamalla yhteydellä siedettävässä ajassa. Lisäksi monesti isommat ohjelmistosivustot sisältävät täysin erilliset tietokantapalvelut ja –palvelimet, joten myös muunlaisista tietokantayhteyksistä on hyvä mainita muutamalla sanalla.

Ensinnäkin, jos Rails-projektissa halutaan käyttää jotain muuta tietokantatyyppiä kuin SQLite, ilmoitetaan siitä projektia luotaessa antamalla lisäargumenttina tietokantatyyppi muodossa

```
rails <projektinimi> -d <tietokantatyyppi>
```

Eli vaikkapa MySQL:ää käyttäessä

```
rails uusiprojekti –d mysql
```

Vaihtoehtoina voidaan antaa `mysql`, `oracle`, `postgresql`, `sqlite2`, `sqlite3` (joka on samalla oletusarvo), `frontbase` tai `ibm_db`. Tämä samalla myös muuttaa database.yml-tiedoston sisältöä, ja vaatii käytettävästä tietokantatyypistä riippuen jonkin verran asetuksien ja käyttöoikeuksien asettelua tietokantaan. Esimerkiksi käytettäessä MySQL-tietokantaa on oletustietokanta-asetukset kehitystietokannalle ovat seuraavanlaiset:

```
development:
adapter: mysql
encoding: utf8
database: blog_development
pool: 5
username: root
password:
socket: /tmp/mysql.sock
```

Eli tietokantaa käyttääkseen pitää myös määritellä missä MySQL-tietokantapalvelin sijaitsee, mitä tietokantaa sieltä käytetään, sekä tunnus ja salasana tietokantatietoihin käsiksi pääsemiseen. Vastaavia asetuksia joutuu myös tekemään muille tietokantatyypeille, sen lisäksi että itse tietokantapalveluun pitää myös olla yhteys Rails-järjestelmästä. Toisin kuin SQLite, joka asentuu Rails-järjestelmän sisään, on vaikkapa MySQL isokokoinen, erillinen, itsenäinen ohjelmisto joka vaihtaa viestejä Rails-järjestelmän kanssa. Kuitenkin, kun tietokantaasetukset on määritelty oikein `database.yml`-tiedostoon, käytetään `rake`-komentoja tietokannan käsittelyyn samalla tavalla kuin SQLiten kanssa.

Käyttöliittymän tekeminen, scaffold

Kun tietokanta on luotu ja määritelty, on aika ryhtyä toteuttamaan itse muistikirjan käyttöliittymää. Käyttöliittymä toteuttaminen on verkko-ohjelmoinnissa periaatteessa seuraavanlainen prosessi; suunnitellaan sivujen layout eli asettelu, määritellään mitä alisivuja, tiedonsyöttölomakkeita ja näkymiä sivuston toimintaan tarvitaan, ja mistä lomakkeesta siirrytään mihinkin ja missä vaiheessa. Eli periaatteessa mitä näkymiä sivustolla tarvitaan, ja mitä toimintoja ohjaimiin pitää suunnitella.

Koska kyseessä on kuitenkin ensimmäinen Rails-sivusto jolla on tarkoitus tehdä myös jotain järkevää, ei ole kannattavaa aloittaa täysin tyhjästä. Rails-paketin mukana tulee näet scaffolding-niminen käyttöliittymägeneraattori, jolla on mahdollista luoda nopeasti jotain käynnistyvää ja toimivaa. Scaffolding-toimintojen avulla saadaan kohtuullisen yksinkertaisesti aikaiseksi blogi- tai muistikirja-tyylisiä ohjelmia, joissa on mahdollisuus luoda, lukea, päivittää tai poistaa viestejä kuitenkaan joutumatta erityisen paljon ohjelmoimaan. Lisäksi Scaffolding-järjestelmän avulla voidaan lisäksi generaattorin luomaa koodia tarkastelemalla tarkemmin tutustua siihen miten Rails-sovellukset toimivat oikeasti. Scaffoldin toimii komennolla

```
ruby script/generate scaffold < nimi> <tietotyypit>
```

Eli vaikkapa

```
ruby script/generate scaffold Merkinta otsikko:string teksti:text
```

Tällä komennolla pyydetään Rails-ohjelmaa luomaan Merkinta-niminen näkymä- ja ohjainpari, jolla pystytään käsittelemään tietoja, joissa on yhden merkkirivin mittainen otsikko-kenttä ja teksti-niminen vapaa tekstikenttä. Scaffolding luo toimintansa perusteella joukon tiedostoja projektihakemistoon:

- *app\controllers*-hakemistoon luodaan *merkintas_controller.rb*, johon on kirjoitettu käytettävät metodit, joilla lisätään, tarkastellaan ja esimerkiksi poistetaan syötteitä.
- *app\views*-hakemistoon on luotu alihakemisto *merkintas*, jossa on määriteltynä jokaista toimintoa vastaava näkymä.
- *app\views\layouts* –hakemistoon luodaan *merkintas.html.erb*, joka sisältää yhteiset muotoilutiedot ja html doctype-määritelmät kaikille `merkinta`-ohjaimen näkymille.
- *app\models* -hakemistoon luodaan *merkinta.rb*, joka sisältää mallia koskevat tiedot. Oletuksena tiedosto on tyhjä, mutta siihen voidaan esimerkiksi määritellä myöhemmin kuinka mallit toimivat keskenään.
- *db\migrate\xxxxxxxxxxxxxxcreate_merkintas.rb*, jossa xxxx…-merkintä on lukusarja joka kuvaa tiedoston luomishetkeä. Tähän tiedostoon määritellään tietokannan rakenne, joka otetaan käyttöön kun suoritetaan `rake db:migrate`-käsky. Jokainen muutos tietokantaan luo uuden migrate-tiedoston, ja jokainen muutos pitää määrätä päivittämään tietokantamalliin erikseen.

Lista ei ole täysin kattava, mutta sisältää keskeisimmät scaffolding-toiminnon luomat tiedostot. Avataan seuraavaksi tiedosto *app\views\merkintas\index.html.erb* lähdekoodieditoriin ja tarkastellaan mitä se pitää sisällään:

```
<h1>Listing merkintas</h1>

<table>
  <tr>
    <th>Otsikko</th>
    <th>Teksti</th>
  </tr>

<% @merkintas.each do |merkinta| %>
  <tr>
```

```
    <td><%=h merkinta.otsikko %></td>
    <td><%=h merkinta.teksti %></td>
    <td><%= link_to 'Show', merkinta %></td>
    <td><%= link_to 'Edit', edit_merkinta_path(merkinta) %></td>
    <td><%= link_to 'Destroy', merkinta, :confirm => 'Are you sure?', :method =>
:delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New merkinta', new_merkinta_path %> <br/>
```

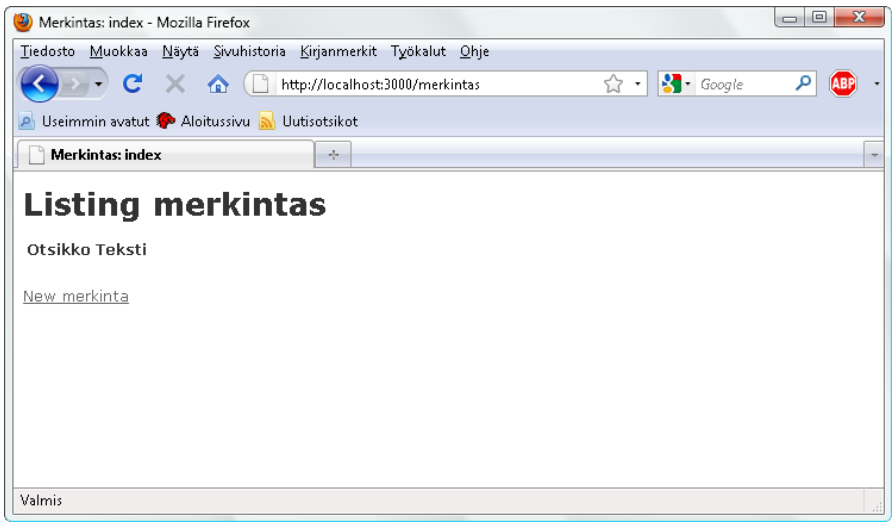
Tällä sivulla määritellään siis viestien päävalikon toiminta ja yleinen muotoilu. Tiedosto näyttää pitkälti normaalilta html-kuvaukselta, mutta esimerkiksi sivulle määritellyssä taulukossa sekä toimintalinkeissä nähdään joitain Rails-kielen komentoja ja syntaksia. Keskeistä kuvasta onkin ymmärtää se, että Rails-tietokannasta tuotavat osat sijoitellaan näkymiin yksinkertaisilla Rails-komennoilla. Lisäksi tämän – ja muiden scaffoldingin luomien näkymä-sivujen – tekstikenttiä ja html-elementtejä muuttelemalla voidaan vapaasti muuttaa sivun muotoilua, koska näkymä on käytännössä html-kuvausta jossa on seassa Rails-kenttiä. Lisäksi scaffolding-toiminnon luomat tekstit esimerkiksi `link_to`-komennoissa voidaan muuttaa esimerkiksi suomenkielisiksi tai lisätä linkkejä tai vaihtoehtoisesti vaikkapa muuttaa taulukon muotoilutapaa.
Tässä vaiheessa voidaan kuitenkin testata miltä sivusto näyttää. Sitä ennen tehdään kuitenkin vielä tietokantamallin päivitys (*migration*), koska tietokantamalli on muuttunut scaffolding-toiminnon käytön myötä. Tämä tapahtuu käskyllä

```
rake db:migrate
```

Nyt kun tietokantakin on saatettu ajantasalle, voidaan virtuaalipalvelin käynnistetää käskyllä

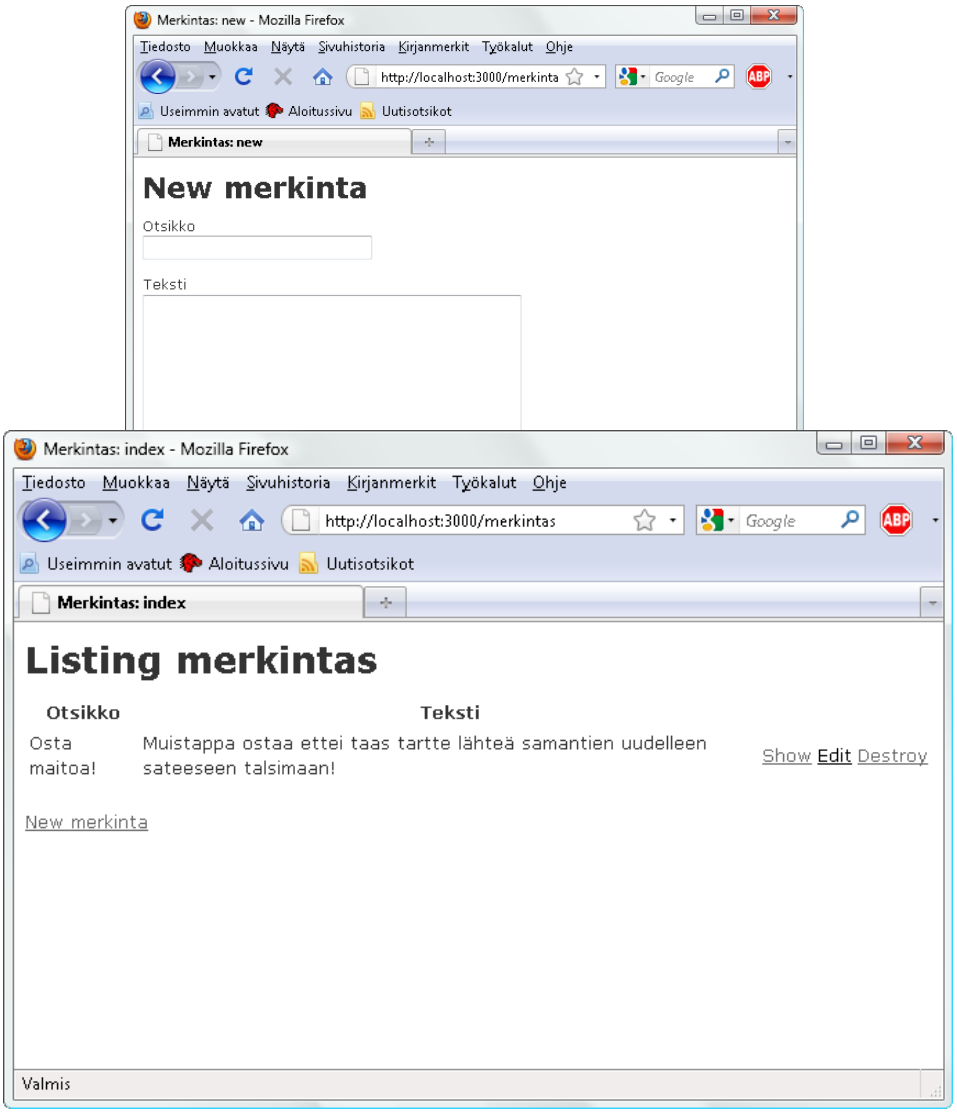
```
ruby script\server
```

Jos kaikki menee oikein, on Rails-sivun aloitussivu edelleen samanlainen kuin aikaisemmassa esimerkissä. Siirrytään nyt sivulle *http://localhost:3000/merkintas* jossa varsinainen muistikirja-ohjelma sijaitsee. Jos kaikki toimii oikein, pitäisi ruudulla olla seuraavanlainen ikkuna:



Scaffolding-generaattorilla luotu muistikirjan aloitussivu.

Ohjelma ei vielä näytä kovin ihmeelliseltä, mutta se on minimalistisesta ulkoasustaan huolimatta täysin toimiva ohjelma! Jos esimerkiksi kokeillaan lisätä uusi merkintä muistioon, saadaan näkyviin seuraava lomake:



Merkintöjen syöttöön tarkoitettu lomake: kentät tulevat suoraan scaffolding-generaattorin kutsussa annetuista lomaketiedoista

Kirjoitetaan tähän kenttään jokin lyhyt viesti, kuten ”Hae maitoa!” tai vastaavaa, ja tallennetaan viesti sekä palataan pääikkunaan painamalla esikatseluruudussa valintaa ”back”. Nyt viesti on tallennettu tietokantaan ja sen pitäisi näkyä muistion pää-sivulla:

Railsilla luotu muistio, jossa on yksi merkintä.

OMIEN TOIMINTOJEN LISÄÄMINEN

Periaatteessa muistikirja-ohjelma olisi nyt valmis; ohjelmassa on mahdollisuus lukea, lisätä, muokata ja poistaa merkintöjä verkossa toimivaan muistikirjaan. Tehdään muistikirjaan kuitenkin vielä jotain päivityksiä ja viimeistelyjä, koska tähän mennessä ei vielä suoranaisesti ole päästy koodaamaan oikeastaan mitään.

Linkkien lisääminen, link_to

Ensimmäinen epäkohta, joka tulee nopeasti ilmeiseksi, on se, että ohjelmassa on erillinen etusivu ja muistikirjan aloitussivu mutta ei mahdollisuutta siirtyä näiden käsin sivun välillä. Luonnollisesti tämä on helpointa korjata lisäämällä etusivulle linkki josta päästään siirtymään itse muistikirjaan, ja muistikirjaan linkki jolla voidaan palata takaisin etusivulle.

Aikaisemmin tarkasteltiin tiedostoa *app\views\merkintas\index.html.erb* ja todettiin, että tämä tiedosto on se, johon scaffolding on määritellyt itse muistiinpanojen käyttöliittymän. Vastaavasti tiedetään, että muistion pääsivuna toimii ohjaimen `koti` sisältämä `index`-näkymä joka on määritelty tiedostoon *app\views\koti\index.html.erb*. Näihin kahteen tiedostoon voidaan lisätä linkit toisiinsa Rails-kehyksen link_to-käskyllä, joka toimii seuraavasti:

```
<%= link_to ”<linkin teksti>”, <metodin kutsu> %>
```

jos kyseessä on ohjaimensisäinen linkki eri näkymään, tai

```
<%= link_to ”<teksti>”, :controller => <ohjaimen nimi>, :action => <metodin kutsu> %>
```

jos kyseessä on samalla siirtyminen ohjaimesta toiseen. Lisäksi linkin osoitteeksi voidaan antaa myös RESTful-malleilla nimetty polku, mutta siihen palataan myöhemmin.

Joka tapauksessa, koska kummassakaan tapauksessa ei pysytä saman ohjaimen sisällä, käytetään pitkää kaavaa kummassakin linkissä, eli joudutaan linkin tekemisen yhteydessä määrittelemään ohjain ja toiminto (näkymä) erikseen. Tämä onnistuu seuraavasti: tiedostoon *app\views\merkintas\index.html.erb* voidaan loppuun lisätä käsky

```
<%= link_to "Takaisin alkuun", :controller => "koti", :action => "index" %>
```

joka luo muistikirjan päävalikkoon linkin takaisin etusivulle. Vastaavasti etusivulle voidaan kirjoittaa linkki

```
<%= link_to "Lue muistiota", :controller => "Merkintas", :action => "index" %>
```

joka taas luo linkin etusivulta muistikirjan varsinaiseen päävalikkoon. Kaikenkaikkiaan, `link_to` onkin varsin tehokas työväline luotaessa linkkejä näkymästä toiseen; linkki määritellään ajonaikaisesti eikä linkin osoite voi täten olla vanhentunut tai osoittaa väärään paikkaan, niin kuin saattaa usein käydä kun toimintoja siirretään esimerkiksi palvelimelta toiselle ja linkkien osoitteet määritellään käsin.

Scaffold-kuvauksen muuttaminen

Rails-ohjelman luominen scaffolding-toiminnon avulla on nopeaa ja sen avulla onkin erinomaisen helppoa saada yksinkertainen tietokanta- ja viestinkäsittelyjärjestelmä aikaiseksi. Mutta entäpä jos alkuperäinen suunnitelma todettiinkin vajanaiseksi? Esimerkin muistikirjassa voisi esimerkiksi olla hyödyllistä pystyä tallentamaan ylös tila siitä, mitkä tehtävät on kesken, mitkä valmiit tai ylipäänsä aloittamatta. Lisäksi, jos muistikirjaa käyttää useampi henkilö, olisi varmaan kohtuullista lisätä kenttä jossa on tieto siitä, kuka viestin on jättänyt.

Tämä on kuitenkin kohtuullisen helppoa toteuttaa käytettäessä scaffold-generaattoria. Jos scaffold-järjestelmällä luotua kuvausta halutaan muuttaa, poistetaan ensin projektitietokansiosta vanha tietokanta. Tämä voidaan toteuttaa helpoiten deletoimalla tietokantatiedosto (oletuksena siis *db\development.sqlite3*) ja db-hakemistossa oleva migrate-kansio, johon on tallennettuna vanhan tietokannan muutoksia koskevat tiedot. Vaikka migrate-toiminto periaatteessa mahdollistaa myös pelkän tietokannan päivittämisen esimerkiksi lisäämällä sarakkeita tauluun, voi sinne aiemmin tallennetut tietueet aiheuttaa ongelmia koska niistä puuttuu uudet tiedot. Lisäksi tässä vaiheessa projektia ei vielä pitäisi olla tarvetta huolehtia siitä, että menettää jotain oikeasti tärkeää, koska työ on edelleen kehitysvaiheessa. Helpointa onkin poistaa koko tietokantatiedosto ja korvata se uudella. Jos kyseessä olisi oikeassa käytössä oleva kanta tai järjestelmä jonka tietoja ei saa menettää, voitaisi päivitystyö tehdä migrate-työkalujen avulla.

Kun tietokantatiedosto ja siihen liittyneet migrate-tiedostot on poistettu, voidaan jatkaa eteenpäin. Seuraavaksi toimitaan kuten aikaisemmin; luodaan ensin uusi tietokanta esimerkiksi komennolla

```
rake db:create
```

ja tämän perään määritellään uusi scaffold-toiminnolla tehtävä käyttöliittymä, johon muutokset on huomioitu, esimerkiksi

```
ruby script/generate scaffold Merkinta aihe:string selite:text tila:string
kirjoittaja:string
```

Kannattaa kuitenkin huomata, että jos käytetään samannimistä ohjainta kuin jo aikaisemmin määritelty, näyttää tulos kutakuinkin tällaiselta:

```
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
     …
overwrite app/views/merkintas/index.html.erb? (enter "h" for help) [Ynaqdh]
```

Tässä vaiheessa Rails pyytää siis lupaa ylikirjoittaa aiemmin määritellyt scaffoldingilla luodut tiedostot. Valinta Y sallii nimenomaisen tiedoston ylikirjoittamisen, n kieltää tiedoston ylikirjoituksen ja a sallii kaikkien tiedoston ylikirjoituksen. Tässä kohdassa voidaan siis valita 'a', jolloin vanhat ohjaintiedostot korvautuvat uusilla. Lopuksi suoritetaan vielä tietokannan päivitys migrate-käskyllä

```
rake db:migrate
```

Jos Rails-ohjelma käynnistetään tämän jälkeen uudelleen, on siinä olevat kentät ja muut lomakkeet päivittyneet vastaamaan uusia asetuksia. Tietenkin pitää huomata, että `scaffold` täysin korvaa aiemmat tiedostot, eikä ainoastaan täydennä ja päivitä näkymä- ja lomaketietoja. Jos lomakkeisiin on tehty kattavia muutoksia, kannattaa vanhat näkymien lähdekooditiedostot ottaa talteen jotta niihin tehdy asettelumuutokset voidaan päivittää uusiin näkymiin myöhemmin käsin.

Tyyliohjeen lisääminen näkymiin

Muistiossa on kuitenkin vielä eräs varsin harmillinen ongelma; sen käyttämä fontti ja tapa rivittää tekstiä vaihtuvat etusivulta muistikirjan päävalikkoon siirryttäessa varsin ikävästi. Tähänkin ongelmaan on kuitenkin Rails-ympäristössä olemassa helppo ratkaisu; tyyliohjeen käyttöönotto.

Aikaisemmin mainittiinkin, että scaffolding-toiminto luo automaattisesti projektihakemistoon `app\views\layout` oman määrittelynsä merkintas-ohjaimen kaikkien näkymien käyttämälleen tyylirakenteelle sekä html-määritteille. Tämän lisäksi se kuintenkin myös luo kansioon `public\stylesheets\ scaffold.css`-nimisen CSS-tyyliohjeen, jossa määritellään mitä fonttia, rivitystä tai esimerkiksi taustaväriä missäkin html-elementissä käytetään. Tietysti paras tapa olisi luoda hakemistoon `app\views\layout` kokonaan uusi tyylimääritelmä `koti`-ohjaimelle samalla tapaa kuin `merkintas`-ohjaimella on, mutta koska kyseessä on yksi ainoa näkymä, voidaan `stylesheet_link_tag`-käskyllä määritellä se, mitä CSS-tyyliohjetta näkymän muodostamisessa käytetään. Komento `stylesheet_link_tag` toimii muodossa

```
<%= stylesheet_link_tag "<css-tiedosto public/-hakemistossa>" %>
```

Eli siis vaikkapa

```
<%= stylesheet_link_tag "/stylesheets/scaffold.css" %>
```

Tämä komento voidaan siis lisätä näkymätiedoston `koti\index.html.erb` ensimmäiseksi riviksi, ja määrätä Rails-projekti käyttämään `scaffold.css`-tyyliohjetta näkymän muotoilussa. Luonnollisesti tämä myös tarkoittaa sitä, että hakemistoon `public\stylesheet\` sijoitetut tyyliohjeet ovat minkä tahansa näkymän käytettävissä `stylesheet_link_tag`-käskyn avulla. Tietysti oman layout-tiedoston luominen on pitkällä aikavälillä järkevämpää, varsinkin jos halutaan että Rails-sovellus täyttää erinäisten html-validaattorien asettamat ehdot meta- ja määrittelytiedoille.

Tyyliohjeista, eli CSS-tyyliohjeista on lisää asiaa liitteessä 2, missä mm. neuvotaan kuinka uusia CSS-ohjeita luodaan.

Kuvien lisääminen näkymiin

Tietenkin verkkosivuilla on myös tavallisesti muutakin sisältöä kuin pelkästään tekstiä ja tekstikenttiä, toisin kuin tähänastisissa esimerkeissä. Siksi tässä vaiheessa onkin hyvä ottaa esimerkki siitä, kuinka Rails-näkymään voidaan lisätä kuva. Ajatuksena olisi lisätä muistion etusivulle otsikkokuva, joka on seuraavan näköinen:

Muistiota varten suunniteltu otsikkokuva nimeltään otsake.png

Toisin kuin tavallisessa HTML-kuvauksessa, ei Rails-projekteissa kuvamateriaalia tarvitse sijoittaa samaan, tai staattisesti määrättyyn osoitteeseen, vaan Rails-projektikansiossa on niille yksi keskitetty sijainti, josta ne voidaan helposti poimia käyttöön samaan tapaan kuin aikaisemmin käytetty tyyliohjeen määrittely. Tämä sijainti on projektihakemistossa oleva kansio nimeltä *public\images*. Tähän kansioon tallennettuihin kuviin voidaan viitata yhdellä lyhyellä Rails-käskyllä, joka menee seuraavasti:

```
<%= image_tag "<kuvatiedostonnimi>", :alt=>"<vaihtoehtoinen teksti>"%>
```

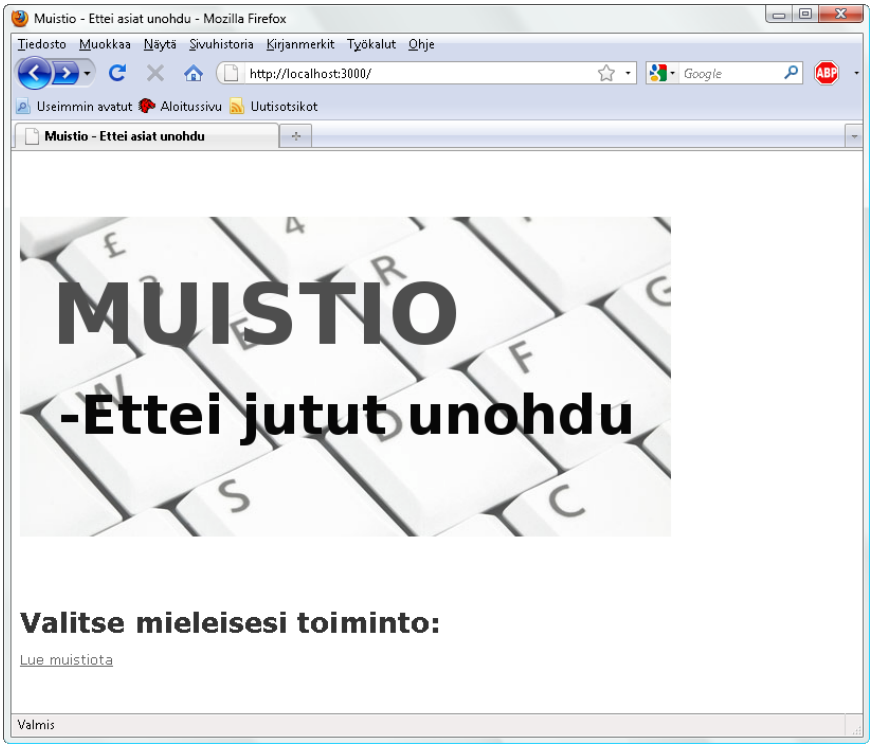
Eli jos kuva on tallennettu nimellä otsake.png, muodossa

```
<%= image_tag "otsake.png", :alt=>"Muistio"%>
```

Muokataan projektin kotisivua, eli näkymän koti sivua index nyt siten, että se on muodossa

```
<html>
<head>
<title>Muistio - Ettei asiat unohdu</title>
</head>

<body>
<%= stylesheet_link_tag "/stylesheets/scaffold.css" %>
<%= image_tag "otsake.png", :alt=>"Muistio"%>
<br/>
<h1>Valitse mieleisesi toiminto:</h1><p/>
<%= link_to "Lue muistiota", merkintas_path %>
</body>
</html>
```

Eli sisältämään pelkästään otsikkokuva, rivityksen vaihto br sekä linkki muistion ohjaimeen merkinta. Etusivun tulisi nyt näyttää tältä:



Muistion uusi etusivu

`image_tag`-käskyn avulla voidaan siis helposti lisätä kuvia projektin näkymiin sen `public/images`-kansiosta. image_tag-komentoon voidaan myös lisätä kuvan kokoa rajoittavia lisäkäskyjä, kuten `:size`, esimerkiksi `:size = "100x100"`, joka pakottaisi kuvan skaalautumaan sata kertaa sata pikseliä kokoiseen tilaan, tai `:mouseover`, joka määrittelee vaihtokuvan joka tulee alkuperäisen kuvan paikalle kun hiiren kursori vedetään sen päälle. Näistä asioista puhutaan kuitenkin enemmän seuraavassa luvussa, kun Rails-kehyksellä luodaan ensimmäinen kokonainen alusta asti itsetehty ohjelma.

YHTEENVETO PROJEKTISTA

Mitä tässä projektissa sitten opittiin? Vaikka työssä ei suoranaisesti vielä ohjelmoitu Rails-kielellä muutamaa käskyä enempää, voidaan ensimmäisestä projektista tehdä jotain huomioita. Ensinnäkin, eräs Ruby on Railsin perusajatuksistahan on *Convetion over Configuration,* vapaasti käännettynä ”Tavanmukaisuus ennen virittelyä”. Tänä kannattaakin ottaa Rubyssa tosissaan; Railsin generaattorityökaluilla saa aikaiseksi pitkälti sellaisia elementtejä, joilla sivuston kokoaminen onnistuu hetkessä, mutta oikeiden osien löytämiseen voi mennä päiviä, varsinkin jos osia otetaan myös esimerkiksi jälkiasennuksena RubyGems-puolelta, mistä löytyy mm. kokonaisia kirjautumisen- ja istunnonhallintaelementtejä Rails-sivustoilla käytettäväksi. Tämä työ on usein kuitenkin hyödyllistä, koska ”virittelemällä” toteutettujen toimintojen kanssa saattaa helposti tuhlautua viikkokin ylimääräistä aikaa.

Lisäksi toinen perusajatus, *Don’t repeat yourself,* eli ”Älä toista itseäsi” on myös selkeästi mukana jo tässä projektissa. Näkymiin tulevat komponentit, kuten kuvat, layout-tiedot ja tyyliohjeet määritellään yhteen paikkaan, ja niitä käytetään sieltä käsin. Samoin sivujen sijainti- ja reititystiedot, tai tietokannan käyttämiseen tarvittava data on määritelty yhteen paikkaan, mistä käsin niitä muutetaan kerralla koko projektille. Tietokantojen tapauksessa tämä nyt tietysti ei ole SQLite-kantaa käytettäessä kovin iso asia, mutta esimerkiksi ulkoiseen tietokantaan kytkeydyttäessä istuntotietojen tallentaminen yhteen paikkaan on varsin järkevää.

Mitä ensimmäisessä esimerkissä sitten ei käsitelty? Tietenkin perusasioita, joita ei ollenkaan kokeiltu tai toteuttu on edelleen paljon, mutta niistä kaikista keskeisin on varmaankin syötettävien tietojen tarkastaminen ja mallien käyttäminen tietojen yhdistämiseen. Esimerkissä myös otettiin jonkin verran erivapauksia mm. tyylitietojen asettelun ja joidenkin teknisten yksityiskohtien suhteen. Teknisessä mielessä itse verkkosivujen kokoaminen jäikin esimerkissä melko pitkälti automaattisten työvälineiden

varaan ja siten vähälle huomiolle.. Näistä asioista puhutaankin seuraavassa luvussa, missä tehdään toisena Rails-projektina verkkosivut, joilla on uutispalsta johon lukijat voivat jättää kommentteja.

LUKU 12

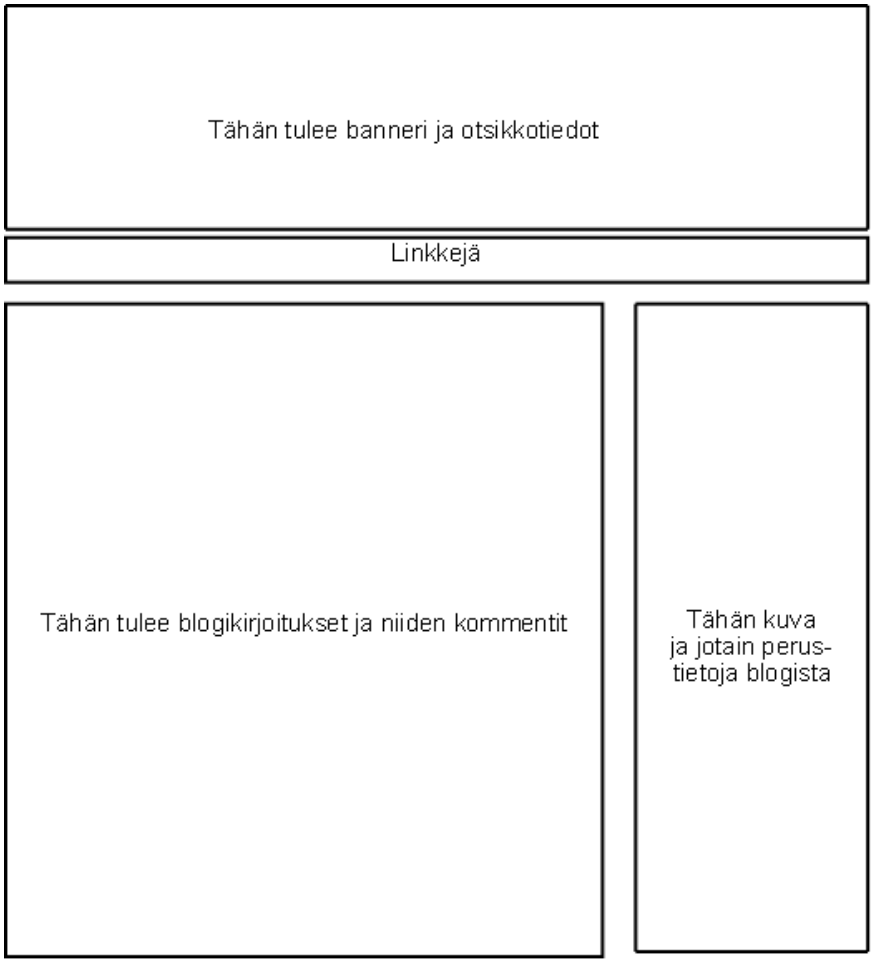
Tiedon määrittely ja käsittely, blogin tekeminen

Projektin luominen
Perussivujen kokoaminen
Tietomallien määrittely
Näkymien ja kenttien luominen
Toisen mallin lisääminen
Projektin viimeistely
Yhteenveto projektista

Tässä luvussa tarkastellaan tietokantojen ja tietomallien käyttämistä on Rails-projektissa, sekä harjoitellaan käsin toteutettavan Rails-sivuston tekemistä. Luvussa tehdään tämän kirjan varsinainen Rails-pääprojekti, jossa yleisimpiä Rails-kehyksen toiminnallisuuksia hyödyntäen toteutetaan verkkoblogi.

PROJEKTIN LUOMINEN

Kirjan toisessa Rails-projektissa on siis tarkoituksena luoda yhden verkkosivun päälle toteutettu palvelu, johon sivun omistaja voi jättää viestejä sivun pääsivulle, ja muut lukijat pystyvät kommentoimaan sivulle jätettyjä juttuja. Käytännössä tarkoituksena on siis koota yksinkertainen blogi, jossa eri käyttäjille tarjotaan erilaisia palveluita; sivun omistaja tai henkilö, jolla on hallussaan salasana, voi kirjoittaa sivulle uusia juttuja, kun taas kommentointi on kaikkien käyttäjien käytettävissä. Tarkoituksena olisi luoda kutakuinkin seuraavaa luonnosta vastaava sivu:



Luonnos verkkosivun layoutille

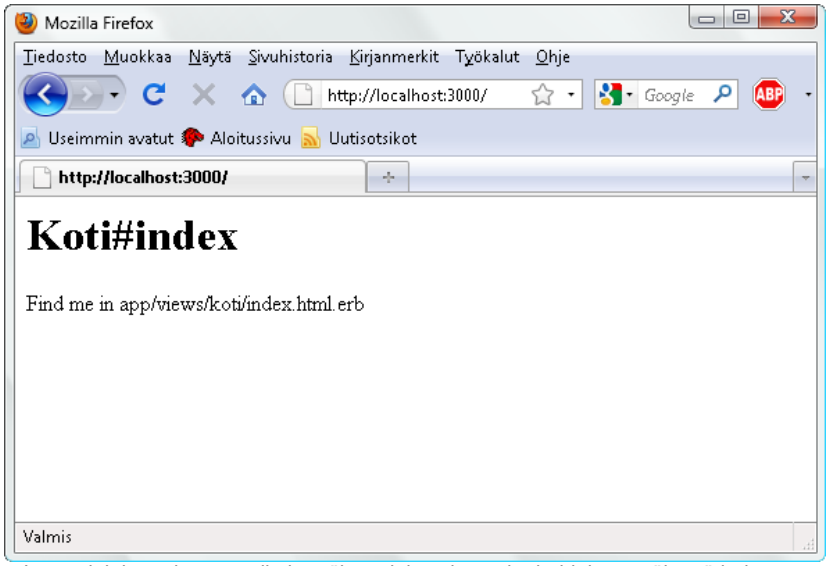
Lisäksi sivuille luodaan alisivu, jota sivun omistaja voi käyttää juttujen lisäämiseen pääsivulle. Koska myös halutaan, että sivut eivät täyty tyhjillä kommenteilla, vaaditaan kommenttikenttään tarkastustoiminto, jolla viestiä ei voi jättää ilman että kommentti- ja kirjoittajakenttä on täytetty. Lisäksi halutaan että kommentti voi saa olla korkeintaan 120 merkkiä pitkä. Samoin varsinaisia uusia viestejä varten halutaan toteuttaa jonkinlainen salasanavarmennus, jolla estetään viestien kirjoittelu aivan keneltä tahansa sekä lisätä viestien lisäämisen ajankohta näkyviin sivulle.

PERUSSIVUJEN KOKOAMINEN

Tähän mennessä kirjassa on läpikäyty uuden projektin luominen jo kahteen otteeseen, joten tällä kertaa käydään tapahtuma läpi lyhyesti seuraavan listan avulla.

- Luodaan uusi projekti nimeltä `blogi`.
- Määritellään projekti käyttämään SQLite-tietokantaa ja luodaan tietokanta oletusasetuksilla.
- Luodaan ohjain `viestis`, ja siihen toiminto `index`, joka asetetaan `config/routes.rb`-tiedostosta projektin kotisivuksi lisäämällä sinne rivi `map.root :controller => "viestis"`.
- Poistetaan oletusaloitussivu *public\index.html*, jotta itsemääritelty kotisivu näkyy oikein.

Jos kaikki on sujunut ilman ongelmia, käynnistyy projektin aloitussivuksi (*http://localhost:3000/*) seuraavanlainen ikkuna:



Jos projekti on alustettu oikein, näkyy aloitussivuna koti-ohjaimen näkymä index

Tässä vaiheessa uusi projekti on saatu aloitettua siten, että siihen voidaan määritellä aloitussivun perusasettelu tiedostoon *app\views\viestis\index.html.erb*. Kuten edellisessä projektissa muistiossa, jaetaan tässä projektissa tyylimäärittelyyn kuuluvat tiedot kolmeen tiedostoon:

- *app\views\viestis\index.html.erb* sisältää itse etusivun asettelun, eli luonnoksen sisältämät kentät muodostettuna html:n table-elementeistä.
- *app\views\layouts* -kansioon määritellään tiedosto *viestis.html.erb*, johon määritellään html-headeri, johon tulee sivuston metatiedot, kuten kirjasinasetukset ja käytettävän html-kuvauksen versiotiedot.
- *public\stylesheets* -kansioon määritellään *tyyliohje.css*, jossa määritellään kuinka sivuston eri html-elementit on tarkoitus selaimessa asetella ja mitä värejä ja fontteja niissä käytetään.

Projektin kannalta on tärkeää, että käytettävän ohjaimen nimi päättyy ”s”-kirjaimeen. Tämä vaatimus tulee Rails-kehyksen tavasta tarjota automaattisia palveluita tietynnimisille ohjaimille (messages, books, cars, users). Antamalla sopiva nimi Rails ymmärtää ohjaimen käyttävän RESTful-mallia ja tarjoaa sille CRUD-perustoiminnot, kuten uusien tietueiden lisäämisen, muokkaamisen ja selailemisen ilman, että jokainen palvelu joudutaan erikseen määrittelemään mm. routes.rb-tiedostoon.

Edellisessä projektissa tämä tehtiin automaattisesti scaffolding-toiminnolla, tällä kertaa toteutetaan vastaavat toiminnot käsin ja jatketaan tältä tasolta vielä vähänmatkaa eteenpäin. Ensimmäisenä tehtävänä voidaan toteuttaa itse html-elementtien asettelu, koska se on varsin suoraviivainen tehtävä.

HTML-sivun rungon tekeminen

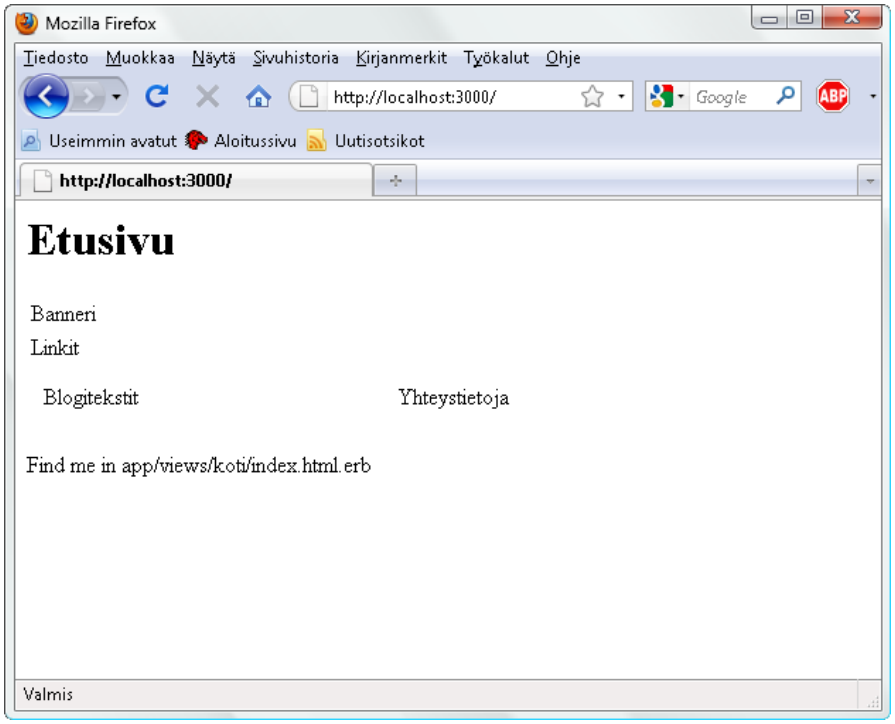
Ensimmäinen asia, mitä projektiin tässä vaiheessa tarvitaan, on pääsivun html- pohja. Jos sivua ei ole määritelty, ei sivulle pystytä siinä tapauksessa määrittelemään Rails-elementtejä, eikä kirjoitettujen osien toimintaa pystyttäisi muutenkaan näkemään ilman toimivaa näkymää. Tämän lisäksi pelkän html-kuvauksen avulla päästään jo alkuun; toiminnallisuudet on helpointa liittää valmiiseen sivuston runkoon siinä vaiheessa kun ne on saatu toimimaan.

Jos tarkastellaan pääsivun asettelua, voidaan sen todeta olevan helppo toteuttaa html-taulujen avulla. Ensin määritellään taulu, jossa on kaksi riviä, joista ensimmäisellä on sivustolle tarkoitettu otsikkokuva, ja sen alla rivi erillisille linkeille. Tämän alapuolelle voidaan määritellä toinen taulu, jossa on yhdellä rivillä kaksi rinnakkaista solua. Niistä ensimmäiseen tulee blogikirjoitukset Rails-elementillä toteutettuna, ja toiseen tietoja blogin kirjoittajasta ja esimerkiksi yhteystietoja kuten sähköpostiosoite tai linkki Facebook-profiiliin.

Itse sivun asettelua koskeva html-runko-osa voidaan siis jo tässä vaiheessa toteuttaa tiedostoon *app\views\viestis\index.html.erb* html-kuvauksena, ja sen pitäisi näyttää koodina kutakuinkin tältä:

```
<h1>Etusivu</h1>

<table borderwidth=0 cellspacing =5 width = "800">
  <tr>
  <td>Banneri </td>
  </tr>
  <tr>
  <td>Linkit </td>
  </tr>
</table>

<table width= "800">
  <tr>
  <td>Blogitekstit</td>
  <td>Yhteystietoja</td>
  </tr>
</table>
<p>Find me in app/views/viestis/index.html.erb</p>
```

Kannattaa muistaa, että tähän tiedostoon kuvaillaan ainoastaan html-sivun runko; muut osat kuten tyylimääritteet tai pääosa on tarkoitus määritellä sivun näkymien yhteiseen app\views\layout-hakemiston tiedostoon viestis.html.erb. Jos taulukointi on tehty oikein, näyttää projektin pääsivu virtuaalipalvelimella kutakuinkin tältä:



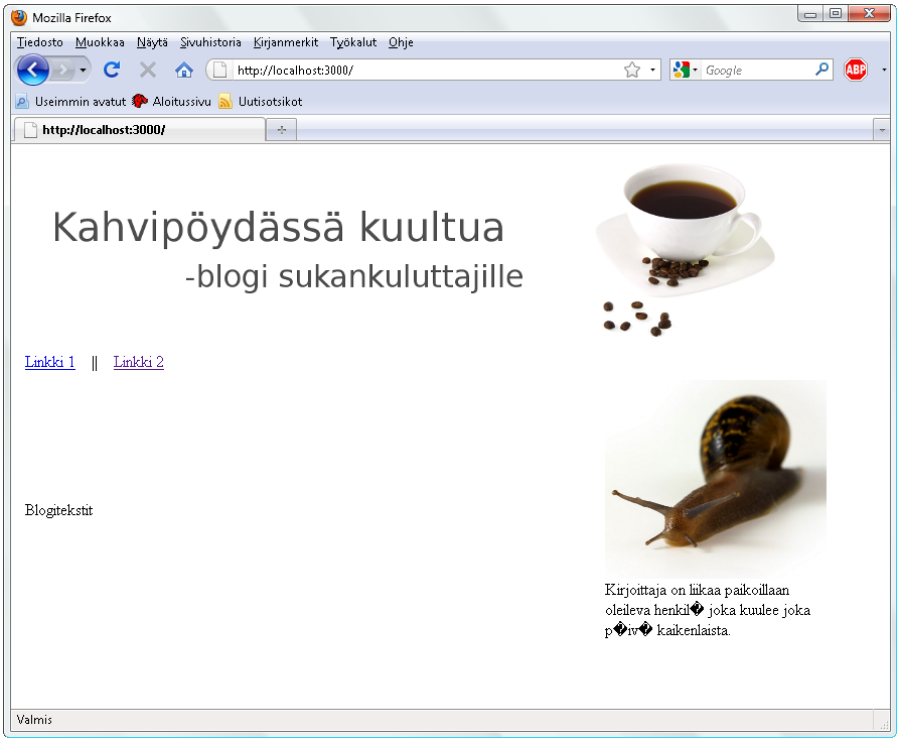
Verkkoblogin pääsivun runko, tekstikentät ovat kutakuinkin keskenään sijoiteltu samoin kuin suunnitelmassa.

Etusivu ei vielä näytä kovin kummoiselta, mutta tämä ei vielä kerro paljoakaan siitä miten pitkällä sivu itse asiassa jo on. Esimerkiksi banneri ja otsikkotiedot saadaan valmiiksi lisäämällä yksi kuvalinkki, ja otsikon alapuolella oleva linkkipalkki voidaan halutessa toteuttaa joko kirjoittamalla linkit tekstimuodossa tai tekemällä linkeille kuvat. Samoin sivupalkin kuva ja tekstit voidaan syöttää lomakkeelle suoraan käsin. Oletetaan, että blogia varten on tehty jo aikaisemmin banneri, joka on nimeltään banneri.png. Lisäksi käytössä on toinen kuva, omakuva.png, jossa on kuva joka halutaan laittaa sivupalkkiin kirjoittajan profiilikuvaksi. Kuten aikaisemmassakin projektissa, myös tällä kertaa voidaan nämä kuvat siirtää projektikansioon *\public\images*, jotta ne saadaan otettua helposti käyttöön Rails-näkymässä. Samalla voidaan ottaa ylimääräinen h1-tason otsikko pois, koska sivulla on oma otsikko joka tulee kuvana. Muutetaan siis sivun runko seuraavaan muotoon:

```
01  <table borderwidth=0 cellspacing =5 width = "800" >
```

```
02  <tr>
03  <td><%= image_tag ”banneri.png", :alt=>"Kahvipöydässäkuultua"%></td>
04  </tr>
05  <tr>
06    <td><a href = "www.linkkitähän.fi">Linkki 1</a>       ||   
<a href="">Linkki 2</a> </td>
07  </tr>
08  </table>
09  <table width = "800" >
10  <tr>
11  <td>
12  <table width = "575">
13    <tr>
14    <td>Blogitekstit</td>
15    </tr>
16  </table>
17  </td>
18  <td><%= image_tag "omakuva.png", :alt=>"Laiskuri itse" %><br/>
19  Kirjoittaja on liikaa paikoillaan oleileva henkilö joka kuulee joka päivä
kaikenlaista.</td>
20  </td>
21  </tr>
21  </table>
```

Vaikka sivun määritelmä saattaa näyttää hieman uhkaavalta, ei sivun muotoiluun todellisuudessa tehty montaakaan muutosta. Keskeisin muutos on se, että alempaan tauluun lisättiin yhteen ruutuun uusi taulu. Tämä toteutettiin sen vuoksi, että taululle voitiin antaa oletusleveys (575 pikseliä), joka muuttumattomana riipppumatta siitä, millaista tekstiä sivulle on tallennettu. Järjestely ei ehkä ole teknisesti kaikista kaunein mahdollinen, mutta helppo ja yksinkertainen tapa saada aikaiseksi sivuille jonkinlainen älykäs rakenne. Samalla ohjelman etusivulle lisättiin kuvat `image_tag` –käskyillä. Nyt etusivun pitäisikin näyttää seuraavanlaiselta:



Etusivun runko, jossa perusmuotoilut ja kuvat muodostavat jo selkeästi suunnitelmaa muistuttavan rakenteen.

Nyt kun etusivu on valmis, tarvitaan vielä lisää näkymiä; esimerkiksi näkymä, jonka kautta voidaan syöttää etusivulle uusia tekstejä, vastaava näkymä komenttien jättämiselle sekä mahdollisesti muita toimintoja, joita katsotaan tarpeelliseksi sivuston valmistuessa pidemmälle. Tehdään kuitenkin ensin

pääsivun muotoilut valmiiksi ja tarkastellaan sitten enemmän sitä, mitä kaikkea projektissa joudutaan seuraavana tekemään. Aloitetaan sivun viimeistely pää- ja metatietojen lisäämisellä.

Pää- ja metatietojen lisääminen

Koska tässä projektissa on tarkoituksena toteuttaa useita näkymiä yhden ohjaimen alle, on pää- ja metotiedot helpointa luoda erilliseen muotoilu-tiedostoon, joka määritellään projektihakemistoon nimellä

```
app\views\layouts\[ohjaimennimi].html.erb
```

eli tässä tapauksessa `app\views\layouts\viestis.html.erb`. Koska tällä kertaa ei projektin luomisessa käytetty apuvälineinä scaffold-generaattoriskriptiä, ei hakemistossa vielä ole tällaista tiedostoa, ja se joudutaankin luomaan käsin.

Tämä tiedosto siis sisältää kaikki html-dokumentin tiedot, jotka normaalisti kirjoitetaan dokumentin head-osaan tai ennen sitä. Tämän lisäksi tiedostoon voidaan myös kerralla määritellä kaikille ohjaimen näkymille yhteisiä arvoja, kuten esimerkiksi käytettävä CSS-tyyliohje. Luodaan siis tiedosto `viestis.html.erb`, ja kirjoitetaan sinne seuraavat rivit:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/DTD/strict.dtd">

<html>
  <head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8"/>
  <meta name=”description” content=”kahvittelublogi” />

  <title>Kahvipöydässä kuultua: verkkoblogi</title>
  </head>

 <body>

 <%= yield %>

</body>
</html>
```

Tiedostoon sijoitetaan kaikki normaalit html-kuvauksen pääosaan tulevat tiedot, kuten käytetty html-kuvauksen versio, sivuston toteuttamiseen käytetty merkistö sekä muut metatiedot. Tärkein rivi näkymässä on kuitenkin `yield`-käsky:

```
<%= yield %>
```

Tämä rivi kertoo Rails-sivustoa käyttävälle Ruby-tulkille, että varsinainen perusnäkymä tulee tähän kohtaan sivua. Tämä johtuu rivillä olevasta, myös perus-Rubysta löytyvästä, `yield`-käskystä, joka siirtää toiminnan koodi-osiosta toiseen. Käytännössä tämä Rails-lohko siis korvataan palvelimelta sivua kutsuttaessa sivun runko-osan sisältävällä html/Rail-koodilla hakemistosta `app\views\viestis\`.

Tässä vaiheessa sivun rakenne alkaa olla valmis, mutta varsinainen tyyliohje on edelleen määrittelemättä. Seuraavaksi sivustolle voidaan vielä lisätä CSS-tyyliohje, jotta sivuston perusilme saadaan valmiiksi.

CSS-tyyliohjeen lisääminen

Tyyliohjeen lisääminen on yhtä yksinkertaista kuin kuvien lisääminen. Määritellään projektihakemistoon *\public\stylesheets* tiedosto `sivusto.css`, ja kirjoitetaan siihen alla olevat määritelmät. Nämä määritelmät kertovat selaimelle miltä runko-osan tekstin, rivinvaihtojen, taulukon tekstin, sivulle luotaville linkkien sekä otsikkotasojen `h1` ja `h2` tekstin tulisi näyttää:

```
body, p, td {
  font-family: calibri, arial, sans-serif;
  font-size:   16px;
  line-height: 20px;
  vertical-align: top;
```

```
}
h1 {
font-family: calibri, arial, sans-serif;
font-size: 22px;
line-height: 24px;
padding: 5px;
background-color: #DDDDDD;
}
h2 {
font-family: calibri, arial, sans-serif;
font-size: 18px;
line-height: 24px;
padding: 5px;
}

a { color: #000; }
a:visited { color: #222222; }
a:hover { color: #000000; background-color:#DDDDDD; }
```

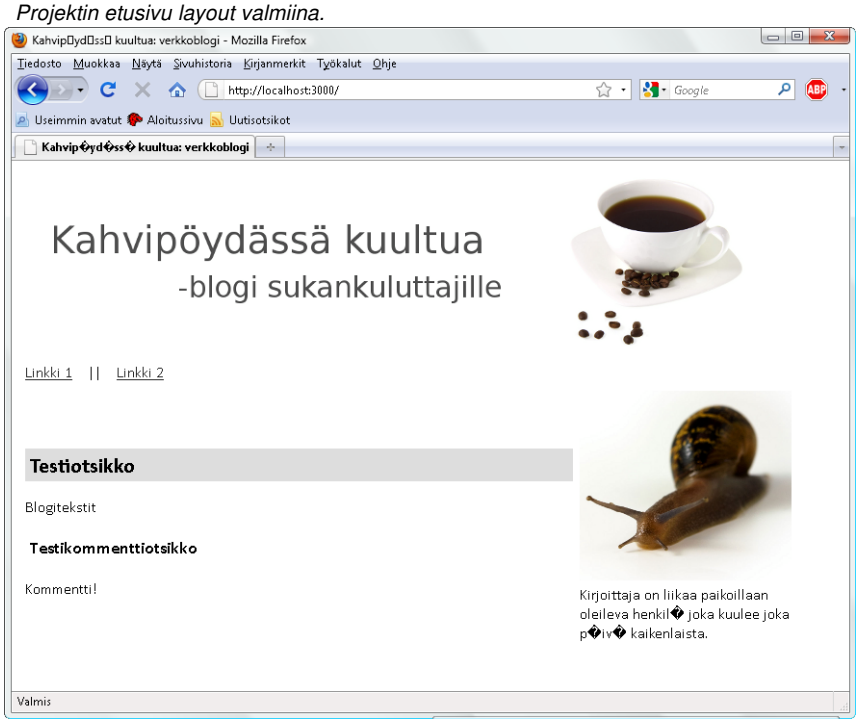
Nämä ohjeet määrittelevät siis, että sivusto käyttää fonttinaan joko calibria, arialia tai jotain sans-serif-fonttia, oletustekstin ollessa 16 pikseliä ja otsikkotasojen jonkin verran perustekstiä enemmän. Lisäksi pääotsikkotasolle `h1` on määritelty korostusväri, joka samalla toimii myös kirjoitettujen viestien erottimena, ja taulukoiden sisällöt on määrätty sijoiteltavaksi siten, että teksti alkaa aina ylimmältä riviltä, eikä keskeltä kuten oletuksena.

Viimeistellään tyyliohjeen luominen veilä lisäämällä edellisessä kappaleessa luotuun tiedostoon *app\views\layouts\viestis.html.erb* seuraava Rails-komento:

```
<%= stylesheet_link_tag 'sivusto.css' %>
```

Tämän seurauksena kyseistä tyyliohjetta käytetään aina, kun Rails-sovelluksessa käytetään koti-ohjaimeen kuuluvaa näkymää. Kun kaikki nyt muokatut tiedosto on tallennettu, voidaan käynnistää palvelin ja tarkastella palvelun verkkosivua, jonka pitäisi nyt näyttää kutakuinkin tältä:

Projektin etusivu layout valmiina.



Tässä kuvassa blogitekstit näyttävään tauluun on lisätty muutama lisärivi tekstiä html-kuvauksen eri CSS-tyylien kokeilua ja esittämistä varten.

Projektin tyyliseikat on nyt siis saatu hoidettua, ja seuraavaksi voidaankin ryhtyä toteuttamaan itse Rails-puolta projektista. Tätä varten joudutaan määrittelemään ensin se, mitä näkymiä ohjelmaan tulee, se mitä tietoa halutaan tallentaa tietokantaan ja se, kuinka eri tietotyypit sopivat toisiinsa. Tämän jälkeen

joudutaan vielä tekemään määrittelyt sille millaista tietoa projektiin ylipäänsä hyväksytään. Aloitetaan kuitenkin siitä että määritellään mitä tietoa ohjelmaan tarvitaan ja luodaan tietokanta, johon kyseisiä tietoja voidaan tallentaa.

TIETOMALLIEN MÄÄRITTELY

Edellisen projektin tietokanta- ja tietomallimääritelmä saatiin valmiiksi tehtynä Scaffold-määritelmän yhteydessä kertomalla Railsille mitä tietokantaan haluttiin. Tällä kertaa ohjelmaan halutaan kuitenkin toteuttaa kaksi erillistä tietomallia, eli viestit, joihin kirjoitetaan varsinainen päämerkintä, ja kommentit, jotka liitetään kirjoitettuihin viesteihin. Lisäksi suunnitteluehtoihin oli merkitty, että viestiin ja kommenttiin molempiin tulee lisätietoina kirjoittajan nimi, ja varsinaiseen viestiin lisäksi myös otsikko ja päiväys.

Tietokannan tietomuodot

Tietokantaa määriteltäessä joudutaan lisäksi miettimään hetki sitä, millaisessa muodossa tiedot tallennetaan. Rails-tietokantoihin voidaan määritellä tietoa useina erilaisina tietomuotoina, kuten string, joka on lyhyt teksti, interger, joka on kokonaisluku tai text, joka tarkoittaa pitkää merkkijonoa. Täysi lista erilaisista tiedontallennusmuodoista on taulukossa 12.1.

Taulukko 12.1 Tietokantojen tietotyypit

Tietotyyppi	Selite
string	Lyhyt merkkijono, kuten nimi, osoite tai otsikko.
text	Pitkä merkkijono; selite, abstrakti, johdanto...
integer	Kokonaisluku
float	Liuku- eli desimaaliluku
datetime (timestamp)	Päiväysmerkint', jossa päivämäärä ja kellonaika jolloin tietue on lisätty-
date (time)	Pelkkä päivämäärä (tai kellonaika) jolloin tietue on lisätty.
binary	Binaarista dataa; esimerkiksi kuvatiedosto tai vastaavaa.
boolean	Totuusarvo.

Lisäksi tietokannalle voidaan antaa muita lisätietoja, kuten syötteen maksimikoko tai oletusarvo lisämääreillä `limit` ja `default`. Kun erilaisia tietotyyppejä verrataan projektin tarpeisiin, voidaan tietokannan rakenteeksi valita seuraavaa:

- *viesti*-merkintä sisältää `string`-tyyppiset kentät `otsikko` ja `kirjoittaja`, `timestamp`-kentän `paivays` ja `text`-kentän `teksti`.
- *kommentti*-merkintä sisältää `string`-tyyppisen kentän `kirjoittaja`, sekä 140:een merkkiin rajoitetun `text`-kentän `viesti`.

Lisäksi merkintöjen keskinäinen suhde on tässä tapauksessa selkeä. Yhdellä *viesti*-merkinnällä voi olla useita *kommentti*-merkintöjä, mutta yksi kommentti voi kuulua ainoastaan yhdelle viestille. Näillä lähtötiedoilla voidaan ryhtyä toteuttamaan itse rakennetta.

Tietomallin luominen ja määrittely

Ensimmäisenä lisätään projektiin tarvittavat tiedostot, joihin määritellään tietokannalle haluttu rakenne. Tämä toteutetaan projektihakemiston juuressa annettavilla generaattori-komennoilla, joilla luodaan kaksi uutta mallia projektiin. komento toimii muodossa

```
ruby script\generate model [nimi]
```

Annetaan malleille samalla niitä kuvaavat nimet, eli Viesti ja Kommentti, jolloin niiden muodostamisen pitäisi tulostaa kutakuinkin tämänkaltainen teksti:

```
C:\Railsohjelmat\blogi>ruby script\generate model Viesti
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/viesti.rb
      create  test/unit/viesti_test.rb
      create  test/fixtures/viestis.yml
      create  db/migrate
      create  db/migrate/20100504165034_create_viestis.rb

C:\Railsohjelmat\blogi>ruby script\generate model Kommentti
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/kommentti.rb
      create  test/unit/kommentti_test.rb
      create  test/fixtures/kommenttis.yml
      exists  db/migrate
      create  db/migrate/20100504165100_create_kommenttis.rb

C:\Railsohjelmat\blogi>
```

Mallin lisääminen projektiin muodostaa malln määrittelyä varten kaksi keskeistä tiedostoa. Hakemistoon *app\models* luodaan mallin nimellä oleva Ruby-tiedosto (nyt esimerkiksi `kommentti.rb`), johon määritellään miten malli liittyy muihin ohjelman malleihin, sekä migrate-tiedoston kansioon *db\migrate*, johon määritellään kuinka malli toteutetaan tietokantaan.

Avataan ensimmäisenä tiedosto *app\models\viesti.rb*. Tiedoston sisältö on tässä vaiheessa seuraavanlainen:

```
class Kommentti < ActiveRecord::Base
end
```

Kommentti-mallin luokka peritään siis Railsin sisäisestä ActiveRecord-luokasta, joka toimii olioiden ja tietokannan tietueiden yhteyden muodostavana luokkana. Jokaiselle ActiveRecord-luokasta perityllä oliolla on Rails-kehyksen puolesta olemassa metodit uusien tietueiden luomista, olemassa olevien tietueiden lukeamista, päivittämistä ja poistamista varten. Tämä mm. vähentää tarvetta kirjoittaa tietokantakoodia Rails-sovelluksissa.

Jotkin oppaat puhuvat ActiveRecordin tarjoamista palveluista CRUD-mallina. Nimi tulee mallin oletuspalveluiden englanninkielisistä nimistä (Create-Read-Update-Delete).

Ensisijaisena toimintona mallitiedostoon määritellään miten mallit toimivat keskenään. Mallien välille voidaan määritellä kolmenlaisia yhteyksiä:

- Yhdestä yhteen (*one-to-one*) tarkoittaa, että malli on liitetty yksi yhteen toisen mallin kanssa. Esimerkiksi vaikkapa silmälaseilla tai autolla on ainaostaan yksi omistaja. Rails-malleille tämä yhteys määritellään avainsanoilla `belongs_to` ja `has_one`.
- Yksi moneen (*one-to-many*) tarkoittaa sitä, että malli voi liittyä useaan toisen mallin esineeseen. Esimerkiksi yhdessä yrityksessä voi olla useita työntekijöitä, tai yhdessä talossa monta asukasta. Rails-malleille tämä yhteys määritellään avainsanalla `has_many`.
- Moni moneen (*many-to-many*) tarkoittaa että mallin alkiot voivat liittyä useaan toisen mallin alkioon. Esimerkiksi kirjalla voi olla monta lukijaa, ja yksi lukija on voinut lukea monta kirjaa. Tätä yhteyttä kuvataan avainsanalla `has_and_belongs_to_many`.

Mallien yhteyksiä muodostaessa yhteys joudutaan määrittelemään kumpaankin suuntaan. Määrittely tehdään lisäämällä mallin olioon koodirivi, joka on muotoa

```
[avainsana] :[mallin nimi]
```

Seuraavaksi lisätään määritelmä Kommentti-malliin. Koska kommentti kuuluu aina vain yhdelle viestille, määritellään tämä mallin lähdekoodiin seuraavasti:

```
class Kommentti < ActiveRecord::Base
  belongs_to :viesti
end
```

Sekä vastaavasti viesti-mallin määrittelytiedostoon `viesti.rb` seuraavasti:

```
class Viesti < ActiveRecord::Base
  has_many :kommenttis
end
```

Nyt tietomallien keskinäiset suhteet on määritelty malleihin. Tämän lisäksi mallien yhteys pitää myös määritellä projektin reititystietoihin, mutta siihen tullaan aikanaan. Nämä muutokset riittävät kuitenkin mallin määrittelyyn. Seuraavaksi keskitytään luomaan tietokanta, jotta jossain vaiheessa päästään ylipäänsä testaamaan että projektisivut toimivat.

Migrate-tiedoston määritteleminen

Migrate-tiedostojen käyttötarkoitus on se, että niillä määritellään tarkalleen miten tietokanta toimii. Vaikkakin migrate-tiedoston määrittelyyn tarvitaan jonkin verran tietokantojen toimintaperiaatteiden ymmärtämistä, on työ paljon helpompi tehdä Rails-kehyksellä kuin suoraan SQL-käskyjä kirjoittamalla. Avataan seuraavaksi projektihakemistosta db\migrate\ tiedosto xxxxxxxxxxxx_create_viestis.rb (xx... määräytyy tiedoston luomishetken mukaisesti). Tiedoston sisältö pitäisi olla seuraavanlainen:

```
class CreateViestis < ActiveRecord::Migration
  def self.up
    create_table :viestis do |t|

      t.timestamps
    end
  end

  def self.down
    drop_table :viestis
  end
end
```

Tiedoston tulkitseminen vaatii hieman tietokantojen ymmärtämistä, mutta käytännössä tiedostossa määritellään mitä tietokannalla tehdään kun migrate-komento suoritetaan (`self.up`), sekä vastaavasti mitä tehdään kun määritelmä poistetaan tietokannasta. Käytännössä tapahtuma ei nyt tee muuta kuin lisää tietokantaan ”viestis”-nimisen taulun johon tallennetaan luotujen tietueiden aikaleimat, tai haluttaessa poistaa kyseisen taulun

Metodiin `self.up` määritellään siis millainen tietokannan taulu luodaan. Tietenkin ensimmäiseksi tulee itse taulun luova käsky `create_table`. Tämän jälkeen määritellään mitä tietoja tietokannan alkioihin on tarkoitus tallentaa; viestien yhteydessä tämä tarkoittaa siis merkkijonokenttiä otsikko ja kirjoittaja, aikaleimaa päiväys ja tekstikentää teksti. Nämä määritellään kirjoittamalla niille sopivat migrate-kommenot seuraavassa muodossa:

```
t.<migrate-komento> <:argumentti>, <:argumentti 2>...
```

Alun `t.` tulee metodin `self.up` alussa olevasta `create_tables viestis do |t|`-komennosta. Se voisi periaatteessa olla mitä tahansa, kuten `taulu` tai `lisaa`, mutta `t` on dokumentaatiossa vakiintunut nimi joten käytetään sitä. Migrate voikin ottaa vastaan seuraavia tietokantakomentoja:

Taulukko 12.2 Migrate-komentoja

Komento	Selite
create_table(nimi)	Luo tietokantaan taulun nimeltä `nimi`.
drop_table(nimi)	Poistaa tietokannasta taulun nimeltä `nimi`.

taulu.column(sarakkeennimi, tyyppi,lisäasetukset)	Lisää tauluun sarakkeen `sarakkeennimi`, johon tallennetaan tietoa muodossa `tyyppi`. Voi myös sisältää lisäasetuksia.
taulu.change(taulun.,vanhan.,uusinimi)	Muuttaa sarakkeen `vanhanimi` nimeksi `uusinimi` taulusta `taulunnimi`.
taulu.remove(sarakkeennimi)	Poistaa taulusta sarakkeen `sarakkeennimi.`
taulu.timestamps	Luo RESTful-mallin käyttämät oletusaikaleimat, joihin tallennetaan tieto luomishetkestä ja viimeisimmästä muokkauksesta.
taulu references (taulunnimi)	Luo tietueeseen tunnistetiedon, jolla taulun tietue voidaan kytkeä toisen taulun tietueeseen.

Koska tietokantaan kaikki tieto tallennetaan siten, että jokainen uusi tietue tulee uudelle riville, ja tiedot tallennetaan sarakkeisiin, tulee tällöin tiedoston rakenteesta kutakuinkin seuraavanlainen:

```
class CreateViestis < ActiveRecord::Migration
  def self.up
    create_table :viestis do |t|
              t.column :otsikko, :string
              t.column :kirjoittaja, :string
              t.column :teksti, :text

      t.timestamps
    end
  end

  def self.down
    drop_table :viestis
  end
end
```

Tietokantaan siis luodaan tarvittavat kentät jotta siihen voidaan tallentaa merkintöjä myöhempää käyttöä varten. Kannattaa huomata, että päiväyksen tallentava aikaleima saadaan suoraan Railsin omista `timestamps`-leimauksista. Näihin tallennetaan automaattisesti luonti- ja muokkausajankohdat `created_at` ja `updated_at`-nimillä. Määritellään samalla myös kommenttien tietomalli. Sen sisällöksi tulee kutakuinkin seuraavaa:

```
class CreateKommenttis < ActiveRecord::Migration
  def self.up
    create_table :kommenttis do |t|
              t.column :kirjoittaja, :string
              t.column :teksti, :text
      t.timestamps
    end
  end

  def self.down
    drop_table :kommenttis
  end
end
```

Kun molemmat tiedostot on tallennettu, suoritetaan vielä tietokannan rakenteen päivitys, eli migraatio, jotta määritellyt muutokset astuvat voimaan. Tämä tapahtuu rake-komennolla

```
rake db:migrate
```

Tämän pitäisi sujua ilman ongelmia ja tuottaa seuraavanlainen tulos:

```
C:\Railsohjelmat\blogi>rake db:migrate
(in C:/Railsohjelmat/blogi)
==  CreateViestis: migrating ==================================================
```

```
-- create_table(:viestis)
   -> 0.0050s
==  CreateViestis: migrated (0.0070s) ============================================

==  CreateKommenttis: migrating ==================================================
-- create_table(:kommenttis)
   -> 0.0040s
==  CreateKommenttis: migrated (0.0080s) =========================================
```

Tietokannan taulut on nyt luotu development-määrittelyssä määriteltyyn tietokantaan. Tälläkertaa tietokantaan ei määritelty mitään perusasioita monimutkaisempaa kuten oletusarvoja tai toissijaisia avaimia. Lisäksi migrate-toimintoa voi myös käyttää versionhallintaan `rake`-työkalun avulla, tästä johtuen mukana on mm. `change`- ja `remove`-komennot, mutta tätäkään toimintoa ei tässä projektissa tarvita johtuen siitä, että meillä ei ole tietokannassa mitään menetettävää, joten se voidaan joka kerta luoda kokonaan uusiksi. Tässä vaiheessa tietokantaa voidaan kuitenkin alkaa käyttämään tiedon tallentamiseen ja käsittelyyn, samalla mahdollistaen sivuston ensimmäiset varsinaiset Rails-toiminnot.

NÄKYMIEN JA KENTTIEN LUOMINEN

Kun sivuston tietokanta on saatu valmiiksi ja määritelty käyttöä varten, voidaan siihen ryhtyä tallentamaan tietoa. Tätä toimenpidettä varten tarvitaan kuitenkin näkymät ja tiedon syöttökentät, joilla tietoa voidaan syöttää tietokantaan sekä tuloskenttä, jonka avulla tallennettuja tietoja voidaan tarkastella. Koska sivun suunnitelmana oli mahdollistaa uusien viestien sekä kommenttien jättäminen, tarvitaan molemmille tapahtumille oma tiedonsyöttönäkymä. Vastaavasti pääsivu taas tarvitsee kentän, johon tallennetut viestit ja viesteihin kirjoitetut kommentit voidaan tallentaa. Seuraavaksi tehdään nämä kolme näkymää omina työvaiheinaan ja tarkastellaan yksi kerrallaan sitä, miten toimiva Rails-sivu saadaan valmistettua. Ensiksi toteutetaan tapa luoda ja tulostaa viestejä, ja tämän jälkeen lisätään sivuille mahdollisuus jättää kommentteja viestien yhteyteen.

Viestien tulostaminen pääsivulle

Ensimmäinen toiminto joka toteutetaan, on viestit pääsivulle tulostavan Rails-koodin tekeminen, jotta tietokantaan tallennettuja tietoja voitaisi tarkastella. Johtuen Rails-ympäristön arkkitehtuurista, tämä tarkoittaa kahta työvaihetta; ensin rakennetaan ohjaimeen toiminnot, jonka avulla ohjain hakee kaikki viestit tietokannasta. Tämän lisäksi tehdään näkymään tarvittavat muutokset, jolla viestit saadaan tulostettua ruudulle.

Ensin tarvitaan siis ohjaimen muutokset, joten avataan projektihakemistosta tiedosto *app\controllers\viestis_controller.rb*, joka on seuraavanlainen:

```
class ViestisController < ApplicationController
  def index
  end

end
```

Ohjaimeen ei toistaiseksi ole siis tallennettuna mitään toiminnallisuuksia. Jotta näkymään voidaan lisätä tallennetut viestit, joudutaan ne pyytämään tietokannasta. Yksinkertaisin tapa tämä toteuttamiseen on lisätä malliin muuttuja, johon tallennetaan kaikki tietokannan viestit. Tämä komento on

```
@viestit = Viesti.find(:all)
```

Komento hakee ja tallentaa muuttujaan `@viestit` kaikki `Viesti`-mallia käyttävät tietokannan tietueet. Tietueet tallennetaan taulukkoon, josta ne voidaan näkymän puolella purkaa lomakkeen kenttiin. Metodi `find` onkin yksi Rails-kehyksen käyttämän ActiveRecord-tietokantamallin tarjoamia oletuskäskyjä, ja sitä käytetään haettaessa tietokannasta tietueita. Tällä kertaa hakuehtoja ei erikseen rajattu vaan argumenttina pyydettiin haettavaksi kaikki tietueet `:all`-argumentilla. Täksi argumentiksi voidaan myös antaa esimerkiksi numeroarvo tai osoite, jolloin kannasta haetaan ainoastaan yksi tietty tietue. Kun kaikki muutokset on tehty valmiiksi, pitäisi `viestis_controller.rb`:n näyttää kutakuinkin tältä:

```
class ViestisController < ApplicationController
  def index
    @viestit = Viesti.find(:all)
  end
end
```

Seuraavaksi tehdään näkymään tarvittavat muutokset. Toisin kuin ensimmäisessä projektissa, jossa pääsivu ja Rails-toiminnot olivat erillään, on tällä kertaa tarkoituksena tuoda viestien tulostus suoraan pääsivulle. Avataan siis projektihakemiston tiedosto *app\views\koti\index.html.erb*, jota edellisen kerran muokattiin muotoilutietojen luomisen yhteydessä. Etsitään html-kuvauksesta se kohta, jossa määritellään viestit sisältävä kenttä. Tähän voidaankin kirjoittaa seuraava Rails-koodi:

```
01  <% if @viestit.blank? %>
02    <h1>Ei uusia viestejä, palaa myöhemmin uudelleen.</h1>
03  <p/>
04  <% else %>
05      <% @viestit.each do |viesti| %>
06          <h1><%= viesti.otsikko %></h1>
07          <p/><%= viesti.teksti %><p/>
```

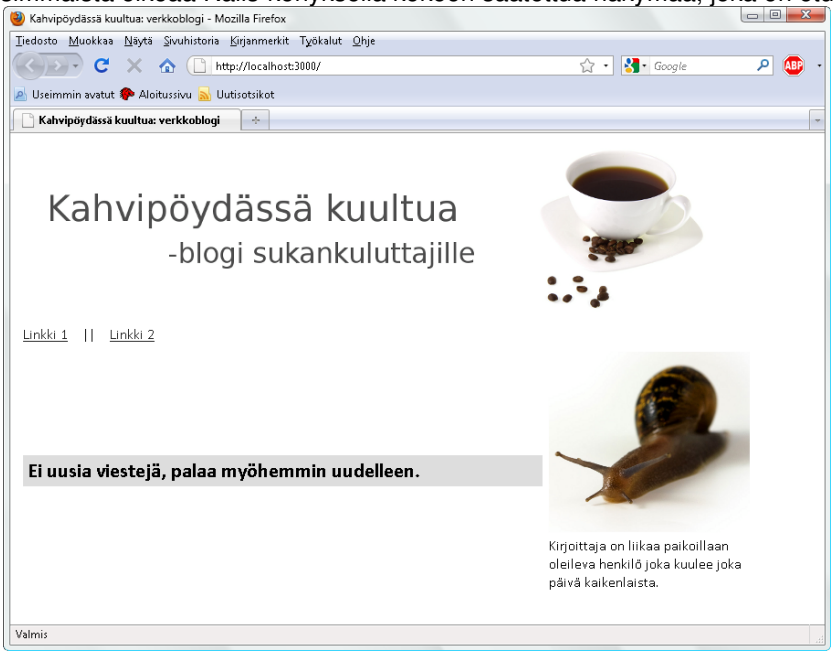
```
08          Kirjoitti: <%= viesti.kirjoittaja %><p/>
09          Tallennettu <%= viesti.created_at %><p/>
10        <% end %>
11    <% end %>
```

Kyseessä on varmasti pisin Rails-koodi mitä tähän mennessä on kirjoitettu, ja ensimmäinen oikeasti Ruby- ja html-elementtejä keskenään yhdistävä kappale, joten käsitellään se osa kerrallaan. Ensinnäkin, kuten aiemmin puhuttiin, tulee kaikki Ruby-koodi <% %>-merkkien väliin. Vastaavasti html-elementit ovat normaalissa kirjoitusmuodossaan; `<h1>`-merkkien väliin Ruby-koodilla tuleva koodi saa siis otsikon muotoilun, muiden osien tullessa normaalina tekstinä, `<p/>`-merkkien luodessa tekstiin rivinvaihtoja.

Sen sijaan, kuten tarkastellaan rivejä 1, 4, 10 sekä 11, voidaan havaita jotain mielenkiintoista; kyseessä on itse asiassa normaali Rubyn if-valintarakenne, jonka else-osiossa on each-toistorakenne. Ensimmäisellä rivillä testataan, onko ohjaimen määrittelemä `@viestit`-muuttuja tyhjä, ja määritellään sivulle tässä tapauksessa tuleva ilmoitus `"Ei uusia viestejä..."`. Vastaavasti else-osioon sisällytetty each-toisto tulostaa taulukkoon `@viestit` tallennetut viestit yksi kerrallaan siten, että otsikko tulee h1-muotoilulla, sen alapuolelle viestin tekstiosa, ja loppuun tiedot kirjoittajasta sekä päiväys. Nämä arvot määräytyvät normaalisti samaan tapaan kuin perus-Rubyn puolella. Lopuksi rakenne suljetaan kahdella end-käskyllä, joista ensimmäinen sulkee toistorakenteen ja toinen itse if-valinnan.

Tässä nähdään ensimmäistä kertaa käytössä Rails-kehyksen Rubysta perityt ominaisuudet; html-kuvauksen rakentamiseen voidaan vaikuttaa määrittelemällä Rubyn valintarakenteilla mitä sivulle tulostetaan. Vastaavasti toistorakennetta voidaan käyttää esimerkiksi tietokannan syötteiden purkamiseen taulukkoon. Kun tähän yhdistetään mahdollisuus käyttää Ruby-ohjelmointia toimintojen tekemiseen ja tietokannan ohjaamiseen, aletaan olla Rails-kehityksen ytimessä.

Joka tapauksessa tallennetaan molemmat tiedostot, käynnistetään Rails-palvelin ja tarkastellaan ensimmäistä oikeaa Rails-kehyksellä kokoon saatettua näkymää, joka on etusivulla ja näyttää tältä:



Ensimmäinen kokonainen Ruby on Rails-kehyksellä luotu verkkosivu

Vaikka tietokannassa ei vielä ole viestejä joita voitaisi tulostaa etusivulle, voidaan tässä vaiheessa havaita Rails-koodin toimivan. Blogitekstit sisältävään ikkunaan tulostuu viestien tilalle määritelty virheilmoitus eikä tyhjää ruutua.

Tässä vaiheessa on myös huomioitu Ä- ja Ö-kirjaimia käytettäessä tapahtuva ikävä ongelma, Rails ei välttämättä osaa tulostaa niitä oikein vaan antaa virhemerkin kuten ão tai ♦ tai vastaavaa. Tämä voidaan kiertää html-tekstissä html-ohitusmerkeillä ä ö Ä ja Ö jotka vastaavat isoja ja pieniä Ä ja Ö-kirjaimia. Muotoiluun on myös laitettu mukaan muutama -ohitusmerkki, jotka pakottavat selaimen laittamaan tähän kohtaan ylimääräisen välilyönnin joka muussa tapauksessa vain jätettäisi pois. Tästä on hyötyä esimerkiksi silloin, kun halutaan erottaa tekstielementtejä toisistaan. Tätä onkin käytetty esimerkiksi projektin linkki-palkissa erottamaan kaksi linkkiä toisistaan ja niiden erottimena olevista |-merkeistä.

Tietokannasta luettaessa ohitusmerkkejä tarvitsevia ongelmakohtia ei esiinny. Tiedot luetaan näkymistä, tallennetaan tietokantaan ja siirretään kuvaukseen UTF8-merkkeinä, jotka näkyvät moderneilla selaimilla tarkalleen siinä muodossa kuin ne tallennettiin.

Viestinsyöttölomakkeen tekeminen

Kun tulostustoiminto on olemassa, siirrytään vaikeampaan osuuteen, eli itse viestinsyöttöjärjestelmän rakentamiseen. Tätä toimenpidettä varten joudutaan tekemään kolme työvaihetta; ensin lisätään ohjaimeen tarvittavat käskyt, luodaan tiedonsyöttönäkymä ja lopuksi täydennetään projektin reititystietoja, jotta Rails-kehys tunnistaa ja löytää uudet palvelut.

Ohjaimeen tehtävät muutokset

Jotta viestien tallentaminen tietokantaan onnistuu, joudutaan tallentamista varten luomaan kaksi uutta toimintoa. Ensimmäinen näistä on uuden mallin mukaisen olion luova metodi `new`. Metodin nimi määräytyy Railsin käyttämän RESTful-mallin pohjalta; `new`-metodi on aina uuden, tyhjän, tietueen luovan metodin nimi ja saa tämän vuoksi kehykseltä joitain erivapauksia siihen liittyen. Joka tapauksessa, ohjaimen new-metodiin ei tarvita muuta kuin pyyntö luoda uusi `Viesti`-luokan olio, joka tallennetaan muuttujaan `@viesti`:

```
def new
   @viesti = Viesti.new
   end
```

Toinen metodi, joka ohjaimeen tarvitaan, on tapa tallentaa viesti tietokantaan. RESTful-nimeämiskäytäntöä noudattaen tälle metodille annetaan nimeksi `create`. `create` itse asiassa luo jälleen uuden `Viesti`-olion, mutta tällä kertaa käyttäen Rails-kehyksen `params`-taulukosta luettuja `viesti`-alkion arvoja, jotka otetaan talteen `new`-metodin näkymästä. Lisäksi `create`-metodi yrittää tallentaa `params`-taulukosta `@viesti`-oliomuuttujaan luetut tiedot tietokantaan mallin `save`-metodilla. Kun tallennus on tehty, siirrytään takaisin `index`-näkymään `redirect_to`-käskyllä. Kokonaisuutena `create`-metodi on seuraavanlainen:

```
  def create
            @viesti = Viesti.new(params[:viesti])
            if @viesti.save
               flash[:notice] = "Uusi viesti tallennettu!"
               redirect_to :action => 'index'
            else
               flash[:notice] = "Tapahtui virhe tallennettaessa."
               redirect_to :action => 'index'
             end
   end
```

Metodista kannattaa myös huomioida `flash`-taulukko, joka on hieman samantyylinen kuin `params`-taulukko. Molemmat ovat Rails-kehyksen omia sisäisiä, automaattisesti luotuja taulukoita, joita voidaan käyttää tiedon välittämiseen näkymien ja ohjainten välillä. `params`-taulukkoon voidaan tallentaa esimerkiksi istuntotietoja tai käyttää lomakkeiden tietojen siirtämiseen tietokantaan. `flash`-taulukkoon taas voidaan tallentaa lyhytaikaismuistiin viestejä, jotka on tarkoitus kertaluontoisesti näyttää käyttäjälle. Näitä on esimerkiksi virheilmoitukset tai toiminnan vahvistukset kuten ”Tallennettu” tai ”Viesti lähetetty”. Näiden käyttämisestä puhutaan myöhemmin lisää.

form_for-lomakkeen tekeminen

Ohjaimeen luotiin siis kaksi metodia, `new` ja `create`. Näistä `new` luo ensin tyhjän mallin, johon on tarkoitus kerätä tietueen tiedot, ja create tallentaa ne tietokantaan. Seuraavaksi tätä tapahtumaa varten on tarkoitus luoda uusi näkymä. Tämä näkymä luodaan tiedostoon *app\views\viestis\new.html.erb*, eli metodin `new` näkymäksi.
Näkymän luomisessa käytetään apuvälineenä `form_for`-komentoa, joka ottaa argumenttina mallin sisältävän olion, ja jota voidaan käyttää yksinkertaisen tiedonsyöttölomakkeen tekemiseen. Koska ohjaimen new-metodi tallentaa uuden malliolion nimellä `@viesti`, annetaan se nyt näkymässä `form_for`-komennolle argumenttina. Lisäksi otetaan näkymään mukaan vielä `index`-näkymässäkin käytetty otsikkona toimiva kuva. Näkymätiedoston tulisi näyttää seuraavanlaiselta:

```
01  <table borderwidth=0 cellspacing =5 width = "800" >
```

```
02   <tr>
03      <td><%= image_tag "banneri.png",
:alt=>"Kahvipöyd&#228;ss&#228;kuultua"%></td>
04   </tr>
05   <tr><td>
06   <h1>Kirjoita uusi viesti</h1>
07          <% form_for(@viesti) do |f| %>
08          <%= f.error_messages %>
09          <p/>
10          <%= f.label :otsikko %><br/>
11          <%= f.text_field :otsikko %>
12          <p/>
13          <%= f.label :kirjoittaja %><br/>
14          <%= f.text_field :kirjoittaja %>
15          <p/>
16          <%= f.label :teksti %><br/>
17          <%= f.text_area :teksti %>
18          <p/>
19          <%= f.submit "Tallenna viesti" %>
20          <p/>
21          <% end %>
22  </td></tr></table>
23  <%= link_to "Palaa tallentamatta", :action => "index" %>
```

form_for toimii syntaksisesti hieman samaan tapaan kuin migrate-tiedostoon tehty taulumääritelmä. Rails-koodi avataan käskyllä, jossa kerrotaan että määritellään `form_for`-käskyllä lomake `@viesti`-muuttujaan tallennetulle mallille, ja tämän jälkeen määritellään käytettävät kentät. `f.label`-komennolla luodaan seliteikkunat, ja heti niiden alapuolelle selitettä vastaava mallin osan määrittelykenttä. `f.submit`-komennolla lomakkeeseen lisätään painike, joka tallentaa tiedot `params`-taulukkoon ja kutsuu ohjaimen `create`-metodia. `form_for`-lomakkeen määrittely päättyy `end`-käskyyn. Lisäksi lomakkeen alalaitaan on vielä lisätty käyttäjälle mahdollisuus poistua lomakkeesta ilman että viestiä tallennetaan.

`form_for` on näppärä työväline tapauksissa, joissa halutaan luoda mallin tietoja käsittelevä lomate. `form_for`-käskyn avulla voidaan luoda automaattiset kentät seuraavien komentojen avulla:

Taulukko 12.3 form_for-käskyn kenttiä

Komento	Selite
f.label :[muuttuja]	Luo selitekentän jonka tekstiksi tulee muuttujan arvo
f.text_field :[muuttuja]	Luo yksirivisen tekstikentän johon kirjoitettu teksti tallennetaan muuttujaan. Tarkoitettu ensisijaisesti string-muuttujille.
f.text_area :[muuttuja]	Luo monirivisen tekstikentän, johon kirjoitettu teksti tallennetaan muuttujaan. Tarkoitettu ensisijaisesti text-muuttujille.
f.checkbox :[muuttuja]	Luo valittavissa olevan kyllä/ei-valintakentän. Tarkoitettu Boolean-arvojen antamiseen.
f.submit [teksti]	Luo tiedot create-metodille lähettävän painikkeen. Painikkeessa lukee määritelty teksti.
f.error_messages	Määrittelee syöttölomakkeeseen paikan, johon virheellisten tietojen syöttämisestä tulevat virheilmoitukset tulostetaan.

Kannattaa kuitenkin huomata, että `form_for` ei salli mallin ulkopuolisia syöteikkunoita omaan lomakkeeseensa. Tätä varten on olemassa komento `form_tag` joka toimii hieman eritavoin. Joka tapauksessa tämä näkymä huolehtii viestien tietojen keräämisestä ja tiedon välittämisestä eteenpäin tallentamisesta vastaavalle ohjaimen metodille.

Reititystietojen lisääminen

Koska projektiin on nyt lisätty uusia palveluita, on tarpeellista myös kertoa reititystiedoille siitä, mistä nämä toiminnot löytyvät ja miten ne toimivat. Koska tässä projektissa käytetään RESTful-toimintamallin mukaista ohjainta `viestis` - jonka nimi vielpä päättyy ”s”-kirjaimeen vaaditun käytännön mukaisesti –

riittää, kun reititystietoihin lisätään seuraava rivi mihin tahansa kohtaan ennen ennen aiemmin lisätty `map.root`-riviä:

```
map.resources :viestis
```

RESTful-mallin käyttäminen tarjoaa oletusarvoisesti CRUD-perustoiminnot, luoden annetulla komennolla reititystiedot oletusarvoisesti seitsämälle metodille:

- *index*-metodi sisältää aina ominaisuuden listata kaikki tallennetut tiedot.
- *new*-metodiin tehdään toiminnot jolla luodaan uusia tietueita.
- *create*-metodi tarjoaa toiminnot joilla uudet tietueet tallennetaan tietokantaan.
- *show*-metodi näyttää yhden alkion erillisessä näkymässä.
- *edit*-metodiin määritellään työkalut muokata olemassa olevaa alkiota.
- *update*-metodi tallentaa edit-metodilla tehdyt muutokset.
- *destroy*-metodin avulla tuhotaan tietueita tietokannasta.

Jotta nämä automaattisesti määritellyt toiminnot toimivat oikein, on yllä listattuja metodinnimiä käytettävä toimintojen kutsuissa. Näitä kaikkia metodeja ei kuitenkaan ole pakko määritellä. Jos jokin CRUD-toiminto katsotaan tarpeettomaksi – kuten vaikkapa viestien muokkaus – riittää kun ne vain jätetään yksinkertaisesti tekemättä.

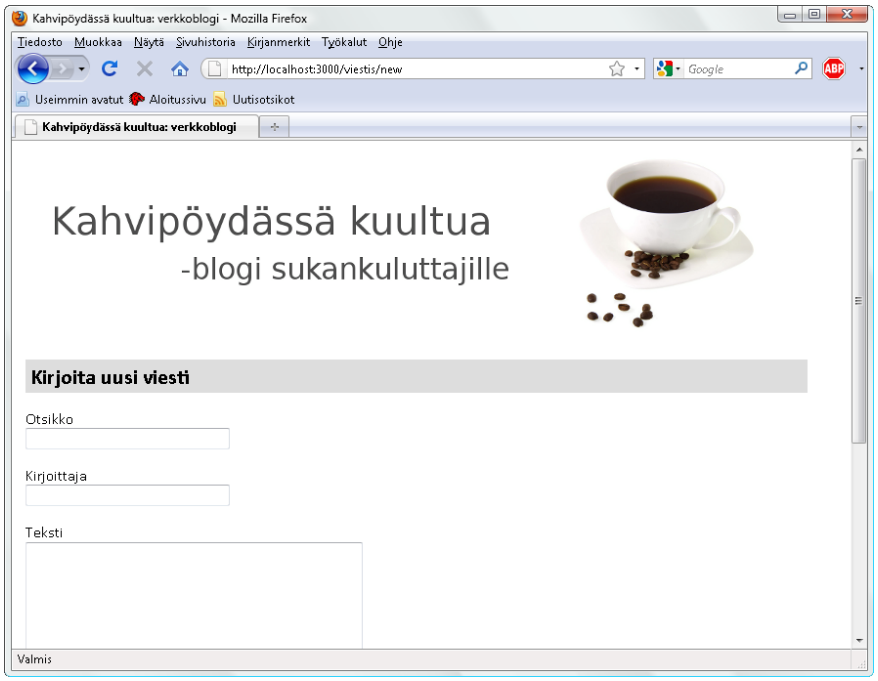
Jos RESTful-arkkitehtuurista halutaan irtautua tai ohjaimeen lisätä toimintoja jotka tulevat mallin ulkopuolelta, pitää ylimääräiset metodit määritellä reititystietoihin. Tämän tekemiseen on useita eri lähestymistapoja, syntaksin ollessa kutakuinkin muotoa

```
map.connect 'henkilo/:numero', :controller => 'henkilo', :action => 'numero'
```

Tämä esimerkiksi määrittelisi `henkilo`-nimisen ohjaimen omistavan `vuoro`-metodin, joka linkittyisi sivustoon osoitteeseen `www.sivut.com/henkilo/ <tunniste>`. RESTful-mallia käyttäville ohjainten metodeille tätä ei kuitenkaan tarvitse tehdä erikseen; riittää kun annetaan komento

```
map.resources :<ohjaimennimi>
```

Nyt kun myös reititystiedot on käsitelty, voidaan jälleen käynnistää Rails-virtuaalipalvelin ja tarkastella tehtyjä muutoksia. Ensinnäkin, etusivun sivupalkissa pitäisi nyt olla linkki ”Lisää uusi viesti”, josta aukeaa seuraavanlainen ikkuna:



Viestiensyöttöikkuna

Kun tähän lomakkeeseen kirjoitetaan tiedot ja painetaan ”Tallenna”, pitäisi viestin ilmestyä näkyviin etusivulle. Tästä tiedetään, että tietokanta toimii ja ottaa vastaan dataa.

Työvaiheen viimeistely

Nyt kun perustoiminnot on tehty, pitäisi sivustolle pystyä lähettämään uusia viestejä ja viestien pitäisi tulla näkyviin normaalisti. Sivuille voidaan kuitenkin tehdä vielä kaksi muutosta. Ensinnäkin, sivuston viestit näyttäisivät tällä hetkellä tulevan sivulle siten, että uusin viesti jää alimmaiseksi, koska uusin viesti menee aina tietokantaan seuraavalle riville ja find-metodi palauttaa tietueet tallennusjärjestyksessä. Toisekseen, ohjaimeen lisätty ominaisuus kertoa käyttäjälle viestin tallentumisesta tietokantaan ei vielä tällä hetkellä tee mitään.

Viestien kääntäminen siten, että uusin viesti tulostetaan ensin, on helppo korjata. Koska `find`-metodi hakee kaikki tietokannan viestit taulukkona, voidaan taulukko yksinkertaisesti ensin tallentaa apumuuttujaan, ja apumuuttujan avulla kääntää ympäri Rails-muuttujaa varten. Tämä tapahtuu muuttamalla tiedoston viestis_controller.rb index-metodin ensimmäiset rivit muotoon

```
muisti = Viesti.find(:all)
@viestit = muisti.reverse
```

jolloin index-näkymän käyttämä @viestit-taulu on käännetty niin päin, että viimeisenä tallennetut viestit tulostetaan ensimmäisinä. Tallennetusta viestistä saatava ilmoitus taas saadaan näkyviin lisäämällä index-näkymään (`app\views\viestis.index.html.erb`) yksi uusi Ruby-elementti ennen itse viestit tulostavaa elementtiä:

```
<% if flash[:notice] %>
<h2><%= flash[:notice] %></h2><p/>
<% end %>
```

Kyseessä on yksinkertainen valintarakenne, jossa ensin testataan onko flash-toiminnon notice-muuttujassa viestiä. Jos on, se tulostetaan näkymään h2-otsikkotasolla. Nyt käyttäjä saa huomiorivin aina tallennettuaan viestin.

Kaiken kaikkiaan, tässä vaiheessa projektin etusivun pitäisi näyttää seuraavanlaiselta:

Etusivu ensimmäisen työvaiheen jälkeen, flash-viesti näkyy ainoastaan välittömästi suoritetun tapahtuman jälkeen; viesti katoaa kun selaimen ikkuna päivitetään.

Tässä vaiheessa tiedoston *app\controllers\viestis_controller.rb* pitäisi olla seuraavanlainen:

```
01  class ViestisController < ApplicationController
02    def index
03      muisti = Viesti.find(:all)
04   @viestit = muisti.reverse
05   respond_to do |format|
06    format.html
```

```
07    format.xml {render:xml => @viestit}
08    end
09   end
10
11   def new
12    @viesti = Viesti.new
13
14   end
15
16   def create
17    @viesti = Viesti.new(params[:viesti])
18    if @viesti.save
19      flash[:notice] = "Uusi viesti tallennettu!"
20      redirect_to :action => 'index'
21    else
22      flash[:notice] = "Tapahtui virhe tallennettaessa."
23      redirect_to :action => 'index'
24    end
25   end
26  end
```

Lisäksi tiedoston *app\controller\views\index.html.erb* pitäisi olla seuraavanlainen:

```
01  <table borderwidth=0 cellspacing =5 width = "800" >
02    <tr>
03      <td><%= image_tag "banneri.png",
:alt=>"Kahvipöyd&#228;ss&#228;kuultua"%></td>
04    </tr>
05    <tr>
06    <td><a href = "www.linkkitähän.fi">Linkki 1</a>    ||    <a
href="">Linkki 2</a> </td>
07    </tr>
08  </table>
09  <table width = "800" >
10    <tr>
11    <td>
12    <table width = "575">
13      <tr>
14      <td>
15  <% if flash[:notice] %>
16  <h2><%= flash[:notice] %></h2><p/>
17  <% end %>
18  <% if @viestit.blank? %>
19    <h1>Ei uusia viestej&#228;, palaa my&#246;hemmin uudelleen.</h1><p/>
20  <% else %>
21      <% @viestit.each do |viesti| %>
22          <h1><%= viesti.otsikko %></h1>
23          <p/><%= viesti.teksti %><p/>
24          Kirjoitti: <%= viesti.kirjoittaja %><p/>
25          Tallennettu <%= viesti.created_at %><p/>
26      <% end %>
27  <% end %>
28  </td>
29  </tr>
30  </table>
31    </td>
32  <td><%= image_tag "omakuva.png", :alt=>"Laiskuri itse" %><br/>
33  Kirjoittaja on liikaa paikoillaan oleileva henkil&#246;
34  joka kuulee joka p&#228;iv&#228; kaikenlaista.<p/>
35  <%= link_to "Kirjoita viesti", :action => "new" %></td>
36  </td>
37  </tr>
38  </table>
```

Tähän mennessä toisessa projektissa on periaatteessa toteutettu ainoastaan käsin toimintoja, jotka luotiin automaattisesti ensimmäisessä projektissa Scaffolding-toiminnolla. Nyt kun viestien käsittely on saatu toimimaan halutulla tavalla, voidaan siirtyä eteenpäin, ja lisätä viesteihin kommentit.

TOISEN MALLIN LISÄÄMINEN

Sivuston ensimmäinen osa on valmis; Sivuston muotoilu on tehty, ja sivustolle voi tallentaa viestejä jotka tallennuksen jälkeen näkyvät pääsivulla. Seuraava vaihe onkin lisätä komentit olemassa olevalle sivulle. Samalla päästään tutustumaan myös lisää mallien ja reititystietojen käyttämiseen Rails-ympäristössä.

Kommenttien lisääminen

Tämä työvaihe on varmasti koko projektin työläin yksittäinen tehtävä. Sen lisäksi, että projektiin lisätään kokonaan toinen CRUD-malli, pitää se yhdistää olemassa olevaan. Käytännössä tämä tarkoittaa reititystietojen ja olemassa olevien näkymien muokkaamista, sekä muutaman uuden näkymän ja ohjaimen tekemistä.

Valmistelut ja reititystietojen päivittäminen

Kommentit määritellään toisena ohjaimena, joka kytketään viesteihin reititys- ja mallitiedoilla. Aloitetaan siis kommenttien toteuttaminen luomalla uusi ohjain nimeltään kommenttis. Jälleen kerran tulee muistaa, että johtuen nimeämiskäytännöistä, tulee RESTful-yhteensopivan ohjaimen nimen päättyä s-kirjaimeen. Luodaan ohjaimen runko `script/generate Controller`-komennolla:

```
C:\Railsohjelmat\blogi>ruby script/generate Controller kommenttis
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/kommenttis
      exists  test/functional/
      exists  test/unit/helpers/
      create  app/controllers/kommenttis_controller.rb
      create  test/functional/kommenttis_controller_test.rb
      create  app/helpers/kommenttis_helper.rb
      create  test/unit/helpers/kommenttis_helper_test.rb

C:\Railsohjelmat\blogi>
```

Tämä luo projektiin kaikki perustiedostot joita työvaiheessa tarvitaan, poislukien näkymien määrittelyyn käytettävät html.erb-tiedostot. Seuraavaksi kommenttis-malli pitää kytkeä kiinni viestis-ohjaimeen. Tarvittavat kytkennät malli-tiedostojen välillä on jo toteutettu aikaisemmin, mutta itse tietokantamääritelmään ja reitityksistä vastaavaan routes.rb-tiedostoon joudutaan tekemään kaksi muutosta.

Ensinnäkin, avataan kommenttis-tietokannan määrittelevä tiedosto `xxxxxx_create_kommenttis.rb` projektihakemistosta *db\migrate*. Tiedostossa määriteltyyn tietokantataulun kuvaukseen lisätään jokaiselle tietueelle references-tyyppinen kenttä, jolla kommenttitaulun tietueet kytketään sopivaan viesti-mallin tietueeseen. Tämä tapahtuu lisäämällä `self.up`-metodiin rivi

```
t.references :viesti
```

Muutoksen jälkeen self.up-metodin pitäisi olla seuraavanlainen:

```
01   def self.up
02      create_table :kommenttis do |t|
03            t.column :kirjoittaja, :string
04            t.column :teksti, :text
05            t.references :viesti  #Viittaus oikeaan viestiin.
06        t.timestamps
07      end
08    end
```

Koska tässä vaiheessa muutettiin tietokantojen kuvausta, poistetaan vanhentunut tietokantatiedosto `sqlite.development` hakemistosta *db* ja luodaan tietokanta uudelleen komennolla `rake db:migrate`. Kun tämä on tehty, muutetaan vielä hakemistossa *config* olevaa `routes.rb`-tiedostoa. Myös tälle tiedostolle joudutaan kertomaan, että `viestis`- ja `kommenttis`-mallit liittyvät toisiinsa. Tämä

saadaan tehtyä lisäämällä edellisessä vaiheessa tehtyyn map.resources-määritelmään lisäys `:has_many => :kommenttis`, joilloin koko rivi on seuraavanlainen:

```
map.resources :viestis, :has_many => :kommenttis
```

Nyt reititykset ja muut tausta-asetukset on saatu valmiiksi, ja seuraavaksi työn alle voidaan ottaa itse ohjain- ja näkymätiedostot. Aloitetaan itse ohjaimesta; se löytyy projektihakemistosta *app\controller* nimellä `kommenttis_controller.rb`.

Ohjainten ja näkymien tekeminen

Kommetteja luovaan ohjaimeen ei tarvita montaakaan toimintoa. Kommenttiohjaimella ei tehdä muuta kuin luodaan ja tallennetaan uusia kommenteja; jopa kommenttien näyttäminen annetaan viestis-ohjaimen hoidettavaksi. Kommenttiohjaimeen ei siis tarvita CRUD-metodeista kuin `new` ja `create`, joiden avulla kaikki toiminnot tehdään. Nämä metodit ovat periaatteessa aivan samanlaisia kuin viestejä käsiteltäessä, ainoan eron ollessa se, että Railsille pitää kommentteja käsiteltäessä kertoa se, mille viestille kommentti kuuluu. Tämä tehdään hakemalla kommentin omistavan viestin tiedot `params`-taulukosta nimellä `viesti_id`, jonka Rails-kehys määrittelee automaattisesti:

```
@viesti = Viesti.find(params[:viesti_id])
```

Koska reititystietoihin on määritelty että jokaisella viestillä on useita kommentteja, on jokaisella viesti-mallilla kommenttis-niminen jäsentaulukko. Tähän taulukkoon voidaan lisätä uusia tietueita komennolla `build`, joka lisää taulukkoon uuden alkion:

```
@kommentti = @viesti.kommenttis.build
```

Kuten viestien kanssa, ei new-metodin build-käskyyn anneta argumentteja, koska tarvittavat arvot kerätään vasta `new`-näkymän lomakkeella. `create`-metodiin `build`-komentoon annetaan argumenttina `params`-taulukosta `kommentti`-alkioon tallennetut tiedot. Kokonaisuutena `kommenttis_controller.rb`:n pitäisi siis näyttää seuraavalta:

```
01  class KommenttisController < ApplicationController
02
03    def new
04      @viesti = Viesti.find(params[:viesti_id])
05      @kommentti = @viesti.kommenttis.build
06    end
07
08    def create
09      @viesti = Viesti.find(params[:viesti_id])
10      @kommentti = @viesti.kommenttis.build(params[:kommentti])
11      if @kommentti.save
12       flash[:notice] = "Kommenttisi on tallennettu!"
13       redirect_to :controller => "viestis", :action => "index"
14      else
15       flash[:notice] = "Jotain meni pieleen!"
16       redirect_to :controller => "viestis", :action => "index"
17      end
18    end
19  end
```

`create`-metodiin tehtiin myös muutama muu lisäys. Ensinnäkin jälleen kerran `flash`-taulukkoon lisättiin käyttäjän saama ilmoitus siitä että kommentti on tallennettu. Lisäksi tallennuksen jälkeen siirrytään automaattisesti takaisin pääsivulle, eli suoritetaan viestis-ohjaimen index-metodi, jota sitäkin joudutaan päivittämään.

Äsken puhuttiin siitä, miten ohjaimelle kommentit näkyvät viestin sisällä olevana taulukkotyyppisenä jäsenmuuttujana. Tämä tarkoittaa siis sitä, että ohjaimen jokaisella viesti-muuttujalla on sisällään kommenttis-niminen taulukko, johon on tallennettuna kaikki kyseisen viestin kommentit. Tämä kommenttitaulukko voidaankin tulostaa lisäämällä jokaiseen yksittäisen viestintulostuksen sisään toistorakenne, joka tulostaa jokaiseen viestiin kaikki kommentit. Muutetaan siis tiedoston *app\views\viestis\index.html.erb* viestit tulostava koodi seuraavaan muotoon:

```
01  <% if flash[:notice] %>
02  <h2><%= flash[:notice] %></h2><p/>
03  <% end %>
04  <% if @viestit.blank? %>
05    <h1>Ei uusia viestej&#228;, palaa my&#246;hemmin uudelleen.</h1><p/>
06  <% else %>
07      <% @viestit.each do |viesti| %>
08          <h1><%= viesti.otsikko %></h1>
09          <p/><%= viesti.teksti %><p/>
10          Kirjoitti: <%= viesti.kirjoittaja %><p/>
11          Tallennettu <%= viesti.created_at %><p/>
12          <% if viesti.kommenttis.blank? %>
13             <h2> Viesti&#228; ei ole kommentoitu. </h2>
14          <% else %>
15             <h2> Viestin kommentit: </h2>
16            <% viesti.kommenttis.each do |kommentti| %>
17                 <p/>
18                <%= kommentti.kirjoittaja %> kirjoitti:
19                        <%= kommentti.teksti %> <p/>
20              <% end %>
21           <% end %>
21          <%= link_to "Kommentoi viesti&#228", new_viesti_kommentti_path(viesti) %>

22    <% end %>
23  <% end %>
```

Koodin alkuosa on edelleen sama kuin aikaisemmin: jos `flash`-taulussa on `notice`-alkioon tallennettuna viestejä, ne näytetään ruudulla. Tämän jälkeen tulostetaan viestit yksi kerrallaan ruudulle each-toistorakenteen avulla. Varsinaiset muutokset alkavatkin riviltä 12. Tässä kokeillaan onko viesti.kommentis-taulukko tyhjä, eli onko viestiä kommentoitu, ja tulostetaan ilmoitus mikäli viestille ei ole kommentteja. Jos viestille löytyy kommentteja, käydään each-toistolla läpi viesti.kommenttis-taulukko ja tulostetaan kommentit yksi kerrallaan viestin perään. Lopussa on lisäksi linkki, jonka kautta voidaan lisätä kommentteja viesteihin. Tästä linkistä kannattaa huomata argumentti viesti, joka kertoo kommenttis-ohjaimelle sen mihin viestiin kommentti liittyy. Linkissä käytettävä komento `new_viesti_kommentti_path` määritellään automaattisesti Rails-kehyksen reititystietojen ja RESTful-mallin nimeämiskäytäntöjen pohjalta, ja määrää siis tekemään linkin siihen metodiin, missä luodaan uusi kommentti viesti-malliin ja lähettää sille argumenttina viestin tiedot.

Vastaavia ”oletussijainteja” tarkoittavia käskyjä luodaan useita, mm. viestis_path joka viittaa viestis-ohjaimen index-metodiin tai edit_name_path, joka viittaa name-ohjaimen RESTful-metodiin edit. Oletuskutsujen tärkein ominaisuus on se, että ne toimivat ja määräytyvät automaattisesti aina, kun käytetty ohjain noudtattaa RESTful-arkkitehtuuria. Oletetaan, että ohjelmaan on luotu RESTful-malli nimeltä `:users`, joka on reititetty komennolla `map.resources :users`. Tälle malli luodaan oletussijainnit määräytyvät seuraavan lautukon mukaisesti:

Taulukko 12.4 CRUD-mallille määräytyvät oletussijainnit

Oletussijainnin nimi	URL-osoite	Kutsuttava CRUD-komento
users_path	/users	index
new_users_path	/users/new	new
user_path(tunniste)	/users/:tunniste	show
edit_user_path	/users/:tunniste/edit	edit

Tunniste on tavallisimmin numero, joka kertoo muokattavan tietueen sijainnin. Esimerkiksi `user_path(10)` viittaa aina user-mallin tietueeseen nro 10, mutta esimerkiksi näkymässä `user_path(user)` tarkoittaisi kutakuinkin ”näytä tämä user-malli omassa show-näkymän ikkunassa”. Arvo tietysti määräytyy näkymässä käytetyn each-toistomuuttujan mukaan.

Viimeinen lisäys on itse näkymän tekeminen. Koska kommentteja syötetään hieman samaan tapaan kuin viestejä, voidaan tähän vaiheeseen ottaa malliksi viestien syöttämiseen käytetty näkymä. Ainoa

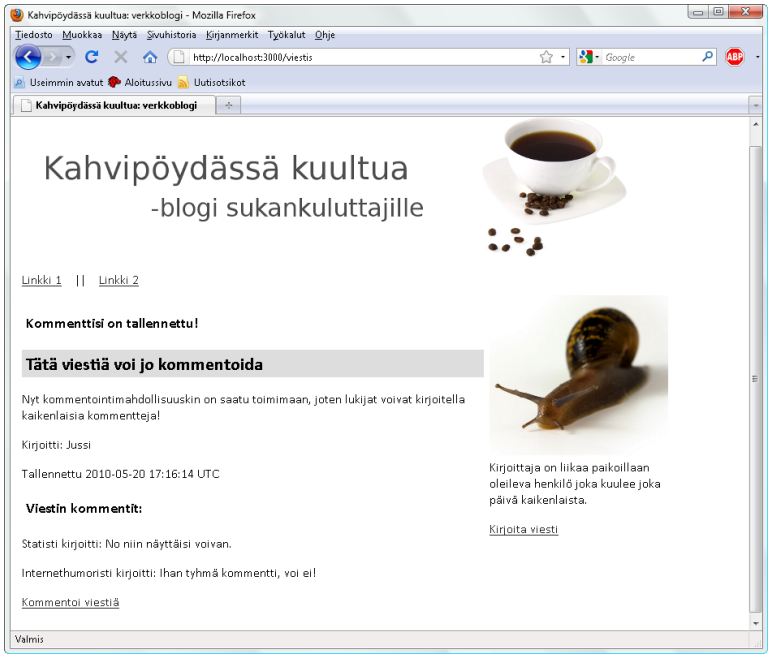
kaaviossa on se, että `form_for`-käskylle annetaan argumenttina viestin ja kommentin tiedot. Kopioidaan `viestis`-ohjaimen `new`-näkymä `kommenttis`-ohjaimen `new`-näkymäksi, eli kopioidaan *app\views\viestis\new.html.erb* osoitteeseen *app\views\kommenttis\new.html.erb* ja muokataan se seuraavaan muotoon:

```
<table borderwidth=0 cellspacing =5 width = "800" >
  <tr>
    <td><%= image_tag "banneri.png", :alt=>"Kahvipöyd&#228;ss&#228;kuultua"%></td>
  </tr>
  <tr><td>
    <h1>Kirjoita kommentti viestiin <%= @viesti.otsikko %></h1>
            <% form_for([@viesti, @kommentti]) do |f| %>
            <%= f.error_messages %>
            <p/>
            <%= f.label :teksti %><br/>
            <%= f.text_area :teksti, :size =>"50x10" %>
            <p/>
            <%= f.label :kirjoittaja %><br/>
            <%= f.text_field :kirjoittaja, :size =>"50" %>
            <p/>
            <%= f.submit "Tallenna viesti" %>
            <p/>
            <%end %>
</td></tr></table>

<%= link_to "Palaa tallentamatta", viestis_path %>
```

Näkymässä ei ole paljoakaan uutta tietoa; ainoa suoranainen muutos on `form_for`-käskyn saamat argumentit. Yksityiskohtana kannattaa huomata näkymän alareunassa oleva link_to-käsky, jossa on taas käytetty reititystietojen perusteella määrättyä `viestis_path`-oletussijaintia, sekä `text_area`- ja `text_field`-kentille lisäparametreina annetut kokotiedot luotavan syöttöikkunan koolle. Tämä automaattisesti luotu sijaintimääritelmä siis viittaa viestis-ohjaimen index-metodiin, ja onkin lyhyempi tapa kirjoittaa aikaisemmin käytetty linkinmäärittelymuoto

```
<%= link_to "Palaa tallentamatta", :controller => "viestis", :action => "index" %>
```

Viimeistelytoimenpiteenä voidaan vielä kopioida asettelu-tiedot `viestis`-ohjaimesta `kommenttis`-ohjaimen käyttöä varten. Tämä onnistuu helpoiten ottamalla tiedostosta *app\views\layouts\viestis.html.erb* kopio ja tallentamalla se samaan kansioon nimellä `kommenttis.html.erb`, tai kirjoittamalla näkymään `stylesheetlink_tag`-komento. Koska kommenttis-ohjaimeen voidaan vielä tulla tekemään lisää näkymiä, on tässä vaiheessa mielekkäämpää ottaa layouts-tiedostosta sille oma kopio, josta samalla



saadaan myös html-otsaketiedot näkymien lomake-sivuille. Nyt kommenttien käyttöönottoa varten tarvittavat muutokset pitäisi olla valmiit, ja seuraavaksi voidaankin käynnistää virtuaalipalvelin ja tarkastella uusia sivuja. Kommenttien tulostamisen ja näyttämisen pitäisi sujua ilman ongelmia, ja etusivun näyttää kutakuinkin tältä:

Viestejä voi nyt kommentoida; kommentit tulevat näkyviin viestin alle.

Viestien ja kommenttien lisätoiminnot

Kun tässä vaiheessa tarkastellaan ohjelmaa, voidaan siinä huomata vielä yksi pieni kauneusvirhe: ylipitkät kentät. Tällä tarkoitetaan tässä yhteydessä sitä, että jos jollekin viestille on kirjoitettu todella paljon kommentteja, tai vaihtoehtoisesti useita viestejä, kasvaa etusivun pituus varsin nopeasti sietämättömän pitkäksi. Tähän voidaan kuitenkin puuttua tekemällä muutama muutos sivuston rakenteeseen.

Ensinnäkin, lisätään sivuille näkymä, jossa ruudulle ladataan ainoastaan yksi viesti, ja kaikki siihen liittyvät kommentit. Tähän näkymään menemistä varten lisätään pääsivun näkymään linkki ”Avaa omaan ikkunaan” viestin loppuun. Lisäksi muutetaan toimintaa siten, että pääsivulla on näkyvillä ainoastaan kolme uusinta viestiä. Aloitetaan toimintojen rakentaminen muuttamalla veistis-ohjaimen index-näkymän koodia siten, että uudet rajoitukset astuvat voimaan:

```
01  def index
02    muisti = Viesti.find(:all)
03    pituus = muisti.length
04    if pituus > 3
05      @viestit = muisti[-3..-1].reverse
06    else
07      @viestit = muisti.reverse
08    end
09  end
```

Muutos on hyvin yksinkertainen; jos kaikki viestit sisältävä taulukko on yli kolme alkiota pitkä, suoritetaan siitä leikkaus johon otetaan talteen kolme viimeistä alkiota komennolla `muisti[-3..-1]` ja tallennetaan tämä näytettävien viestien listaksi. Muussa tapauksessa näytetään sivu kuten ennenkin, eli mitään leikkausta ei tehdä koska sitä ei tarvita.

Seuraavaksi ohjelmaan lisätään näkymä viestien tarkasteluun erillisessä ikkunassa. Koska toiminto on sisällytetty CRUD-malliin, ei sen toteuttamiseen tarvita paljoa vaivaa. Tässä tapauksessa riittää, kun `viestis_controller.rb`-tiedostoon lisätään uusi metodi nimeltä show, joka hakee tietokannasta viestin params-taulukkoon tallennetun id-parametrin avulla:

```
def show
  @viesti = Viesti.find(params[:id])
end
```

Käytännössä yhden viestin näyttäminen toimii kuten index-listaus, mutta näytettäviin viesteihin ei tallenneta kuin yksi nimenomainen viesti. Metodille luotava näkymä (tiedosto *app\views\viestis\show.html.erb*) voikin olla yksinkertaisesti riisuttu versio index-näkymän viestin tulostavasta koodista, eli vaikkapa seuraavanlainen:

```
01  <h1><%= @viesti.otsikko %></h1>
02  <p/><%= @viesti.teksti %><p/>
03  Kirjoitti: <%= @viesti.kirjoittaja %><p/>
04  Tallennettu <%= @viesti.created_at %><p/>
05  <% if @viesti.kommenttis.blank? %>
06     <h2> Viesti&#228; ei ole kommentoitu. </h2>
07  <% else %>
08     <h2> Viestin kommentit: </h2>
09     <% @viesti.kommenttis.each do |kommentti| %>
10       <p/>
11       <%= kommentti.kirjoittaja %> kirjoitti:
12       <%= kommentti.teksti %> <p/>
13       <% end %>
14  <% end %>
```

```
15  <%= link_to "Takaisin p&#228;&#228;sivulle", viestis_path %>
```

Tähän näkymään pääsemiseksi pitää vielä luoda uusi linkki index-näkymään. Lisätään siis seuraava `link_to`-komento `viestis`-ohjaimen `index`-näkymään ”Kommentoi viestiä”-linkin perään:

```
<%= link_to "Avaa omaan ikkunaan", viesti_path(viesti) %>
```

Tämä komento siis kutsuu viesti-mallin CRUD-metodia `show` ja antaa sille argumentiksi viestin tiedot. Oletussijainti `viesti_path` tulee suoraan Rails-kehyksen omista automaattisesti reititystietojen mukaan määräytyvistä tiedoista, eikä sitä tarvitse tämän tarkemmin määritellä.

Parametrin välittäminen lomakkeesta toiseen

Seuraavaksi tehdään vielä yksi suurempi muutos, jossa samalla harjoitellaan aiemmin useaan otteeseen mainitun params-taulukon käyttöä. Rails-kehys ylläpitää params-taulukossa tietoja siitä, mitä arvoja esimerkiksi lomakkeissa on tallennettu tai mihin viestiin edellisessä näkymässä viitattiin. Käytännössä params-taulukko sisältää joukon nimettyjä alkioita. Toisin kuin normaalissa taulukossa, jossa alkioihin viitatataan numeroilla, kuten vaikka `lista[1]` tai `osoite[0]`, on params-taulukossa alkioilla nimet, kuten `params[:nimi]` tai `params[:tilitiedot]`. Tässä käytössä taulukko on esimerkiksi tallennettaessa uusia viestejä tietokantaan, jolloin kaikki lomakkeen tiedot tallentuvat viesti_id-nimiseen alkioon. Lisäksi vastaavalla tavalla toimii mm. flash-taulukko; sen alkioon `notice` tallennetaan kaikki hetkellisesti sivulla esitettävät viestit. params-taulukkoa voidaan kuitenkin käyttää myös eräänlaisena muuttujamuistina, jonka avulla voidaan vaikuttaa siihen, miten verkkosivu toimii.

Helpoin tapa määritellä params-taulukkoon arvoja on antaa ne esimerkiksi link_to-käskyn yhteydessä. `link_to`-käskystä on jo kerrottu, että siinä voidaan antaa argumentit `controller` ja `action`, joilla määritellään ohjain ja toiminto johon luotava linkki viittaa. Komennon yhteydessä voidaan kuitenkin antaa mikä tahansa nimen ja arvon yhdistelmä, joka tallentuu params-taulukkoon. Esimerkiksi linkki

```
<%= link_to ”Etene”, :action =>”acceptform”, :tila = >”1”, :asento =>”eteen” %>
```

ohjaisi linkistä painettaessa näkymään acceptform, ja tallentaisi params-taulukkoon kaksi uutta arvoa; `params[:tila]` joka saa arvon `1` ja `params[:asento]`, joka saa arvon `eteen`.

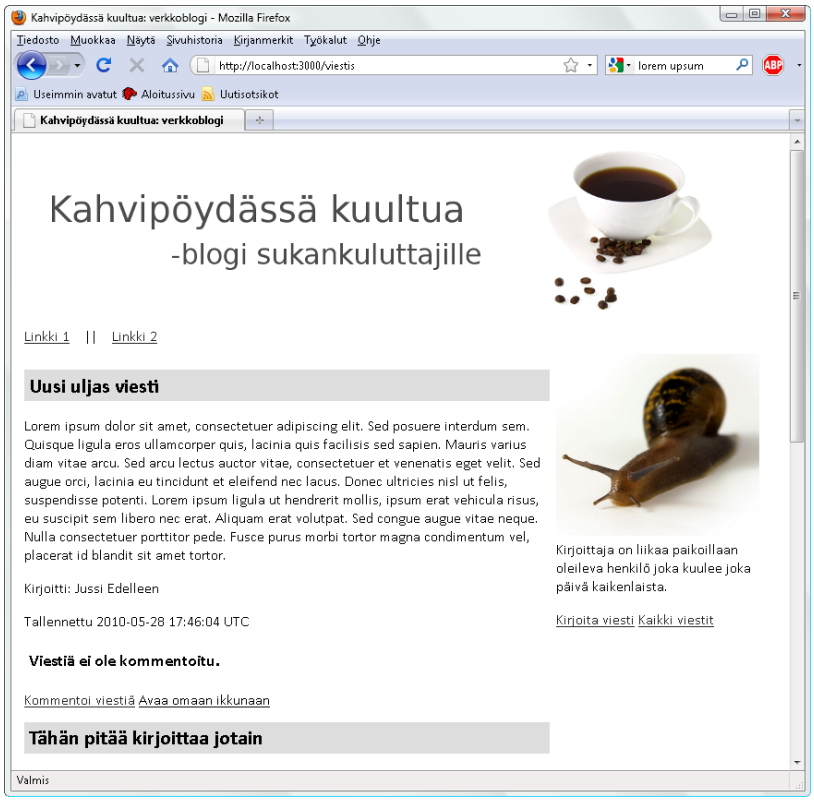
Mihin tätä ominaisuutta voidaan sitten käyttää? Tavallisin esimerkki voisi olla vaikkapa sivun tarkasteluasetusten tai istuntotietojen tallentaminen. Esimerkkiprojektissa on tähän mennessä muutettu sivunäkymiä siten, että ainoastaan kolme uusinta viestiä näytetään. Seuraavaksi lisätään näkymään vaihtoehto tarkastella kaikkia viestejä lisäämällä näkymään yksi uusi parametri show, jonka arvoksi laitetaan all silloin kun linkkiä painetaan. Lisätään siis seuraava `link_to`-käsky `viestis`-ohjaimen näkymään `index`, sivupalkkiin ”Kirjoita viesti”-linkin perään:

```
<%= link_to "Kaikki viestit", :action => "index", :show => "all" %>
```

Käytännössä linkki kutsuu index-metodia, eli lataa etusivun uudelleen, mutta sillä erotuksella, että alkio `params[:show]` on saanut arvon `all`. Tätä parametrimuutosta voidaan nyt käyttää hyväksi näkymän ohjaimessa; muokataan koodia siten, että jos `:show` saa arvon `all`, näytetään kaikki viestit. Tämä on voidaan toteuttaa helposti tekemällä ohjaimeen `app\controllers\ viestis_controller.rb` pieni Ruby-ohjelmakoodin muutos. Muutetaan viestilistan leikkauksesta vastaavaa valintaehtoa siten, että parametrin arvo tarkastetaan:

```
01  def index
02    muisti = Viesti.find(:all)
03    pituus = muisti.length
04    if pituus > 3 and not params[:show] == "all"
05      @viestit = muisti[-3..-1].reverse
06    else
07      @viestit = muisti.reverse
08    end
09  end
```

Nyt pääsivulla olevan ”Kaikki viestit”–linkin painaminen aiheuttaa sen, että viestilistan pituuden mukaan tapahtuvaa leikkausta ei toteuteta. Kaikenkaikkiaan, tässä vaiheessa ohjelmaan on saatu valmiiksi kaikki siinä käytettävät näkymät, ja sivuston yleisilme alkaa olemaan pitkälti suunnitellun kaltainen:



Sivut valmiina. Uudet toiminnallisuudet toimivat linkkien kautta.

PROJEKTIN VIIMEISTELY

Projekti alkaa olla valmis, ja käytännössä ainoastaan kaksi vaadittua toimintoa on enää toteuttamatta; salasanavarmennus ja syötetyn tiedon oikeellisuuden tarkastaminen. Niistä ensimmäisenä toteutetaan syötetyn tiedon tarkatus, jonka avulla päästään eroon virheellisesti muotoilluista viesteistä ja kommenteista. Tämä jälkeen lisätään vielä yksinkertainen salasanavarmennus pitämään huolen siitä, ettei kuka tahansa pysty kirjoittelemaan blogiin viestejä, vaan että kirjoittajan pitää tietää salasana jotta hänen kirjoituksensa tallennetaan.

Syötetyn tiedon tarkastaminen

Vaikka kaikki toiminnot ovatkin teknisesti valmiina, on sivuston keskeinen ongelma edelleen se, että tallennettavia tietoja ei tarkasteta millään tavalla. Esimerkiksi kommenttien jättäminen onnistuu edelleen ilman nimeä, tai vaihtoehtoisesti keskeneräisten viestien jättäminen onnistuu vahingossa painamalla ”Tallenna”-painiketta viestinsyöttökentässä. Tähän epäkohtaan puututaankin ensimmäisenä viimeistelytoimena, ja se onkin varsin helppo toteuttaa.

Rails-kehykseen on lisätty useita sisäänrakennettuja tarkastuksia ja testejä, joiden avulla voidaan tarkastaa että syötetty tieto on toivotunkaltaista. Tämä tapahtuu määrittelemällä mallin määritelmät sisältävään tiedostoon (projektihakemiston *app\models* tiedostot} lisäehtoja mallin luomiselle. Lisäehtoja lisätään malliin antamalla luokkamääritelmään komentoja muodossa

```
[ehto] [argumentit]
```

eli vaikkapa

```
validates_presence_of :omistaja
```

Joka tarkastaa että mallin omistaja-kentälle on määritelty jokin muu arvo kuin nul, tyhjä merkkijono tai pelkkiä whitespace (eli välilyönti, sisennys tai rivinvaihto) merkkejä. Ohessa onkin lista yleishyödyllisistä Rails-kehyksen testeistä:

Taulukko 12.5 Mallien tarkastamiseen käytettäviä perustestejä		
Testin nimi	Hyödyllisiä lisäargumentteja	Selite
validates_presence_of <kenttä>		Tarkastaa, että kenttä saa jonkin muun arvon kuin nul, tyhjän merkkijonon tai whitespace-merkkejä
validates_length_of <kenttä>	minimum, maximum, within, is	Tarkastaa että kentän saama teksti on oikeanmittainen.
validates_numericality_of <kenttä>		Tarkastaa, että kentän arvo on numeroarvo.
validates_uniqueness_of <kenttä>		Varmistaa, että kentän arvo on ainutlaatuinen, eli ettei samaa arvoa ole jo annettu toiseen tietueeseen. Vrt. sosiaaliturvatunnus tai opiskelijanumero.
validates_inclusion_of <kenttä>	:in => (taulukko tai lukulista)	Testaa, että kentän arvo on lukulistan sisältä tai vastaa taulukon alkiota.

Kun malleihin on lisätty tarkastuksia, pitää syötettävän tiedon mennä läpi kaikista tarkastuksista, tai tieto yksinkertaisesti hylätään. Tämä tarkoittaa siis sitä, että yritykset syöttää tarkastuksienvastaista tietoa epäonnistuu, eikä esimerkiksi create-komennon save-käsky tallenna tietuetta tietokantaan. Lisätään seuraavaksi projektin mallien määritelmiin muutama tarkastus, jolla varmistetaan että kaikki toivotut kentät täyttyvät. Muutetaan siis tiedostoa `app\models\kommentti.rb` seuraavaan muotoon:

```
01  class Kommentti < ActiveRecord::Base
02  belongs_to :viesti
03  validates_presence_of :kirjoittaja, :teksti
04  validates_length_of :teksti, :maximum => 120,
05  end
```

sekä tiedosto `app\models\viesti.rb` seuraavaan muotoon:

```
01  class Viesti < ActiveRecord::Base
02  has_many :kommenttis
03  validates_presence_of :otsikko, :kirjoittaja, :teksti
04  validates_length_of :otsikko, :minimum => 4
05  validates_length_of :kirjoittaja, :minimum => 3
06  validates_length_of :teksti, :minimum => 50
07  end
```

Uusien kommenttien syötön yhteydessä siis tarkastetaan, että molemmat kentät, kirjoittaja ja teksti, saa jonkin arvon, ja että itse kommentti on korkeintaan 120 merkkiä pitkä. Samoin viestin tallentamisessa otetaan käyttöön samantyyliset säännöt; jokaisessa kentässä on oltava jokin arvo sen lisäksi, että niille määritellään myös minimipituudet. Malliin tehdyt muutokset vaikuttavat yrityksiin lisätä uusia viestejä: virheellisen viestin jättäminen johtaa ainoastaan virheilmoitukseen koska väärin muotoillun viestin tallentaminen tietokantaan ei enää onnistu:

Sivulle yritettiin lisätä viesti jossa ei ollut otsikkoa; tarkastus esti viestin tallentamisen tietokantaan.

Yksinkertaisen salasanakyselyn lisääminen

Viimeinen projektille annettu vaatimus oli se, että viestien lisääminen onnistuu ainoastaan henkilöltä, jolla on lupa lisätä viestejä verkkosivulle. Tätä varten ohjelmaan lisätään vielä yksinkertainen salasanavarmennus uusien viestien lisäämistä varten. Koska sivuilla ei kuitenkaan ole minkäänlaista käyttäjänvalvontaa, kuten esimerkiksi sisäänkirjautumista tai oikeastaan edes listaa käyttäjistä, toteutetaan järjestelmä yksinkertaisimmilla mahdollisella tavalla. Tämä tarkoittaa sitä, että lisätään uuden viestin vastaanottavaan näkymään (*app\views\viestis\new.html.erb*) lomaketietoihin ylimääräinen kenttä salasana, joka tallennetaan parametrina ja välitetään viestin tietokantaan tallentavalle ohjaimelle.

Näkymässä käytettiin for_form-muotoa lomakkeen tekemiseen. Tämän käskyn ongelma on kuitenkin siinä, että siihen ei voida liittää kenttiä, jotka tallentavat argumenttina annetun tietomallin ulkopuolista tietoa. Koska viesti-mallissa ei ole ”salasana”-nimistä kenttää, aiheuttaa tämä virheen jos kenttä koitetaan tallentaa form_for-lomakkeella. Siksi joudutaankin käyttämään form_tag-muotoa, jolla on mahdollista luoda malleista riippumattomia kenttiä. Yksinkertaisimmillaan `form_tag` toimii siten, että ensin määrätään luotavaksi yleinen kenttä komennolla

```
<% form_tag do %>
```

Tämän jälkeen määritellään luotavat kentät käskyillä

```
<%= <kentäntyyppi> <tallennusnimi> %>
```

eli vaikkapa

```
<%= text_field_tag "nimi" %>
```

joka siis luo lyhyen tekstikentän, jonka saama arvo tallennetaan params-taulukon tietueeseen `nimi`. Kentän tyyppinä voi olla esimerkiksi `text_area_tag` joka luo suuremman tekstinsyöttöikkunan, `label_tag` joka luo otsikkokentän tai `password_field_tag`, joka luo tekstinsyöttökentän jossa merkit on piilotettu. Lomakkeen loppuun tulee submit_tag muodossa `<%= submit_tag "Painikkeen teksti" %>` ja lomakkeen sulkeva `<% end %>`-merkki. Submit-käskyn jälkeen lomakkeen kentät tallennetaan parametreinä params-taulukkoon paikoille `params[:<tallennusnimi>]`. Lisätään siis seuraava form_tag-kaavake uuden viestin tallentavaan lomakkeeseen:

```
<% form_tag do %>
   <p/>Salasana: <%= password_field_tag "salasana" %>
```

```
<% end %>
```

Laitetaan lisäkenttä `form_for`-lomakkeen sisään siten, että `salasana`-parametri tallennetaan samalla `submit`-komennolla kuin muut uudet tiedot. Käytännössä tämä toteutetaan siten, että tiedonsyöttölomake muokataan seuraavanlaiseksi:

```
01  <% form_for(@viesti) do |f| %>
02    <%= f.error_messages %>
03    <p/>
04    <%= f.label :otsikko %><br/>
05    <%= f.text_field :otsikko %>
06    <p/>
07    <%= f.label :kirjoittaja %><br/>
08    <%= f.text_field :kirjoittaja %>
09    <p/>
10    <%= f.label :teksti %><br/>
11    <%= f.text_area :teksti %>
12    <% form_tag do %>
13      <p/>Salasana: <%= password_field_tag "salasana" %>
14    <% end %>
15    <p/>
16    <%= f.submit "Tallenna viesti" %>
17    <p/>
18  <%end %>
```

Koska lomakkeessa on jo yksi `submit`-kenttä, ei `form_tag` tarvitse omaa submit-osaa vaan salasanaksi annettu syöte tallennetaan samalla kertaa muiden form_for-kaaviossa määriteltyjen kenttien kanssa. Seuraavaksi tehdään salasanan tarkastus itse viestin tallennuksesta vastaavaan create-metodiin, joka sijaitsee tiedostossa *app\controllers\viestis_controller.rb*.

Määritellään salasanaksi vaikkapa kestosuosikki ”`salakala`”. Ohjaimen tallennusta yrittävään valintarakenteeseen voidaan lisätä Ruby-koodina lisäehto, että tallennusta yritettäessä `params[:salasana]` on saatava arvo ”`salakala`”. Käytännössä tämä tarkoittaa sitä, että metodi tulee muokata seuraavaan muotoon:

```
01  def create
02    @viesti = Viesti.new(params[:viesti])
03    if params[:salasana] == "salakala" and @viesti.save
04      flash[:notice] = "Uusi viesti tallennettu!"
05      redirect_to :action => 'index'
06    else
07      flash[:notice] = "Tapahtui virhe tallennettaessa."
08      redirect_to :action => 'index'
09    end
10  end
```

Nyt ohjelma aiheuttaa virheen, mikäli käyttäjä ei anna viestilomakkeen salasanaa oikein. Tässä tapauksessa flash-taulukkoon tallennetaan virheilmotius ja viesti jätetään tallentamatta. Järjestely ei ehkä ole kovin tietoturvallinen – salasana ei ole käyttäjäkohtainen ja se on luettavissa selkokielisenä lähdekoodista – mutta on tämän projektin tarpeisiin varsin riittävä. Se estää satunnaista surffailijaa lisäämästä viestejä sivuille, eikä ole triviaalisti selvitettävissä esimerkiksi käyttäjän päästä sivujen lähdekoodia tarkastelemalla.

YHTEENVETO PROJEKTISTA

Tässä vaiheessa ohjelman kaikki päätoiminnot on toteutettu. Sivuilla on projektisuunnitelmassa määritellyn kaltainen rakenne, jossa sivuilla on otsikkokenttä ja sen alapuolella tilaa linkeille. Lisäksi sivujen toiminnot täyttävät annetut vaatimukset; viestejä voi kommentoida, uusien viestien lisäämistä varten tulee tietää salasana ja sivulle kirjoitettavan tekstin tulee täyttää annetut minimivaatimukset. Periaatteessa tämä voisi olla se vaihe projektissa, kun pystytään siirtymään pelkästä kehitystyöstä testaukseen, myöhemmin mahdollisesti jopa siirtyä kokeilemaan sivuston toimintaa julkisessa verkossa.

Mitä tästä projektista sitten jää käteen? Periaatteessa projektissa opeteltiin kaksi uutta asiaa: yhdistämään kaksi RESTful-mallia toisiinsa ja käyttämään parametreja sivuston tietojen välittämisessä. Lisäksi käytiin vaihe vaiheelta läpi kaikki ne työvaiheet, jotka aikaisemmin jätettiin scaffold-toiminnon toteutettavaksi, samalla tutustuen Rails-kehyksen malli/ohjain/näkymä-järjestelmän toimintaan käytännössä. Jos projektia jatkettaisi tästä vaiheeta eteenpäin, olisi seuraava työvaihe varmaankin toteuttaa mahdollisuus poistaa asiattomia kommentteja tai muokata viestejä, kumpikin toimintoja jotka ”oikeille” Internet-sivuille varmasti tarvitaan. Kummankin toiminnon toteuttaminen onnistuu CRUD-toimintojen avulla, joten ainoa varsinaisesti suunnittelutyötä vaaativa työvaihe olisikin luoda salasanavarmennus toimintoihin. Toisaalta taas, tämä lisäisi peruskäyttäjälle tarpeettomien linkkien määrää sivulla, joten järkevämpää olisi luultavasti tehdä erilliset ylläpitosivut, joiden kautta valvojat voisivat hallita viestejä ja kommentteja.

Tähän mennessä on kuitenkin läpikäyty Rails-ympäristön perusasiat, ja jatkotoimenpiteet ovat pitkälti omista mielenkiinnonkohteista kiinni. Seuraavassa luvussa keskustellaan vielä joistain Rails-kehyksen toiminnoista, joita harjoitusprojekteissa ei käytetty, sekä kerrotaan ohjeita siihen, miten tästä vaiheesta kannattaa jatkaa eteenpäin.

LUKU 13

Muita Rails-toimintoja

Mitä seuraavaksi?
Huomioita Rails-kehyksestä

Tässä luvussa tarkastellaan sitä, mitä kaikkea Rails-projekteihin voidaan vielä aikaisemmissa projekteissa opetellun lisäksi tehdä. Tämä tarkoittaa siis virhesivujen tekemistä, sivustojen testaamista, sekä Rails-ohjelman laittamista verkkoon. Luvun lopussa tarkastellaan sitä, mihin suuntaan Rails-kehys on lähiaikoina kehittymässä sekä siitä, mistä on helpointa löytää kehystä koskevaa lisätietoa.

MITÄ SEURAAVAKSI?

Tässä vaiheessa ensimmäinen oikea Rails-sovellus alkaa olla siinä vaiheessa, että sivujen testaaminen voitaisiin aloittaa. Tämä tarkoittaa siis sitä, että Rails-projektissa siirryttäisi developer-tilasta test-tilaan, jossa sovellukselle voitaisiin määritellä erilaisia testiskenaarioita ja tapahtumia, joiden avulla kokeillaan että kaikki toimii niin kuin on tarkoitettu. Kun testaus on saatu valmiiksi, olisi vuorossa Rails-sovelluksen siirto varsinaiselle palvelimelle, mistä sivusto näkyisi julkiseen verkkoon.

Kehitystilan vaihtaminen

Rails-kehyksessä määritellään kolme erilaista käyttötilaa; development jossa Rails-ohjelma on tarkoitus rakentaa ja saattaa toimimaan, test jossa sovellus testataan ja production, joka on tarkoitettu siihen vaiheeseen, kun sovellusta käytetään oikeasti. Näitä tiloja on mahdollista vaihtaa antamalla projektikansiossa komento

```
set RAILS_ENV=<tila>
```

johon asetukseksi <tila> voidaan siis antaa joko `development`, `test` tai `production`. Asetuksen näkyvin ominaisuus on siinä, että se käyttöönottaa eri tietokanta-asetukset, mahdollistaen esimerkiksi esitäytettyyn testaus-tietokantaan siirtymisen toimintojen testaamista varten. Joissain tapauksissa kannattaa myös huomata, että palvelut olettavat tiettyjen asioiden tapahtuvan tietyssä kehitystilassa; Esimerkiksi Rails-sovelluksen laittaminen verkkoon Passenger-työkalun avulla toimivalle palvelimelle olettaa automaattisesti, että siirrytään käyttämään production-tilaa.

Virhesivujen määritteleminen

Testauksella päästään eroon isoimmasta määrästä virheitä, mutta kaikista niistä ei koskaan päästä eroon. Ohjelmiin voidaan sanoa aina jäävän ainakin muutama huomaamaton virhe tai ongelma, itse asiassa täysin virheettömän koodin tekemisen voidaan sanoa olevan käytännössä mahdotonta heti kun koodirivien määrä kasvaa riittävän suureksi. Tämän vuoksi virhetilanteessa esitettävien sivujen tekeminen ei ole ainoastaan ylivarovaisuutta vaan varmasti hyödyllistä.

Tämän vuoksi projektikansiossa public on kaksi HTML-tiedostoa, *404.html* ja *500.html*, jotka Rails esittää siinä tapauksessa, että sivun tekemisessä tapahtuu virhe. Näitä sivuja on mahdollisuus muokata sisältämään esimerkiksi ilmoitus ”Jotain meni rikki, palaa myöhemmin uudelleen” tai antamalla sähköpostiosoite, johon viasta voi kertoa. Kannattaa kuitenkin huomata, että sivut eivät salli Ruby-koodin käyttämistä muotoilussaan, vaan hyväksyvät ainoastaan HTML-kuvausta.. Lisäksi nämä sivut eivät toimi paikallisesti WEBrick-virtuaalipalvelinta käytettäessä; kun selailu tapahtuu *http://localhost*-sijainnissa, ei Rails käytä näitä virhesivuja vaan esittää virheilmoitussivun, jossa on tietoa tulkin tilasta ja käytössä olleista asetuksista.

Rails-ohjelman testaaminen

Yksi Rails-kehyksen epätavallisempia ominaisuuksia on sen sisäänrakennettu tuki erilaisille testausmuodoille. Jos esimerkiksi tarkastellaan mitä kansioita uuden projektin luominen generoi, sisältää ne oletuksena test-nimisen kansion projektihakemistossa. Tähän hakemistoon voidaan määritellä erilaisia testejä ohjainten toimintaa, järjestelmän toimintaa ja osien yhdistämistä varten. Näitä testitapauksia voidaan määritellä YAML- tai CSV-muodossa, ja niiden käyttäminen onkin näppärää silloin, kun projektin koko alkaa kasvaa, ja jokaisen muutoksen jälkeen testattavia toimintoja sekä yhteensopivuuksien tarkastuksia on useita. Ohjeita näiden testitapausten tekemiseen löytyy verkosta useita, ja projektihakemistossa olevan tiedoston *test\test_helper.rb* ohjeiden avulla pääsee alkuun.

Rails-ohjelman siirtäminen verkkoon

Tässä kirjassa on siis tehty Rails-kehyksellä verkkoblogi, johon olisi mahdollista kirjoittaa viestejä ja antaa lukijoiden jättää kommentteja kirjoitettuihin viesteihin Lisäksi ohjelma on testattu ja todettu että kaikki näyttäisi toimivan ilman ongelmia. Enää olisikin edessä se vaihe, jossa ohjelma on tarkoitus siirtää verkkoon, pois omalta työkoneelta. Mitä tähän sitten varsinaisesti tarvittaisi?

Helpoin yhdistelmä varmaankin on käyttää Mac- tai Linux-palvelinkonetta, johon on asennettu Apache-verkkopalvelin ja toivotut tietokantatyökalut, kuten MySQL. Tälle yhdistelmälle on mahdollista asentaa RubyGems-järjestelmän kautta Phusion Passenger (*http://www.modrails.com*) – niminen Rails-palvelinrajapinta. Tämän jälkeen Rails-sivun laittaminen verkkoon vaatii enää yksinkertaisen, 4

riviä pitkän asetustiedoston määrittely, jolla kerrotaan Apachelle missä Rails-ohjelman public-kansio sijaitsee, esimerkiksi:

```
<VirtualHost *:80>
  ServerName www.kahviblogi.net
  DocumentRoot /railsjutut/kahviblogi/public/
</VirtualHost>
```

Tämä määrittely kytkisi verkkosivun *www.kahviblogi.net* (joka tietenkin pitää ensin rekisteröidä palvelimen käyttöön) hakemistossa */railsjutut /kahviblogi/public/* olevaan Rails-sovellukseen.

Windows-puolella tilanne ei ole näin yksinkertainen. Ensinnäkin, tarvitaan palvelinkäyttöön sopiva Windows-käyttöjärjestelmän versio, jotka eivät ole kovinkaan yleisiä yksityiskäytössä. Tällöinkään Windows-puolelle ei löydy yksinkertaista tapaa saada Rails-palvelin toimimaan. Yksi keino on Apache- ja Rails-tarvikkeiden lisäksi asentaa RubyGems-paketti `redcloth` sekä `Rmagick`-niminen ohjelma sekä asettaa proxy-palvelu kytkemään Apache ja Rails toisiinsa. Tämä järjestely kuitenkin hidastaa palvelun toimintaa jonkin verran, joten jos palvelimelle odotetaan useaa yhtäaikaista käyttäjää, ei järjestely ole järkevin mahdollinen.

Yksinkertaisinta onkin käyttää Linux-palvelinkonetta, ja siinä avoimen lähdekoodin työvälineitä jos palvelun on tarkoitus tulla oikeaan, vakavamieliseen käyttöön. Tämä kombinaatio onkin eräänlainen standardi Rails-kehittäjien keskuudessa, ja sille löytyy hyvin työkaluja sekä apua mikäli jokin osa ei suostu toimimaan yhteistyössä muiden kanssa. Yksi vaihtoehto on tietysti luoda Windows-palvelimelle Linux-virtuaalikone jossa palvelinta käytetään, mutta tämä vaatii jo jämerämpää rautaa, eikä muuta sitä perusajatusta että *nix-ympäristössä Rails-kehys on helpointa saada verkkoon.

HUOMIOITA RAILS-KEHYKSEN KEHITYKSESTÄ

Kuten aiemmin sanottu, Ruby on Rails on kokonaisuuta suhteellisen uusi ja kehittyy edelleen jatkuvalla vauhdilla. Tämän kirjan kirjoitushetkellä Rails elääkin hyvin mielenkiintoisia aikoja; uusi pääjulkaisu 3.0.0 on Beta-vaiheessa, joka tulee jonkin verran muuttamaan järjestelmän toimintaa, mutta ei paljoakaan vaikuta tässä kirjassa käsiteltyihin perusasioihin. Kaksi eniten vaikuttavaa asiaa ovat pienet muutokset komentorivikäskyjen syntaksiin, esimerkiksi

```
rails g scaffold post otsikko:string
```

aiemmin käytetyn

```
ruby script/generate scaffold post otsikko:string
```

sijaan. Ylipäänsä muutokset tarkoittaa sitä, että `script/*`-komennot korvautuvat `rails`-alkuisilla käskyillä. Lisäksi reititystietoihin tulee jotain muutoksia johtuen siitä, että toimintojen sisäiseen hierarkiaan ja nimiavaruuksiin tehdään muutoksia.

Kaikenkaikkiaan tämä tarkoittaa sitä, että Rails-kehyksen käyttäminen ja jatko-opiskelu tämän kirjan ohjeista eteenpäin vaatii verkossa surffailua, koska uusin tieto löytyy aina Rails-yhteisön kotisivujen kautta, ja eri versioiden välillä tapahtuu jonkin verran muutoksia. Rails-kommuunin kotisivuilta löytyvä Documentation-alisivu (*http://rubyonrails.org/documentation*) sisältääkin paljon yhteisön kirjoittamia ohjeita jatkotoimenpiteitä varten. Nämä ohjeet eivät kuitenkaan monesti ole kovin aloittelijaystävällisiä, vaan niissä keskitytään esittelemään asia helppotajuisuuden ja käytettävyyden kustannuksella. Lisäksi verkosta löytyy myös muita lähteitä, mutta usein niitä käytettäessä tulee törmänneeksi versio-ongelmiin. Joskus esimerkiksi Googlen löytämä materiaali saattaa olla muutaman vuoden vanhaa, mikä Rails-kehyksessä voi tarkoittaa suurtakin eroa lähteiden välillä. Tämän vuoksi onkin hyvä tarkastaa, että annetut ohjeet koskevat nimenomaisesti sitä Rails-kehyksen versiota, jonka kanssa työskennellään.

LOPPUSANAT

Tässä vaiheessa on käyty läpi kaikki Ruby-ohjelmoinnin perusasiat kahdekassa perusteita käsittelevässä luvussa, sekä tutustuttu Rails-ohjelmakehykseen kolmen esimerkkiprojektin avulla. Näissä esimerkkiprojekteissa ensimmäisessä kokeiltini Rails-projektin tekemistä ja luotiin Rails-kehyksellä näkymiä ja niihin liittyvä ohjain. Toisessa projektissa tehtiin Rails-kehyksen automaattisten toimintojen avulla muistikirja, jossa oli mahdollista lisätä, muuttaa ja poistaa merkintöjä RESTful-perusteisen CRUD-mallin tarjoamien toimintojen avulla. Kolmannessa esimerkissä perusmallia vietiin eteenpäin liittämällä siihen toinen malli, luomalla halutut näkymät ja toiminnot käsin, sekä tekemällä lisätoimintoja kuten salasanavarmennus parametrien avulla.

On totta, että kirjan ohjeilla ei vielä suoranaisesti voi sanoa itseään Rails-asiantuntijaksi, mutta kirjan projekteissa läpikäydään kaikki perusasiat, joiden pohjalta on mahdollista ymmärtää kehittyneempiä konsepteja ja ryhtyä harjoittelemaan verkosta saatavien resurssien avulla. Esimerkiksi verkkosarjakuvia sisältävän sivun tekeminen vaatisi sitä, että malliin määritellään viestiksi datakenttä johon kuva tallennetaan, samalla tehden etusivun siten, että ainoastaan uusin tallennettu viesti näkyy. Vastaavasti verkkokauppa lähtisi rakentumaan lisäämällä ostoskorille kaksi parametria, joista ensimmäinen on ostoskorin sisältöä edustava taulukko ja toinen lisättävä tai poistettava tuote. Esimerkkien määrä on rajaton, ja itse asiassa hyvin äkkiä huomaakin, että CRUD-toimintojen avulla saadaan hyvin nopeasti toteutettua melkein mitä tahansa datankäsittelyä, oli kyseessä blogi, verkkokauppa, sarjisportaali tai keskustelupalsta.

Kirjassa on siis käsitelty kaikki, mitä Ruby-ohjelmointikielen ja Rails-kehyksen perusasioista pitää tietää. Tästä eteenpäin opettelua kannattaa jatkaa valitsemalla jokin mielenkiintoinen projekti tai idea, ja lähteä toteuttamaan sitä opiskelumielessä. Tämän kirjan perusasioiden ja verkosta saatavien lisäohjeiden avulla kehittyneemmät aiheet alkavat aueta hyvin nopeasti.

LIITE 1

Tarvittavien ohjelmien asentaminen

Ruby-ohjelmointiympäristön asennus
Ruby on Rails-kehyksen asennus

Tässä liitteessä opetetaan vaihe vaiheelta kaikkien työvälineiden asentaminen ja tarpeellisten asetusten määrittäminen kirjan tehtäviä varten.

RUBY-OHJELMOINTIYMPÄRISTÖN ASENNUS

Tässä liitteessä käydään kootusti läpi kaikkien kirjassa käytettävien Ruby-työkalujen asennusvaiheet. Näitä ohjeita noudattamalla on mahdollista ladata ja asentaa kerralla kaikki työkalut kirjassa esiteltyjen tehtävien tekemiseen.

Ruby-ohjelmointiympäristö

Tärkein työkalu, ja ensimmäinen verkosta ladattava paketti, onkin itse Ruby-ohjelmointiympäristö. Tämä paketti löytyy helpoiten Ruby-ohjelmointikielen kotisivuilta osoitteesta *www.ruby-lang.org*, yläpalkin linkin *Downloads (www.ruby-lang.org/en/downloads/)* kautta.

Ruby-yhteisön kotisivut

Downloads-sivulta löytyy lista erilaisista asennusversioista useille eri käyttöjärjestelmille, kuten Windows, Linux ja Solaris-käyttöjärjestelmiin. Versionumeroita voidaan tulkita siten, että pakettien nimet ovat aina muotoa **Ruby-X.X.X-pYYY,** jossa X.X.X kertoo tulkin versionumeron ja pYYY käännöksen versionumeron. Näistä tärkein on X.X.X-alkuosa; jälkimmäinen numero voi muuttua jos käännöksestä löydetään jotain korjattavaa tai paranneltavaa.

Lisäksi jokaiselle käyttöjärjestelmälle löytyy useimmiten useampi eri asennusversio; kirjan julkaisun aikaan 1.8.6, 1.8.7 sekä 1.9.1. Versioiden saatavuus voi myös vaihdella eri käyttöjärjestelmien välillä; Solarikselle perusversioksi suositellaan Ruby 1.8.7:aa, kun taas OS X ja Windows tarjoaa oletuksena 1.8.6:sta. Esimerkiksi tämän kirjan tehtävät on tehty versioilla 1.8.6 sekä 1.8.7, jotka ovat muutamia yksityiskohtia lukuun ottamatta keskenään yhteensopivia.

Rubya kokeilu- ja tutustumismielessä käyttävien kannattaa näistä versioista aina valita omalle käyttöjärjestelmälle tarkoitettu uusin vakaa, asennuspakettina saatava, versio. Esimerkiksi Windows-käyttöjärjestelmälle tämä löytyy kohdasta **Ruby on Windows** nimellä **One-click installer**. Tämä paketti on kooltaan keskimäärin 25 megatavua ja sisältää kaikki perustyökalut ja kirjastot, joita kirjan ensimmäisen osan tehtävissä tarvitaan. Ladataan asennuspaketti vaikkapa työpöydälle asennusta varten.

Windows-käyttöjärjestelmälle löytyy myös kolmannen osapuolen asennuspaketteja osoitteesta http://rubyinstaller.org/. Täältä löytyvät paketit sisältävät uudempia versioita Ruby-järjestelmästä, mutta eivät ole ”virallisia” Ruby-asennuspaketteja.

Kun paketti on ladattu, voidaan aloittaa itse asennus. Koska asennuksen yhteydessä kirjoitetaan asetuksia rekisteritietoihin, on ehdottoman tärkeää että asennus suoritetaan järjestelmänvalvojan

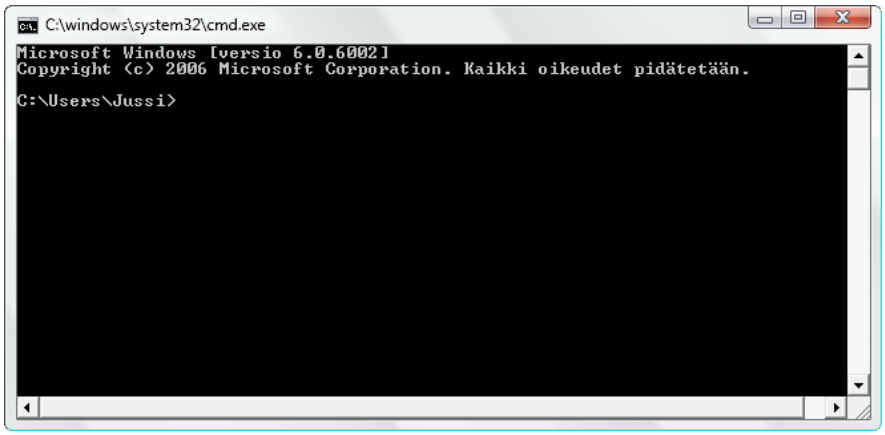
oikeuksilla. Muussa tapauksessa esimerkiksi komentorivikehotteen käskyt ei toimi, ja tämä aiheuttaa ongelmia viimeistään Rails-kehystä käytettäessä. Kun asennus aloitetaan, aukeaa ruutuun seuraavanlainen tervehdysikkuna:

Asennuksen aloitusikkuna

Tästä ikkunasta jatketaan eteenpäin painamalla ”Next >”, sekä seuraavassa ikkunassa ”I Agree”, jolloin tullaan ikkunaan jossa valitaan asennettavat komponentit. Valitaan pakollisen Ruby-paketin lisäksi SciTE, joka on Ruby-kielen mukana toimitettava lähdekoodieditori, sekä Enable RubyGems, joka mahdollistaa RubyGems-paketinhallintaohjelman käyttämisen myöhemmin esimerkiksi Rails-kehystä asennettaessa. Valitaan ”Next >”, sekä seuraavassa ikkunassa hyväksytään asennuspaikaksi oletussijainti *C:\Ruby*. Nyt painamalla ”Next >” Ruby aloittaa asennuksen.

Tässä vaiheessa asennusohjelma tekee rekisterimerkinnät, rekisteröi Ruby-lähdekooditiedostojen päätteet käyttöjärjestelmälle ja kopioi tiedostot. Koska tiedostoja on useita, voi tämä vaihe kestää muutaman minuutin. Kun työ on valmis, tulee ikkunaan teksti ”Completed” ja ”Next >”-painike aktivoituu. Painamalla tästä päästään asennuksen lopetusikkunaan, josta on halutessa mahdollista avata asennuksesta lisätietoja antava readme.txt. Painetaan ”Finish” jolloin asennusohjelma lopettaa. Koska asennuksen yhteydessä tehtiin rekisteriin muutoksia ja merkintöjä, joudutaan tässä vaiheessa käynnistämään tietokone uudelleen.

Kun tietokone on käynnistynyt uudelleen, tarkastetaan nopeasti että asennus on tapahtunut oikein. Painetaan ensin *Windows+R* ja kirjoitetaan aukeavaan ikkunaan *CMD* sekä painetaan Enter. Tämän pitäisi avata komentokehotteen ikkuna:



Windows-komentokehotteen ikkuna

Komentokehotteen käyttämisestä puhutaan lisää Rails-kehyksen käyttämisen yhteydessä, nyt kirjoitetaan kehotteeseen käsky *gem* ja painetaan *Enter*. Ruudulle pitäisi tulla kutakuinkin seuraava teksti:

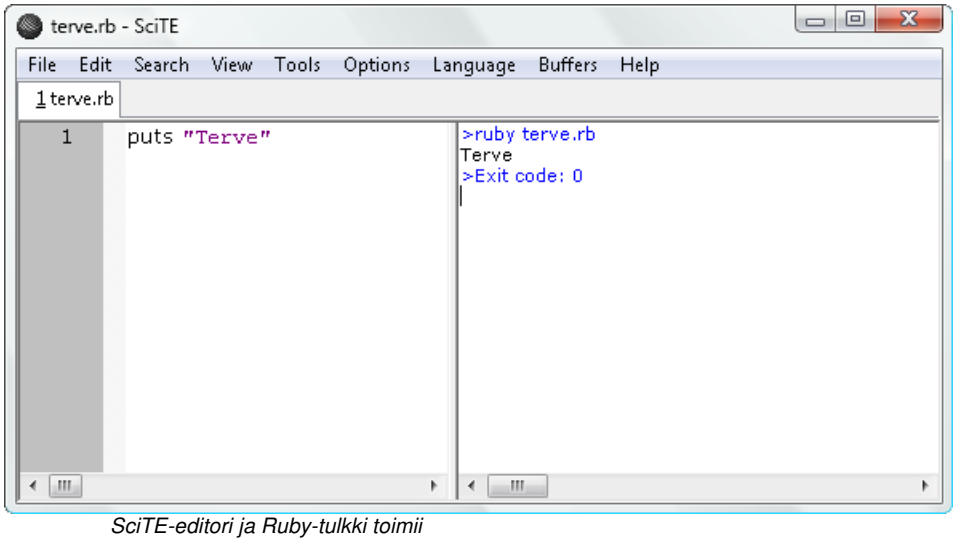
```
  RubyGems is a sophisticated package manager for Ruby. This is a
  basic help message containing pointers to more information.

[tästä leikattu joukko erilaisia käyttöohjeita]

    Further information:
      http://rubygems.rubyforge.org
```

Jos tämä teksti tulee ruudulle, tarkoittaa se sitä että komentokehotekäskyt on rekisteröity oikein. Jätetään tämä ikkuna auki taustalle, sillä sen avulla asennetaan hetken päästä itse Ruby on Rails-ohjelmointikehys.

Seuraavaksi tarkastetaan että paketin mukana asennetut työkalut ovat asentuneet oikein. Avataan Käynnistä-valikko ja tarkastellaan sen hakemistolistaa. Tästä listasta pitäisi löytyä kansio nimeltä *Ruby-XXX-YYY*, jossa *XXX-YYY* kuvaa asennetun Rubyn versionumeroa. Tästä kansiosta pitäisi myös löytyä kolme pikakuvaketta; *SciTE*, joka on Rubyn kanssa käytettävä lähdekoodieditori, *fxri*, joka on Ruby-kielen toiminnoista kertova interaktiivinen ohjekirja sekä *irb,* joka avaa Ruby-tulkin komentokehotteeseen. Lisäksi kansiossa on Ruby-tulkin poistamista varten erillinen kuvake, joka käynnistää asenuksen poistoon tarkoitetun työkalun. Valitaan pikakuvakkeista *SciTE*, jonka jälkeen SciTE-editori käynnistyy samankaltaiseen kuvaan kuin 1. luvun esimerkissä. Jos ohjelma käynnistyi ilman ongelmia, kirjoitetaan ensimmäiselle riville

```
puts "Terve"
```

ja tallennetaan ohjelma esimerkiksi työpöydälle (valikko *File*, toiminto *Save*) vaikkapa nimellä *terve.rb*. Tämän jälkeen valitaan valikosta *Tools* toiminto *Go* (pikanäppäin *F5*), jolloin editori suorittaa käskyn Ruby-tulkilla ja tulostaa seuraavan vastauksen:



SciTE-editori ja Ruby-tulkki toimii

Tässä vaiheessa on Ruby-ohjelmointiympäristön asennus saatu tehtyä sekä testattua. Seuraava vaihe on asentaa Ruby on Rails-kehys, joten mikäli tarkoituksena on tehdä kirjan alkuosan tehtäviä, voidaan tässä vaiheessa siirtyä takaisin lukuun 2.

RUBY ON RAILS-KEHYKSEN ASENNUS

Ruby on Rails on helpointa asentaa hyväksikäyttäen RubyGems-pakettienhallintatyökalua, joka toimii komentokehotteessa. Edellisessä työvaiheessa avattiin komentokehotteen ikkuna, joten jatketaan työskentelyä siellä. Koko rails-kehys ja suurin osa sen tarvitsemista peruspaketeista saadaan antamalla yksinkertaisesti seuraava komento komentokehotteessa ja painamalla *Enter*:

```
gem install rails
```

Tämä työvaihe lataa verkosta joukon paketteja ja asentaa ne osaksi Ruby-ohjelmointiympäristöä. Työvaihe kestää jonkin aikaa, eikä asennus alkuun näytä tulostavan mitään. Jonkin ajan kuluttua seuraavan tekstin pitäisi kuitenkin tulla näkyviin:

```
C:\Users\Jussi>gem install rails
Successfully installed rake-0.8.7
Successfully installed activesupport-2.3.8
Successfully installed activerecord-2.3.8
Successfully installed rack-1.1.0
Successfully installed actionpack-2.3.8
Successfully installed actionmailer-2.3.8
Successfully installed activeresource-2.3.8
Successfully installed rails-2.3.8
8 gems installed
Installing ri documentation for rake-0.8.7...
Installing ri documentation for activesupport-2.3.8...
Installing ri documentation for activerecord-2.3.8...
Installing ri documentation for rack-1.1.0...
Installing ri documentation for actionpack-2.3.8...
Installing ri documentation for actionmailer-2.3.8...
Installing ri documentation for activeresource-2.3.8...
Installing ri documentation for rails-2.3.8...
Installing RDoc documentation for rake-0.8.7...
Installing RDoc documentation for activesupport-2.3.8...
Installing RDoc documentation for activerecord-2.3.8...
Installing RDoc documentation for rack-1.1.0...
Installing RDoc documentation for actionpack-2.3.8...
Installing RDoc documentation for actionmailer-2.3.8...
Installing RDoc documentation for activeresource-2.3.8...
Installing RDoc documentation for rails-2.3.8...
```

Kun asennuksen ensimmäinen vaihe on valmis, päivitetään vielä kaikki Ruby-ympäristön paketit ajantasalle antamalla RubyGemsille komento päivittää itsensä ja kaikki Ruby-asennuspaketit komennolla

```
gem update --system
```

Tämän työvaiheen jälkeen asennetaan vielä SQLite-laajennus, jotta Rails-sovelluksille saadaan tehtyä helposti pieniä ja vaivattomia tietokantoja. SQLite:n asennuspaketit saadaan sivuilta *http://www.sqlite.org*, valitsemalla yläpalkista kohta *Download*.

l1k008.tif

SQLite-projektin kotisivut

Selataan Download-sivua alaspäin, kunnes päädytään kohtaan *"Precompiled Binaries For Windows"*. Näistä voidaan valita paketti *sqlitell-3_6_XXX.zip*, joka on kooltaan kutakuinkin 250 kilotavua ja jonka selite alkaa tekstillä "*This is a DLL of the SQLite library without the TCL bindings*.". Paketti sisältää tiedostot *sqlite3.dll* ja *sqlite3.def*. Ladataan paketti työpöydälle ja kopioidaan sen sisältämät tiedostot seuraaviin kansioihin:

```
<windowsin asennuskansio>\System32
<Rubyn asennuskansio>\bin
```

eli jos Windows-käyttöjärjestelmä ja Ruby-ympäristö ovat asennettu oletusarvoilla, ovat nämä kansiot

```
C:\Windows\System32
C:\Ruby\bin
```

Kopioinnin onnistumista varten tarvitaan järjestelmänvalvojan oikeudet. Lopuksi asennetaan vielä Rubygems-moduuli *sqlite3-ruby* komennolla

```
gem install sqlite3-ruby
```

Tämä työvaihe voi antaa joukon ”No definition”-tyyppisiä virheitä, mutta ne eivät johdu siitä että itse paketti olisi viallinen vaan että paketille ei ole olemassa interaktiivisen kirjaston hakemia dokumentaatioita. Toisinsanoen, näistä virheistä ei tarvitse tälläerää välittää. Kannattaa kuitenkin huomata, että koska tässä osassa asennettiin järjestelmäkansioon uusi kirjasto (sqlite3.dll), pitää tietokone käynnistää vielä kerran uudelleen ennen Rails-projektien aloittamista.

Kun tietokone käynnistyy uudelleen, ovat kaikki asennusvaiheet saatu valmiiksi, ja työpöydälle kertyneet asennuspaketit voidaan poistaa. Nyt kun myös Rails-kehys on toimintavalmiina, voidaan siirtyä takaisin kirjan varsinaisiin harjoituksiin.

LIITE 2

HTML-kuvauskielen perusteet

HTML-kuvausten tekeminen
HTML-tagit
CSS-tyyliohjeet
Esimerkkisivuja

Tässä liitteessä käsitellään lyhyesti HTML-kuvauskielen sekä CSS-tyyliohjeiden perusasiat, joiden avulla on mahdollista tehdä näkymien muotoilumäärittelyjä Ruby on Rails-sivustoille.

HTML-KUVAUSTEN TEKEMINEN

HTML (*Hypertext markup language)* on avoin kuvauskieli, jonka avulla määritellään selaimelle miltä verkkosivun tulisi näyttää. Tässä liitteessä käsitellään HTML-kuvauksen tekemistä ja kuvausmerkkien käyttämistä sen verran, että kirjan toisen osan on Rails-kuvausten määrittelyjen ymmärtäminen on mahdollista.

Perusteet

HTML-sivut sivut perustuvat elementteihin, joissa määritellään mikä osa kuvausta mikäkin teksti on. Käytännössä tämä tarkoittaa sitä, että ilmoitetaan mitä tekstiä seuraavaksi tulee, kirjoitetaan teksti ja merkataan että kuvaus on loppunut. Elementtien kuvausmerkit ovat aina < >-sulkeiden sisällä, ja sulkeva merkki alkaa </-muodossa, eli vaikkapa seuraavasti:

```
<otsikko> Tervetuloa sivuille </otsikko>
<viesti> Jotain muuta </viesti>
```

Käytännössä tämä tarkoittaisi sitä, että teksti ”Tervetuloa sivuille” on otsikko, ja sitä seuraava teksti ”Jotain muuta” on viesti, koska otsikko päättyy lopetusmerkkiin </otsikko>.

Tässä kappaleessa käytetyt kuvausmerkit ovat ymmärtämisen helpottamiseksi suomenkielisiä ”pseudomerkkejä”. Varsinaiset HTML-kuvausmerkit on listattu seuraavaan kappaleeseen.

Lisäksi kuvausmerkin yhteydessä voidaan antaa myös argumenttejä, kuten esimerkiksi kuvan yhteydessä annetaan kuvan sijainti, ja mahdollisesti haluttu leveys ja korkeus:

```
<kuva lähde= ”omakuva.png” korkeus= 100 leveys=100></kuva>
```

Lisäksi rakenteet voivat olla sisäkkäisiä: esimerkiksi linkkien tekemiseen muille sivuille on olemassa rakenne, johon laitettu rakenne toimii sivulla linkkinä, oli se sitten kuva tai tekstiä:

```
<linkki osoite= ”http://www.ruby-lang.org”>
<kuva ...></kuva> </linkki>
```

Sisäkkäiset rakenteet ovat keskeisessä osassa erityisesti tauluja (*table)* tehtäessä, koska siinä rivit ovat taulun sisällä ja sarakkeet rivien sisällä, mutta tähän palataan myöhemmin. Lisäksi html-kuvaus tavallisesti vielä erotellaan ainakin kahteen osaan; pääosa (head) jossa sijaitsevat sivua koskevat tekniset tiedot ja runko-osa (body) jossa itse sivun sisältö sijaitsee. Lisäksi vielä koko kuvauksen ympärille laitetaan erillinen html-merkki, joka määrittelee html-kuvauksen alkavan ja loppuvan:

```
<html>
 <pääosa>
    <otsikko> Tervetuloa! </otsikko>
 </pääosa>
 <runko>
   <teksti> Tässä on rautaisannos HTMLää. </teksti><rivinvaihto/>
   <teksti> Tämä on uudella rivillä. </teksti>
 </runko>
</html>
```

Esimerkistä kannattaa myös huomata <rivinvaihto/>, joka on yksiosainen itseensä loppuva merkki. Tällä merkillä ei ole ”lopetusosaa”, ja sitä merkataan päätteellä />. Lisäksi kannattaa huomata, että tekstieditorien muotoilut, rivinvaihdot, sisennykset ja vastaavat, eivät toistu HTML-kuvauksessa ellei niitä erikseen merkata niitä varten tarkoitetuilla HTML-kuvausmerkeillä.

Selaimen roolista

HTML-sivu siis kootaan elementeistä, jotka kertovat selaimelle minkätyyppisenä tekstinä tietyt osat sivusta tulisi käsitellä. Perusrungon lisäksi kuitenkin määrätään erikseen vielä pää- (*head*) ja tyyppi (*doctype*)-tiedot, joissa selaimelle kerrotaan mm. otsikkotiedot, mistä sivulla käytettävä CSS-tyyliohje löytyy sekä mitä html-tyyppiä käytetty kuvaus edustaa.

Koska html-standardista on useita eri versioita, on tavallista että verkkosivun alussa on kutakuinkin seuraavanlainen rivi:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/DTD/strict.dtd">
```

Tämä kertoo selaimelle, että sivun html-kuvaus on html-standardin version 4.01-mukainen ja käyttää kuvauksessa `strict`-kuvausmerkkisarjan merkkejä. Lisäksi selaimille vielä tavallisesti määritellään minkä tyyliohjeen mukaisesti html-elementit määritellään seuraavantyylisellä rivellä:

```
<link rel="stylesheet" href="tyylit.css"  type="text/css" />
```

CSS-tyyliohjeessa selaimelle kerrotaan, kuinka erilaiset html-elementit on tarkoitus esittää; fontti, kirjasinkoko, käytettävät värit ja vastaavat määritellään yhdellä kertaa tyyliohjeessa. Erilaiset selaimet saattavat kuitenkin tulkita ohjeita hieman eritavoin, johtaen siihen, että sivut saattavat selainohjelmaa vaihtamalla näyttää erilaisilta. Tähän ongelmaan on olemassa ratkaisu tekemällä sivusta eri selainohjelmilla tuotettavia versiota, mutta nykyisellään ongelma ei enää ole yhtä suuri kuin esimerkiksi vielä viisi vuotta sitten. Lisäksi yksi Ruby on Rails-sivustojen ominaisuuksista on siinä, että se on itse asiassa varsin hyvin eri selainohjelmia tukeva ympäristö, joten ongelmaa ei tavallisesti pääse muodostumaan.

Appletit ja koodi verkkosivuilla

HTML-kuvaukseen on mahdollista lisätä myös erilaisia ladattavia osia, kuten Javascript- tai PHP-koodia, Ruby on Rails-elementtejä tai vaikkapa ActiveX tai Java-appletteja lisäämällä niihin sopiva kuvausmerkki (RoR:llä <% %>, PHP:llä esimerkiksi <?php ?>). Tässäkin tapauksessa lopullinen komponentin sijoittelu selaimen ikkunaan riippuu siitä, miten muu html-kuvaus on aseteltu. Eniten html-sivun ulkonäkyyn vaikuttavat asiat ovatkin itse rungon muotoilutiedot sekä CSS-tyyliohje.

Lisäksi kannattaa huomata ero itse sivun tekemiseen käytettävän kuvauskoodin ja selaimen saaman tiedon välillä: Esimerkiksi on Rails- ja PHP-osiot kuvauskoodista eivät päädy vastaanottavalle selaimelle, vaan ainoastaan niiden koodilla generoitu html-kuvaus. Tämä sen vuoksi, että selain ei välttämättä ymmärrä kuin ainoastaan html-kuvauksia; on Rails- ja PHP-koodit suoritetaan palvelimen päässä ja selaimelle lähetetään ainoastaan koodien tuottama tulos html-kuvauksena jolloin erillisiä on Rails-ajoympäristöjä ei tarvita. Jos tätä verrataan esimerkiksi Java-appletteihin, lähetetään selaimen läpi koko valmis Java-ohjelma, mutta tällöin myös selaimen päästä pitää löytyä Java-ohjelman ajamiseen tarvittavat työkalut, kuten esimerkiksi Java Runtime Environment.

HTML-kuvausmerkit eli tagit

Mitä kuvausmerkkejä html-kuvaukseen sitten on mahdollista käyttää? Edellisessä kappaleessa käytettiin pääasiassa suomenkielisiä merkkien nimiä siksi, että asia olisi helpompi ymmärtää, mutta jos vastaavat nimet ajatellaan englanninkielisinä, ei merkit varsinaisesti eroa paljoakaan. Oheisessa taulukossa on listattuna HTML 4.01-kuvauksen keskeisimmät ja tärkeimmät kuvausmerkit:

Taulukko L2.1 Tärkeimmät HTML4.01-kuvaus- ja muotoilumerkit

Merkin tunnus	Selite	Esimerkki
<a>	Akkuri, jonka avulla voidaan luoda linkki toiseen html-sivuun tai sivun sisäiseen linkkin. Katso lisäksi ”esimerkit”.	`<a href= ”http://www.osoite.fi”> Linkki sivulle!</a>`
<b>	Tekstin lihavointi.	`<b>Lihavoitua tekstiä </b>`
<body>	Määrittelee html-dokumentin runko-osan, periaatteessa sisältö-osan, alueen.	
 	Rivinvaihto ilman kappaleväliä. Ei ”lopetusosaa”.	`Tähän rivinvaihto!  `

<h1>, <h2>...<h6>	Määrittelee otsikkotason tekstin.	`<h1>Pääotsikko</h1>` `<h2>Aliotsikko</h2>`
<head>	Määrittelee html-dokumentin pää-osan, jossa määritellään sivun otsikko ja tyyli- sekä lisätiedot.	
	Vaakaviiva; lisää tekstin jälkeen rivinvaihdon ja vaakaviivan. Ei ”lopetusosaa”.	`.. ja tämän jälkeen viiva` `<hr/>`
<html>	Määrittelee html-dokumentin. Aloittaa, ja lopetta html-kuvauksen, ainoastaan <!doctype-tulee ennen html-merkkiä.	
<i>	Tekstin kursivointi.	
<img>	Lisää tekstiin kuvan, ottaa argumentteina kuvan nimen, voi sisältää kokotiedot sekä korvaavan tekstin. Katso lisäksi ”esimerkit”.	`<img src=”kuva.jpg”` `alt=”tässä pitäisi olla kuva” />`
	Määrittelee tekstin listan merkinnäksi, lisää loppuun rivinvaihdon.	`<li>1. kohta</li>` `<li>2. kohta</li>`
<meta>	Sisältää meta-määrittelyjä koskien sivua ja sisältöä, esim. hakusanoja tai tekijätietoja.	
<p>	Lisää tekstiin kappaleenvaihdon (rivinvaihto ja puolikas rivi tyhjää tekstikenttien väliin).	
<table> (<td>, <tr>)	Määrittelee html-taulukon; tr määrittelee rivin, td yksittäisen solun riville. Katso lisäksi ”esimerkit”.	
<title>	Asettaa selaimen ikkunassa näkyvän sivun otsikon.	`<title> Jussin sivut` `</title>`
<!-- >	Kommentti, tämän merkin sisään kirjoitettua tekstiä ei huomioida sivuja luotaessa.	
<!DOCTYPE >	Määrittelee sen, mitä html-kuvauksen versiota käytetään. Tulee aina html-kuvauksen aivan ensimmäiseksi riviksi.	

HTML-kuvausmerkeistä kannattaa huomata, että erilaisia versioita html-kuvauskielestä on useita. Esimerkiksi nykyisellään eniten käytetystä HTML4.01-standardista löytyy kolme eri merkkijoukkoa (Strict, Transitional ja Frameset), jotka määrittelevät erilaiset käyttöympäristöt ja ominaisuudet, joissa merkkiä voi käyttää. Strict sisältää kaikki perusominaisuudet, transitional strictin lisäksi joukon poistuvia tai ei-suositeltuja toimintoja ja frameset mahdollisuuden käyttää frame-ikkunaelementtejä. Lisäksi on olemassa XHTML1.0 ja XHTML1.1-kuvaukset, jotka ovat yleisen XML-kuvauskielen johdannaisia jotka tuottavat HTML4.01-yhteensopivaa kuvausta. Lisäksi lähitulevaisuudessa tullaan näkemään myös HTML-kuvauskielen versio 5, joka tulee sisältämään jotain lisäyksiä html-kuvauskieleen, mutta joka muuten tulee olemaan täysin yhteensopiva version 4.01 kanssa.

Joka tapauksessa html-kuvausten määrittely on yksinkertaista kun niihin tottuu. Yksinkertainen HTML-sivu, joka täyttää kaikki html-kuvauksen osa-alueet, voitaisi toteuttaa vaikkapa seuraavalla tavalla:

```
01  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/DTD/strict.dtd">
02  <html>
03  <head>
04  <link rel="stylesheet" href="tyylit.css"  type="text/css" />
05  <meta name=”description” content=”HTML, yksinkertaisesti” />
06  <title> Minimalistinen HTML4.01-sivu </title>
```

```
07  </head>
08  <body>
09  <h1>Toden totta. </h1>
10  Näin on. <br/>
11  </body>
12  </html>
```

Tämä html-kuvaus sisältää tyyliohjeet ja määrittelee käytetyn kuvaustavan, toteuttaa pää- ja runko-osan sekä kirjoittaa kaksi riviä tekstiä, toisen otsikoituna h1-pääotsikkotasolle. Lisäksi sivulla on jopa meta-tietona annettu lyhyt kuvaus sivun sisällöstä. Tästä olisi hyötyä siinä tapauksessa, että sivu asetetaan verkkoon ja jokin hakukoneen selainrobotti sattuu päätymään sivulle. Itseasiassa, vanhemmat hakukoneet 1990/2000-luvun taitteessa jopa toimivat lähes yksinomaan meta- ja runko-tietojen varassa.

CSS-tyyliohjeet

Toinen keskeinen osa html-kuvausta on CSS-tyyliohjetiedosto, jossa määritellään minkänäköisiä html-elementtien tulisi olla. Vaikka CSS-tyyliohjeiden käyttö onkin teknisesti ollut mahdollista jo 13 vuoden ajan – ensimmäinen versio CSS-ohjeista julkaistiin jouluna 1996 – tapahtui varsinainen html-kuvauksen hajoaminen erillisiksi kuvaus- ja tyylitiedostoihin vasta 2000-luvun puolella; ensimmäinen kunnolla CSS-ohjeita tukeva selain ilmestyi vuonna 2000. Kuvaavaa onkin myös se, että uudemman ja laajemman CSS2-ohjeiston kaikkia toimintoja ei vielä kirjan kirjoitushetkellä toteuta yksikään olemassa oleva julkaistu Internet-selain.

Käytännössä CSS-ohjeet eivät ole teknisesti kovinkaan monimutkaiset. Käytännössä koko CSS-tiedosto onkin rakenteeltaan seuraavanlainen:

```
[elementin nimi] { attribuutti1:arvo ; attribuutti2:arvo... ;}
[elementin nimi] { attribuutti1:arvo ; attribuutti2... ;}
```

Eli periaatteessa vaikkapa seuraavanlainen:

```
h1 {color:#00ff12; font-size:30px;}
body {color:blue; font-family:”Verdana”;}
a:link {color:#303030;}
```

Tämä tyyliohje kertoo selaimelle, että h1-otsikot piirretään RGB-värillä `#00ff12` (tummahko virhreä) ja 30 pikseliä korkeina, runko-osan erikseen merkkaamaton teksti sinisenä, Verdana-fontilla ja avaamaton hyperlinkki värillä `#303030` (tummanharmaa). Ne elementit, tai elementin osat, joita CSS-tyyliohje ei määrittele, tässä esimerkissä vaikkapa h2-otsikot, toteutetaan selaimen oletusarvoilla. CSS-tyyliohjeeseen voidaan määrätä kaikille html-elementeille omat tyylit, mukaan lukien body, jolle voidaan antaa esimerkiksi taustakuva tai taustaväri CSS-ohjeen avulla, tai taulukot, joiden reunaviivojen paksuutta ja otsikoiden korostuksia voidaan muuttaa paljonkin tyyliohjeiden avulla..

Ruby on Rails-projekteissa käytettävät CSS-tyyliohjeet esitellään niihin liittyvien työvaiheiden yhteydessä tarkemmin. Seuraavassa kappaleessa käydään vielä joitakin esimerkkejä liittyen monimutkaisempien html-elementtien käyttämiseen.

Koska HTML4.01 ja CSS ovat molemmat avoimia standardeja, löytyy niihin verkosta paljon ilmaista ja käytännöllistä lisämateriaalia.

Esimerkkejä html-elementeistä

Tässä vielä lopuksi esimerkkejä joistakin monimutkaisemmista html-elementeistä, joiden käyttämisessä on välttämätöntä antaa lisäattribuutteja tai ymmärtää kuinka sisäkkäiset rakenteet toimivat.

IMG: Kuvan lisääminen

Mahdollisuus lisätä kuvia verkkosivulle on varmasti yksi tärkeimpiä ominaisuuksia, jotka aikanaan toivat Internetin korvikkeeksi Usenet- ja BBS-pohjaisille verkkopalveluille. Kuvien lisääminen onnistuukin html-kuvauksessa kohtuullisen helposti. IMG-merkki toimii siten, että siihen annetaan tietona esitettävän kuvan nimi, ja mahdollisesti lisämääreinä esimerkiksi kuvan käyttöön annetun tilan koko. Kuvan lisääminen onnistuu seuraavasti:

```
<img src=”<kuvan nimi>” alt=”<kuvaus jos kuvaa ei löydy>” />
```

eli vaikkapa

```
<img src=”terve.png” alt= ”Tervetuloa”/>
```

Tällöin mikäli palvelimella on html-kuvauksen kanssa samassa hakemistossa terve.png-niminen kuva, se laitetaan html-kuvauksen merkin paikalle. Jos kuvaa ei löydy, tulee kuvan tilalle teksti ”Tervetuloa”. Lisäksi voidaan antaa seuraavat lisäparametrit:

- `height`: Kuvan maksimikorkeus pikseleinä tai prosentteina kuvan alkuperäisestä koosta. Kuvaa skaalataan alaspäin jos annettu maksimikorkeus on pienempi kuin kuvan alkuperäinen korkeus.
- `width`: Kuvan maksimileveys, toimii samalla tavoin kuin height.

A: Linkin lisääminen

Linkkien määritteleminen ja lisääminen on html-kuvauksissa se asia, joka erottaa html-dokumentin tavallisesta tekstikentästä. Linkkejä lisätään <a>-merkillä, johon määritellään kohde, johon linkki osoittaa, sekä mahdollisesti ikkuna, johon likki avataan. Linkin lisääminen tehdään seuraavasti:

```
<a href=”<osoite>”>Linkin teksti tai kuva</a>
```

eli esimerkiksi

```
<a href=”http://www.ruby-lang.org”>Rubyn kotisivut</a>
```

Tällöin selain luo tekstiin ”Rubyn kotisivut” –tekstipätkän, jossa on hyperlinkki Ruby-ohjelmointikielen kotisivuille. Lisäksi kuvia voidaan käyttää linkkeinä laittamalla <a>-merkkien väliin <img>-merkki:

```
<a href=”http://www.kotisivut.joo”><img src=”apina.png”/></a>
```

Tällöin sivuille tuleva kuva toimii linkkinä annettuun osoitteeseen.

TABLE: Taulukon määritteleminen

Kaikista html-elementeistä taulukon tekeminen on varmaankin samalla kaikista vaivalloisinta, mutta myöskin tehokkain tapa saada sivulle yksinkertainen, mutta samalla pitävä muotoilu. Taulu-elementti toimii siten, että ensin määritellään taulun kuvaus <table>-muotoilumerkillä, ja tämän jälkeen rivit <tr></tr>-muotoilumerkkiparilla. Tämän lisäksi <tr>-merkkien väliin määritellään <td>-merkeillä rivin sarakkeet, jokaiselle riville erikseen. Asia on kuitenkin lienee helpointa hahmottaa käytännössä:

```
<table border= ”1”>
  <tr>
    <td> 1. Rivin 1. solu </td>
    <td> 1. Rivin 2. solu </td>
  </tr>
  <tr>
    <td> 2. Rivin 1. solu </td>
    <td> 2. Rivin 2. solu </td>
  </tr>
 <tr>
    <td> 3. Rivin 1. solu </td>
    <td> 3. Rivin 2. solu </td>
 </tr>
</table>
```

Tuottaa seuraavantyylisen taulukon:

1. Rivin 1. solu	1. Rivin 2. solu
2. Rivin 1. solu	2. Rivin 2. solu
3. Rivin 1. solu	3. Rivin 2. solu

Lisäksi <td>-merkkien kanssa vaihtoiehtoisesti voidaan käyttää <th>-merkkiä, joka kuvaa taulukon otsikkorivin sarakkeita, antaen niille hieman erilaisen muotoilun. Jos taulukkoon halutaan lisää rivejä, lisätään kuvaukseen <tr>-merkkien muodostamia kokonaisuuksia, jos lisää sarakkeita, lisätään jokaisten <tr>-merkkien väliin uusi <td>. Taulukon luovaan <table>-merkkiin voidaan lisäksi antaa parametreina seuraavia arvoja:

- `border`; määrittelee pikseleinä taulukon solujen reunan paksuuden.
- `cellpadding`: määrittelee pikseleinä sen, minkä verran tyhjää pitää solujen sisällön ja reunan väliin vähintään jäädä-
- `cellspacing`: määrittelee pikseleinä sen, kuinka paljon tyhjää solujen väliin jätetään.
- `width`: määrittelee kuinka leveä taulusta tehdään. Arvo voidaan antaa joko pikseleinä, tai prosentteina selaimen ikkunan alasta (0-100%).

Kannattaa kuitenkin huomata, että taulujen tärkein käyttökohde ei suoranaisesti ole mukavien taulukoiden tekemisessä, vaan niitä voidaan hyödyntää verkkosivun asettelun rakentamiseen: Koska sivut monesti rakentuvat keskenään erikokoisista elementeistä, on taulu tehokas apuväline pitämään sivut jonkinlaisessa muodossa. Taulu ei välitä siitä mitä sen soluihin laitetaan, vaan se voi olla toinen taulu, kuvia, linkkejä tai vaikkapa Ruby on Rails-elementti. Tauluun asetettuina elementit, jotka ovat samalla rivillä asettuvat aina samalle korkeudelle lopullisella sivulla. Samoin allekkaisissa soluissa olevat elementit ovat aina allekkain, huolimatta siitä voisiko ne fyysisesti mahtua olemaan vaikkapa rinnakkain. Taulu, jolle annetaan leveydeksi vaikkapa 95% muodostaakin hyvän ”pohjan”, johon määritellä muut elementit siten että ne eivät siirtyile pakoiltaan selaimen ikkunan koon muuttuessa. Tätä tekniikkaa käytetään hyväksy mm. kirjan osassa 2, blogi-projektin etusivulla.