# 1. Omat operaatiot

# Sisällys

- Yleistä operaatioista.
- Mihin operaatioita tarvitaan?
- Oman operaation määrittely.
  - Yleisesti, nimeäminen ja hyvä ohjelmointitapa, määreet, parametrit ja näkyvyys.
- *HelloWorld*-ohjelma tarkemmin.
- Oman operaation paikka ohjelmassa, operaation kutsuminen ja paluuarvo esimerkkien avulla tarkasteltuna.
- Operaatioiden kutsuminen yleisesti.

# Yleistä operaatioista

- Operaatio (metodi, aliohjelma, funktio jne.) eristää jonkin luokan osan nimettyyn lohkoon.
  - *Main*-operaatio tunnistetaan nimestä, johon liittyy lohko.
- Operaatiota kutsutaan luokan ulkopuolelta operaation nimellä, pistenotaatiota käyttäen.
- Kaarisulkupari erottaa operaation ja muuttujan tunnukset.
- Operaatiolle voidaan tarjota tietoja eli parametreja.
- Kutsun kohdalla ohjelman suoritus siirtyy operaatioon, josta palataan kutsukohtaa seuraavaan ohjelman vaiheeseen.
- Esimerkiksi: System.out.println("Moikka!");

# Yleistä operaatioista

- Operaatio voi palauttaa arvon.
- Esimerkiksi: **int** luku = In.readInt();
- Näkyvyyssääntöjen mukaan operaation lohkossa esitellyt tunnukset eivät näy operaation ulkopuolelle.
- Operaatiotkaan eivät välttämättä näy luokan ulkopuolelle.
  - Operaatio voidaan kätkeä tai julkaista näkyvyysmääreillä.
  - *Main*-operaatio on julkinen (**public**), jotta Java-tulkki voi kutsua sitä.

# Mihin operaatioita tarvitaan?

- Operaatioilla pitkä ohjelma voidaan jakaa “palasiin”, jolloin koodia on helpompi hallita, ymmärtää ja ylläpitää.
  - Monimutkaisuuden hallinta modulaarisella ohjelmoinnilla.
- Ohjelman eri kohdissa samana toistuva osa voidaan sijoittaa operaatioon, jolloin koodia ei tarvitse “monistaa” eri kohtiin, vaan pelkkä operaation kutsu riittää ohjelman osan suoritukseen.
  - Ohjelman ylläpito helpottuu.
- Mahdollistavat tiedonkätkennän.
  - Tarkemmin Olio-ohjelmoinnin perusteet -kurssilla.

# Operaatioiden määrittely

- Koostuu otsikosta (header, signature) ja rungosta (body):

```
// Otsikossa määreitä, operaation nimi (tunnus) ja parametrilista.
määreet nimi(parametrilista) {
   // Runko on koottu lause (lohko), joka kirjoitetaan
   // luokkamäärittelyn sisään. Sulkeita ei voi jättää pois.
}
```

- 
```
// Tervehditään ohjelman käyttäjää.
public static void tulostaTervehdys() {
    System.out.println(Moi!); …
}
```

# Operaatioiden määrittely: nimet

- Nimet (tunnukset) ovat yleensä muuttujien nimiä pitempiä.
  - Esimerkiksi: *syote* ja *tarkistaSyote*.
- Kirjoitetaan samoin kuin muuttujat.
  - Nimi aloitetaan pienellä alkukirjaimella.
  - Useasta sanasta koostuvassa nimessä ensimmäinen sana pienellä ja muut isolla alkukirjaimella. (Yhdyssanat poikkeus.)
- Operaatiot kuvaavat toimintoja – tunnuksessa on usein käskymuotoinen verbi.
- Hyvä ohjelmointitapa koskee myös operaatioita.
  - Otsikkoon liittyvä kommentti kertoo suppeimmillaan mitä operaatio tekee.
  - Koottu lause sisennetään ja sen sisältö kommentoidaan.

# Operaatioiden määrittely: määreet

- Määreillä otetaan kantaa muun muassa operaation näkyvyyteen ja operaation mahdollisesti palauttamiin tietoihin.
- **Private**-määreellä operaatio on yksityinen eli sitä voi kutsua vain luokan omissa operaatioissa.
- **Public**-määreellä operaatio on julkinen. Sitä voidaan kutsua pistenotaatiolla myös luokan ulkopuolelta.
- Arvon palauttavalle operaatiolle määritellään tyyppi, joka voi olla mikä tahansa Javan tietotyyppi.
  - Otsikon kommentissa selitetään minkä tiedon paluuarvo sisältää.
- Jos operaatiolla ei ole paluuarvoa, tietotyypin asemasta käytetään **void**-määrettä.
- Tällä kurssilla tarvitaan **static**-määre, jotta ohjelma kääntyisi.

# Operaatioiden määrittely: parametrit

- Parametrilista määrittelee operaatiolle välitettävät tiedot.
  - Lista pilkuin erotettuja parametreja, jotka nimetään ja esitellään kuten muuttujat.
  - Toisinaan käytetään lyhyitä tunnuksia, koska parametrien merkitys tulee selittää otsikkoon liittyvässä kommentissa.
  - Pelkillä suluilla ilmaistaan ettei operaatiolle välitetä tietoja.
- /* Korottaa kantaluvun a potenssiin b ja palauttaa tuloksen.
  */

```
public static int laskePotenssi(int a, int b) {
  …
}
```

# Operaatioiden määrittely: näkyvyys

- Parametrit ja muuttujat eivät näy muihin operaatioihin.
- Mikäli operaation parametrilla ja muuttujalla on sama nimi, tapahtuu nimikonflikti, eikä ohjelma käänny.
- // Korottaa kantaluvun a potenssiin b ja palauttaa tuloksen.

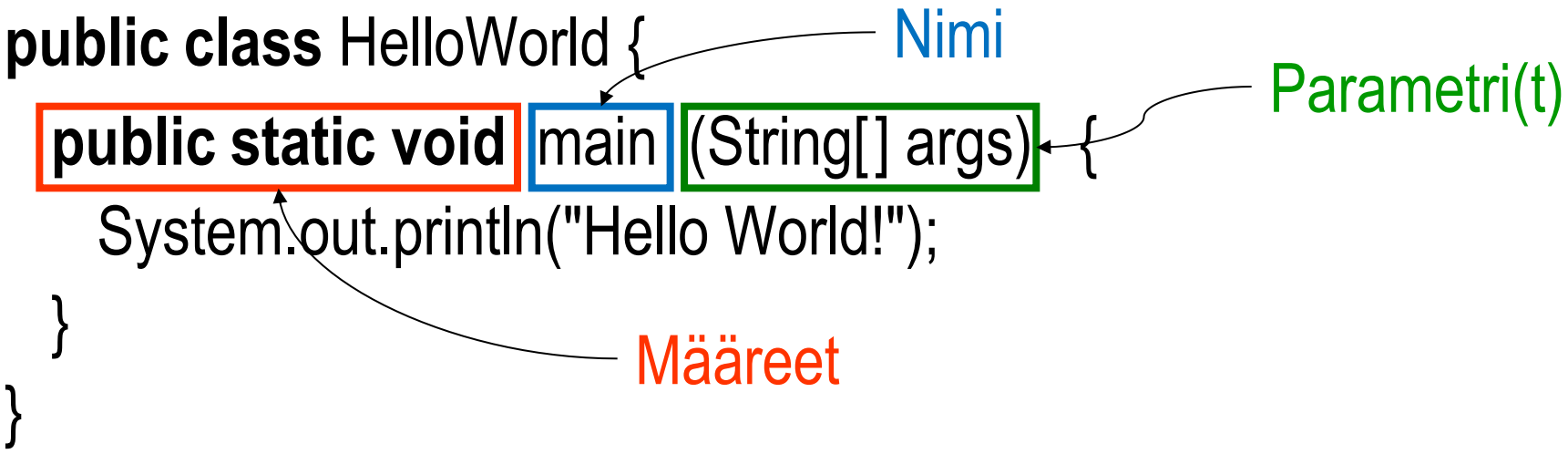
```
public static int laskePotenssi(int a, int b) {
  // Toisen operaation parametri ja  muuttuja eivät näy tänne.
  int tulos = 0; …
}

  // Tarkistaa syötteen. Paluuarvo on true, jos syöte on kunnossa.
  public static boolean tarkistaSyote(String syote) {
    // Ensimmäisen operaation parametrit ja muuttuja eivät näy tänne.
    boolean tulos = true; …
```

# *HelloWorld*-ohjelma

- Jatketaan *HelloWorld*-ohjelman analyysiä.
- Aikaisemmalta kurssilta muistetaan, että
  - ajokelpoisessa Java-ohjelmassa on oltava *main*-operaatio eli niin sanottu pääohjelmaoperaatio,
  - *main*-operaation otsikon on oltava aina tietyn muotoinen, jotta ohjelman suoritus onnistuisi,
  - *main*-operaatio sijaitsee luokkamäärittelyn sisässä ja
  - *main*-operaation lohko sijaitsee luokan lohkon sisässä.

# *HelloWorld*-ohjelma




- *Main*-operaatio on julkinen (**public**) eikä sillä ole paluuarvoa (**void**).
- *Main*-operaatiota kutsutaan automaattisesti. Tästä nimitys pääohjelmaoperaatio.
- Operaation parametreista kerrotaan myöhemmin.

# Omat operaatiot: *sayHello*-operaatio

```
/* Tervehditään maailmaa
 * englanniksi. Huomaa static-
 * määre operaation otsikossa.
 */
public static void sayHello() {
    // Tekstiä näytölle.
    System.out.println("Hello World!");
}
```

- Ennen oman operaation otsikkoa kirjoitetaan aina yleisluonteinen kommentti operaation tarkoituksesta sekä mahdollisista parametreista ja paluuarvoista.
- Oheinen operaatio ei palauta arvoa (**void**).
- Operaatiolle ei myöskään voi antaa parametreja (parametrilista on tyhjä).

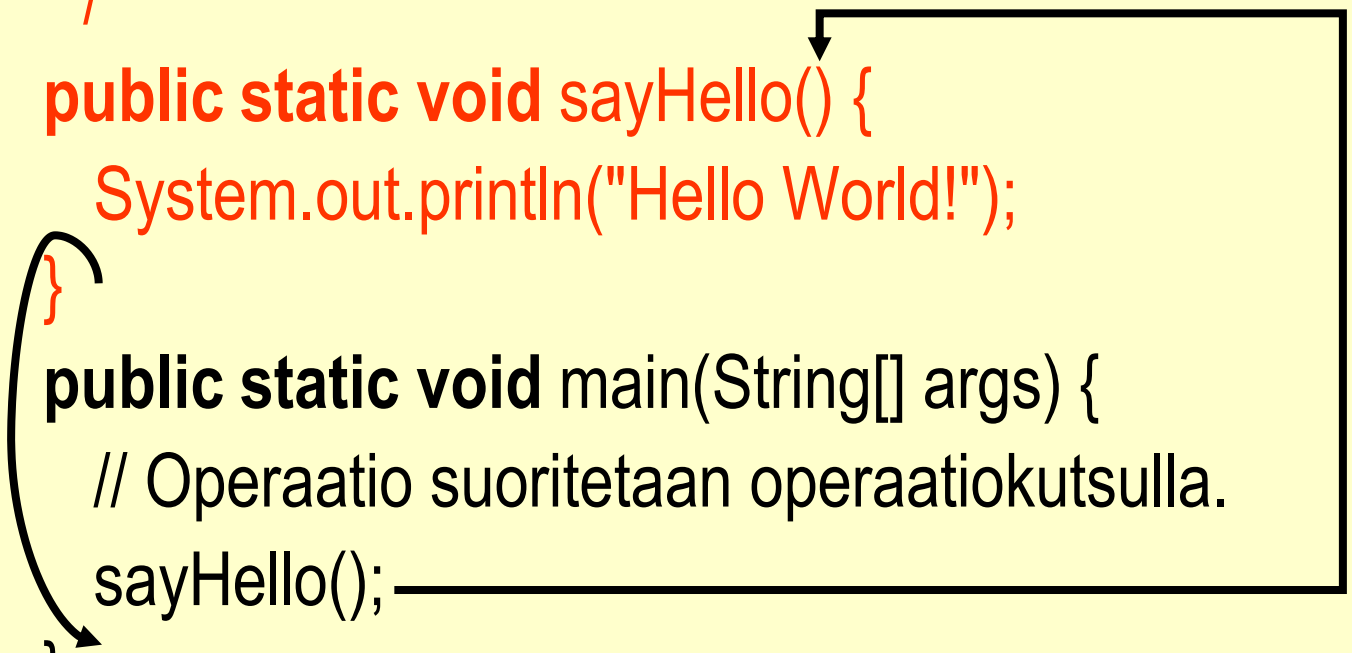
# Omat operaatiot: paikka luokassa

```
public class HelloWorld {
  /* Tervehditään maailmaa englanniksi.
   */
  public static void sayHello() {
    System.out.println("Hello World!");
  }
  public static void main(String[] args) {
    // Kuinka sayHello voidaan suorittaa?
  }
}
```

- Omat operaatiot kirjoitetaan tällä kurssilla samaan luokkaan kuin *main*-operaatio.
- Otsikoihin on siksi muistettava liittää **static**-määre. Koodi ei käänny, jos **static**-määre puuttuu.

# Omat operaatio: kutsuminen

```
public class HelloWorld {
   /* Tervehditään maailmaa englanniksi.
    */
   public static void sayHello() {
      System.out.println("Hello World!");
   }
   public static void main(String[] args) {
      // Operaatio suoritetaan operaatiokutsulla.
      sayHello();
   }
}
```



# Omat operaatiot: kutsuminen

- *SayHello*-operaatiota kutsuttiin luokan sisällä toisesta operaatiosta nimellä ilman pistenotaatiota.
- Operaatiolle ei välitetty parametreja: kutsulause koostuu operaation nimestä ja “tyhjistä” suluista.
- Ohjelman suoritus palasi *main*-operaatioon automaattisesti.
- Operaatio ei palauttanut arvoa (**void**-määre).
- Operaatio olisi voitu sijoittaa myös *main*-operaation jälkeen.
  - Javassa operaatiota ei ole pakko määritellä ennen kuin sitä kutsutaan.

# Omat operaatiot: kutsuminen

- Operaatiolle välitettävät tiedot määritellään parametrilistan avulla.
- Parametri saa arvonsa operaation kutsun yhteydessä.
- Parametriksi voidaan antaa arvo sellaisenaan (literaali), muuttuja, vakio, lauseke, toisen operaation paluuarvo jne.
- Arvoja on oltava oikea määrä, arvojen on oltava oikean tyyppisiä ja arvojen on oltava oikeassa järjestyksessä, jos järjestyksellä on väliä.
- Kutsun yhteydessä ei pidä kirjoittaa näkyviin arvon tyyppiä.
- Kutsussa käytettäviä muuttujia ei tarvitse nimetä samoin kuin parametreja, koska kutsuvan operaation tunnukset eivät näy kutsuttavaan metodiin ja päinvastoin.
- Operaation paluuarvo voidaan sijoittaa muuttujaan, jos tyypit ovat yhteensopivat.

# Omat operaatiot: kutsuminen

- Esimerkiksi potenssin $a^b$ laskevan operaation otsikossa:

  **public static int** laskePotenssi(**int** a, **int** b)

  määritellään **int**-tyyppiset parametrit *a* ja *b*.
- *LaskePotenssi*-operaatiota esimerkiksi *main*-operaatiosta kutsuttaessa on annettava kaksi **int**-tyyppistä arvoa (myös **byte**- ja **short**-arvot käyvät) siten, että kantaluku on ensimmäisen parametrin arvo arvo ja eksponentti toisen parametrin arvo.

  … laskePotenssi(2, 3) … // Kaksi potenssiin kolme.

  … laskePotenssi(3, 2) … // Kolme potenssiin kaksi.

  … laskePotenssi(2.5, 2) … // Virhe, 1. parametriarvo väärää tyyppiä.

  … laskePotenssi(**int** 2, **int** 3) … // Virhe, arvojen tyypit annettu.

# Omat operaatiot: kutsuminen

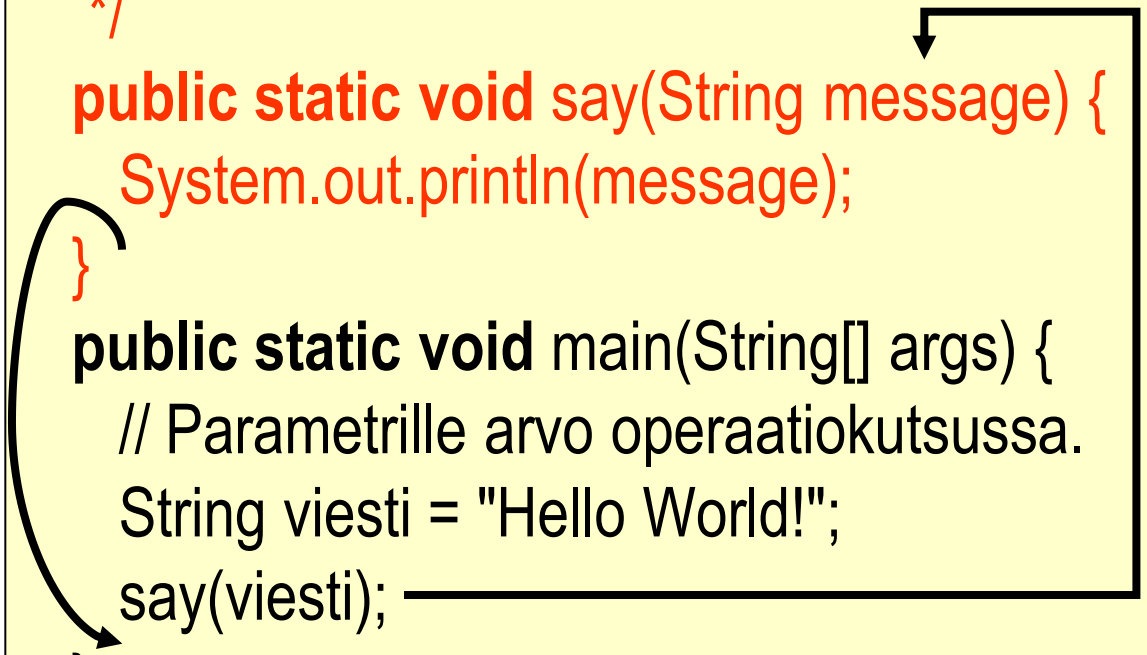
- Edellä määriteltyä operaatiota kutsuttaessa voi muuttujien nimet valita vapaasti. Esimerkiksi *main*-operaatiossa:

```
int kantaluku = 2, eksponentti = 3;
// Parametrien tunnuksia a ja b ei tarvitse käyttää, koska ne eivät näy
// potenssin laskevasta operaatiosta tänne.
… laskePotenssi(kantaluku, eksponetti);  …
```

- Java luo parametrille annetusta arvosta kopion ja sijoittaa sen automaattisesti parametrin tunnukseen liittyväksi arvoksi (pass-by-value, call-by-value).
  - Koska operaatio käsittelee kopiota, eivät operaation saamaansa arvoon tekemät muutokset välity alkuperäiseen arvoon.

# Omat operaatiot: kutsuminen

```
public class HelloWorld {
  /* Tervehditään maailmaa operaation
   * kutsujan antamalla viestillä .
   */
  public static void say(String message) {
    System.out.println(message);
  }
  public static void main(String[] args) {
    // Parametrille arvo operaatiokutsussa.
    String viesti = "Hello World!";
    say(viesti);
  }
}
```




- Otsikossa määritellään, että operaatiolla on String-tyyppinen parametri.
- Operaatiolle annetaan sitä kutsuttaessa String-tyyppinen muuttuja, jonka arvo "Hello World!" kopioidaan ja sijoitetaan parametrin arvoksi automaattisesti.

# Omat operaatiot: parametrit

```
…                                                    3
// Tervehditään n (> 0) kertaa.
public static void sayHello(int n) {
   while (n > 0) {
      System.out.println("Hello World!");
      n--;
   }
}
public static void main(String[] args) {
   int moilkm = 3;
   sayHello(moilkm);
   System.out.println(moilkm);  // 3
} …
```

- Muuttujan arvo kopioidaan ja sijoitetaan parametrin arvoksi automaattisesti.
- Parametrin *n* arvoon tehdyt muutokset eivät vaikuta muuttujan *moilkm* arvoon, koska arvot ovat erilliset.

# Omat operaatiot: parametrit

```
/* Tulostaa näytölle annetulla merkillä n riviä korkean
 * ja m saraketta leveän suorakaiteen.
 */
public static void piirraSuorakaide(int n, int m, char
    merkki) {
  // Parametrien oltava järkevät, jotta edes yritetään
    piirtämistä.
  if (n > 0 && m > 0)
    for (int i = 0; i < n; i++) {                  // Tulostetaan
    n riviä.
      for (int j = 0; j < m; j++)       // Tulostetaan
    yksittäinen
        System.out.print(merkki);  // rivi eli m merkkiä.
      System.out.println();
  }
```

- Parametrien arvot on syytä tarkistaa operaatiossa, jos virheelliset arvot ovat mahdollisia, jotta ohjelmasta saadaan vakaampi.

# Omat operaatiot: paluuarvo

- Operaation tyyppi määrää palauttaako operaatio arvon vai ei.
- **Void**-tyyppinen operaatio ei palauta arvoa.
  - Ohjelman suoritus palaa automaattisesti kutsupaikkaan, kun operaation viimeinen lause on suoritettu.
- Muun tyyppiset operaatiot palauttavat arvon **return**-lauseella.
  - **Return**-lause palauttaa ohjelman suorituksen kutsupaikkaan.
  - Yleisesti: **return** arvo;
- Paluuarvo voidaan tallentaa normaaliin tapaan muuttujaan.
  - Paluuarvon ja muuttujan tyyppien tulee olla yhteensopivat.

# Omat operaatiot: paluuarvo

- Esimerkiksi potenssin $a^b$ laskevan operaation otsikossa:

  **public static int** laskePotenssi(**int** a, **int** b)

  määritellään operaation tyypiksi **int**, jolloin operaation on aina palautettava **return**-lauseella **int**-tyyppinen arvo.

- Sijoitetaan eri tavoin kutsutun operaation paluuarvo muuttujaan esimerkiksi *main*-operaatiossa.

```
int kantaluku = 2, eksponentti = 3;
short i = 10;
int potenssi = laskePotenssi(2, 3);
potenssi = laskePotenssi(kantaluku, eksponetti);
potenssi = laskePotenssi(i - 1, i + 1);
```

# *Max*-operaatio

```
/* Palauttaa suurimman luvuista a, b ja c.
 */
public static int max(int a, int b, int c) {
      int m;
      if ((a > b) && (a > c))
         m = a;
      else if (b > c)
         m = b;
      else
         m = c;
      return m;
}
```

- Parametrilistassa määritellään, että operaatiolle on annettava kolme **int**-tyyppistä arvoa. Arvojen järjestyksellä ei ole väliä.
- Operaatio on **int**-tyyppinen, jolloin operaatiossa on oltava arvon palauttava **return**-lause.

# *Max*-operaatio

```
/* Palauttaa suurimman luvuista a, b ja c.
 */
public static int max(int a, int b, int c) {
    if ((a > b) && (a > c))
        return a;
    else if (b > c)
        return b;
    else
        return c;
}
```

- Tässä *max*-operaation versiossa arvo palautetaan jokaisessa päätös-haarassa saman tien.

# *Max*-operaation kutsuja *main*-operaatiosta

```
public static void main(String[] args) {
   int k = 1, l = 3, m = 2;
   // Parametriarvoiksi muuttujien arvot, paluuarvo muuttujaan.
   int n = max(k, l, m);
   System.out.println(n);
   // Parametriarvoina literaaleja ja muuttujan arvo.
   // Paluuarvo katoaa, koska sitä ei sijoiteta muuttujaan.
   max(111, 222, l);
   // Paluuarvo annetaan toisen operaation parametriarvoksi.
   System.out.println(max(111, 222, 333));
}
```

# Operaation kutsuminen yleisesti

- Luokan *A* operaatioita voi kutsua kolmella tavalla:
  1) Toisesta luokasta *A*-tyyppisen olion kautta.
  2) Toisesta luokasta luokan *A* nimen avulla.
  3) Luokan *A* toisista operaatioista.
- Kohdissa 1 ja 2 voidaan ajatella abstraktisti, että ympäristöstä lähetetty viesti käynnistää operaation.
- Käytännössä operaation kutsuminen tapahtuu tutulla pistenotaatiolla.
- Omia operaatioita kutsutaan tällä kurssilla vain kohdan 3 tavalla, koska kutsuminen luokan ulkopuolelta ei onnistu, kun käytettävissä on vain yksi itse kirjoitettu luokka.

# Operaation kutsuminen olion kautta

- Yleisesti:

  **olio.operaatio(parametrit);**

  missä parametrit ovat operaation parametrilistassa annettua tyyppiä olevia arvoja (literaaleja, vakioita, muuttujia jne.).
- Parametreja ei luonnollisesti anneta, jos operaation parametrilista on tyhjä.
- Esimerkiksi:

```
String nimi = "Minni";
char ekaMerkki = nimi.charAt(0);
```

# Operaation kutsuminen luokan kautta

- Yleisesti:

  **Luokka.operaatio(parametrit);**

  missä parametrit ovat kuten edellä.
- Operaatio on esiteltävä **static**-määreellä, jotta luokan nimellä kutsuminen onnistuisi.
- Tällaiset luokkaoperaatiot ovat hyödyllisiä, kun
  - kootaan yhteen toimintoja, joiden käyttö sujuvampaa suoraan ilman olion esittelyä ja
  - on tarpeen määritellä koko luokalle yhteisiä vakioita.

# Operaation kutsuminen luokan kautta

- Luokkaoperaatiot ovat tuttuja jo *Math*- ja *In*-luokkien yhteydestä.
- Esimerkiksi:

```
double hinta = In.readDouble();
int itseisarvo = Math.abs(luku);
```

- Luokkaoperaatioista voidaan kutsua vain toisia luokkaoperaatioita.
- Tästä syystä *main*-operaation sisältävässä luokassa operaatioiden (ja attribuuttien) esittelyihin on lisättävä **static**-määre.

# 2. Taulukot

# Sisältö

- Yleistä.
- Esittely ja luominen.
- Alkioiden käsittely.
- Kaksiulotteinen taulukko.
- Taulukko operaation parametrina.
- Taulukko ja *HelloWorld*-ohjelma.
- Taulukko paluuarvona.

# Yleistä

- Suurten tietomäärien esittäminen erillisinä arvoina ei ole käytännöllistä.
- On helpompaa koota arvot yhdeksi kokonaisuudeksi eli tietorakenteeksi (data structure), joka mahdollistaa tietojen helpomman organisoinnin keskusmuistiin.
- Tietorakenteita on paljon ja ohjelmointiongelma ratkaisee pitkälti mitkä ovat sopivia tai parhaita.
  - Tarkemmin Tietorakenteet-kurssilla.
- Nyt käsitellään vain taulukko (array), joka on saman tyyppisten muuttujien eli alkioiden (element) kokoelma.

# Yleistä

- Taulukon alkioilla on järjestys.
- Alkiot indeksoidaan 0, ... , *n* - 1, missä *n* on alkioiden lukumäärä (vertaa merkkijono).
- Esimerkki. **Int**-tyyppiset luvut 12, 56, 34 ja 78 sisältävä yksiulotteinen taulukko:

| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Alkio | 12 | 56 | 34 | 87 |

# Esittely ja luominen

- Esitellään hakasulkuja ([]) käyttäen.

  Yleisesti: **tyyppi[] taulukonNimi**;
- Taulukko on viitetyyppiä.
  - Java varaa esittelyn yhteydessä automaattisesti muistia viitteelle, mutta viitteeseen liittyvä olio on luotava **new**-operaatiolla käyttäjän toimesta.
- Luomisen yhteydessä määritellään myös taulukon koko (alkioiden lukumäärä).
- Yleisesti: **taulukonNimi = new tyyppi[koko]**;
- Taulukon alkiot alustuvat automaattisesti.

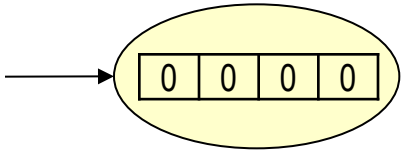
# Esittely ja luominen

- **int**[] satunnaisluvut;

satunnaisluvut ⟶ **null**

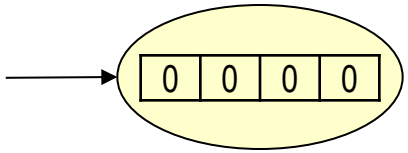
1) Java varaa esittelyn yhteydessä muistia viitteelle ja alustaa viitteen **null**-arvolla.

- satunnaisluvut = **new int**[4];




2) Lauseke **new int**[4]; luo taulukko-olion, alustaa alkiot ja palauttaa olioon liittyvän tunnuksettoman viitteen.

satunnaisluvut




3) Sijoituksen seurauksena *satunnaisluvut*-viite liittyy samaan olioon kuin paluuarvona saatu viite.

# Esittely ja luominen

- Taulukko voidaan alustaa luonnin yhteydessä, jolloin koko määräytyy automaattisesti. Yleisesti:

  **taulukonNimi = new tyyppi[] { arvo1, … , arvo*N* }**;

- Esimerkki. // Esitellään taulukko.

  **int**[] satunnaisluvut;

  // Luodaan 4 alkion kokoinen taulukko.

  satunnaisluvut = **new int**[4];

- Esimerkki. **int**[] satunnaisluvut;

  satunnaisluvut = **new int**[] { 12, 56, 34, 87 };

# Esittely ja luominen

- Taulukon esittely, luominen ja mahdollinen alustus voidaan yhdistää samaan lauseeseen.
- Esimerkki. **int**[] satunnaisluvut = **new int**[4];

  **int**[] satunnaisluvut = **new int**[] { 12, 56, 34, 87 };

  // Esittely, luominen ja alustus lyhemmin.

  **int**[] satunnaisluvut = { 12, 56, 34, 87 };
- Taulukon kokoa ei voi muuttaa jälkeenpäin.
- Kullakin taulukolla on julkinen *length* attribuutti, joka ilmoittaa taulukon koon.
  - Hyödyllinen erityisesti, kun taulukko saadaan parametrina.

# Alkioiden käsittely

- Hakasulkeilla ([]) taulukosta voidaan myös “erottaa” tietty alkio (muuttuja), jonka avulla alkion arvo voidaan lukea tai muuttaa.
- Alkion arvon lukeminen yleisesti:

  **taulukonNimi[indeksiarvo]**
- Alkion arvon muuttaminen yleisesti:

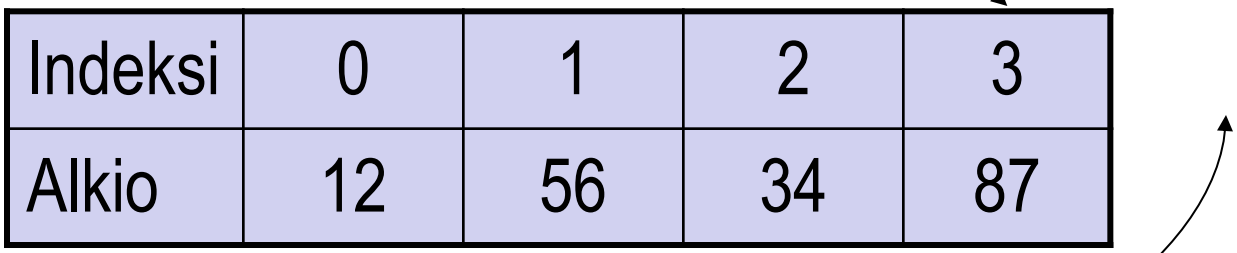
  **taulukonNimi[indeksiarvo] = arvo**;
- Indeksiarvo on **int**-tyyppinen literaali, muuttuja, vakio tai lauseke.

# Alkioiden käsittely

- Indeksointi alkaa nollasta, jolloin 0 ≤ indeksiarvo < taulukon koko.
- Virheellinen indeksin arvo aiheuttaa ajonaikaisen virheen *ArrayIndexOutOfBoundsException*.
- Erityisesti taulukon ylärajan kanssa on syytä olla tarkkana: *n* alkion kokoisen taulukon viimeinen paikka on *n* - 1.
- **For**-silmukka on erityisen kätevä, kun on tarpeen käydä läpi kaikki taulukon indeksiarvot.

# Esimerkki: arvon lukeminen

- System.out.println(satunnaisluvut[3]);  // 87

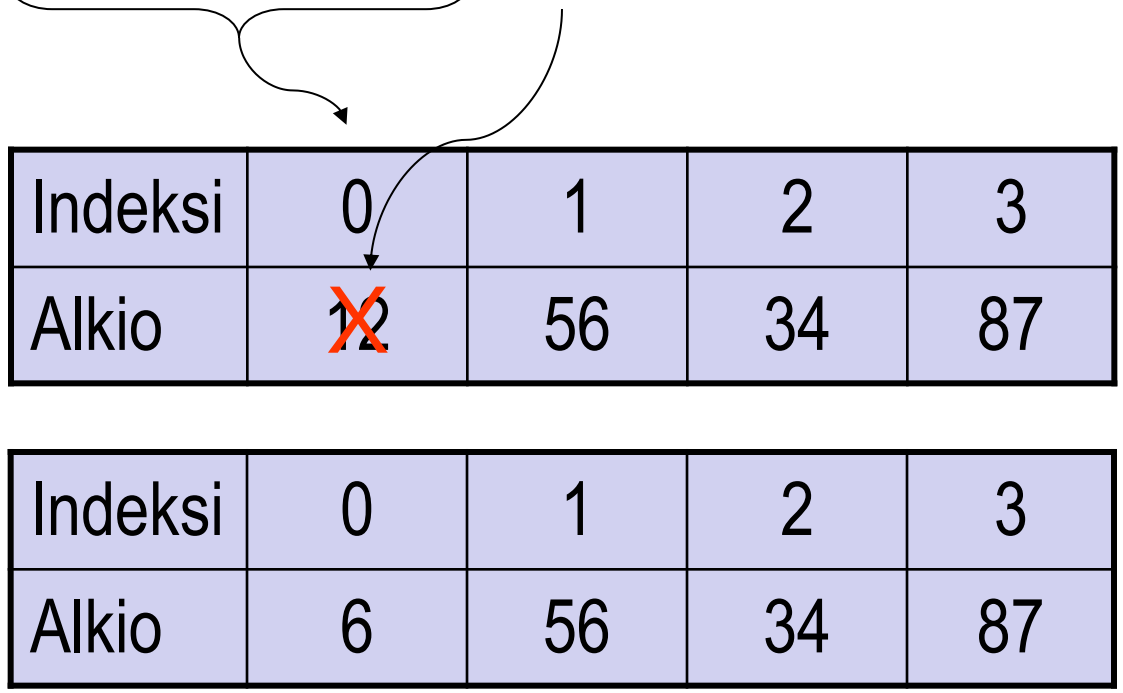


| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Alkio | 12 | 56 | 34 | 87 |

- **int** l = satunnaisluvut[4] + 1;  // Ohjelma kaatuu

# Esimerkki: arvon muuttaminen

- satunnaisluvut[0] = 6;



| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Alkio | ~~12~~ | 56 | 34 | 87 |

| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Alkio | 6 | 56 | 34 | 87 |

- System.out.println(satunnaisluvut[0]);  // 6

# Esimerkki: käsittelyä silmukoilla

```
/* Taulukon käsittelyä silmukoilla: täytetään taulukko satunnaisluvulla
 * ja tulostetaan luvut näytölle.
 */
public class TaulukonKasittelya {
   public static void main(String[] args) {
      // Taulukon koko ja sen suurin luku vakioina.
      final int KOKO = 100; final int MAX = 10;
      // Taulukon esittely ja luominen.
      int[] satunnaisluvut = new int[KOKO];
      // Täytetään taulukko satunnaisluvuilla väliltä [0, MAX].
      for (int i = 0; i < KOKO; i++)
         satunnaisluvut[i] = (int)((MAX + 1) * Math.random());
      // Tulostetaan taulukon alkiot.
      for (int j = 0; j < KOKO; j++)
         System.out.print(satunnaisluvut[j] + " ");
   }
}
```

# Esimerkki: haku taulukosta

```
/* Taulukon käsittelyä silmukoilla:
 * täytetään taulukko satunnaisluvulla ja haetaan
 * käyttäjän antamaa lukua taulukosta.
 */
public class SatunnainenTaulukko1D {
  public static void main(String[] args) {
    // Taulukon suurin luku.
    final int MAX = 100;
    // Taulukon esittely ja luominen.
    int[] satunnaisluvut = new int[10];
    // Täytetään satunnaisluvuilla väliltä [0, MAX].
    for (int i = 0; i < satunnaisluvut.length; i++) {
      satunnaisluvut[i] =
      (int)((MAX + 1) * Math.random());
      System.out.print(satunnaisluvut[i] + " ");
    }
    // Luetaan haettava luku.
    System.out.println("\nAnna haettava luku:");
    int luku = In.readInt();
    // Haetaan.
    boolean hukassa = true;
    int j = 0;
    while (j < satunnaisluvut.length && hukassa) {
      if (satunnaisluvut[j] == luku)
        hukassa = false;
      j++;
    }
    if (!hukassa)
      System.out.print("Löytyi!");
    else
      System.out.print("Ei löytynyt!");
  }
}
```

# Kaksi ulottuvuutta

- Taulukossa voi olla useampia ulottuvuuksia.
- Kahden ulottuvuuden esittämiseen tarvitaan kaksi indeksiä:

toinen ulottuvuus

ensimmäinen ulottuvuus

| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 23 | 44 | 5 | 33 |
| 1 | 3 | 32 | 0 | 11 |
| 2 | 12 | 56 | 34 | 87 |

# Kaksi ulottuvuutta

- Esittely yleisesti: **tyyppi[][] nimi**;
  - Esim. // Esitellään taulukko.
    **int**[][] satunnaisluvut;
- Luominen yleisesti:
  **taulukonNimi = new tyyppi[rivejä][sarakkeita]**;
  - Esim. // Luodaan 12 alkion kokoinen taulukko.
    satunnaisluvut = **new int**[3][4];
- Esittely ja luominen samassa lauseessa:
  **tyyppi[][] taulukonNimi = new tyyppi[rivejä][sarakkeita]**;
  - Esim. // Esitellään ja luodaan 12 alkion kokoinen taulukko.
    **int**[][] satunnaisluvut = **new int**[3][4];

# Kaksi ulottuvuutta

- Alustaminen luomisen yhteydessä:

  **tyyppi[][] taulukonNimi = { rivi1**, … , **rivi*N* }**

  missä **rivi = { arvo1, … , arvo*M* }**

  (myös **tyyppi[][] taulukonNimi =**

  **new tyyppi[][] { rivi1**, … , **rivi*N* }**

  Edellä *N* rivien lukumäärä ja *M* sarakkeiden lukumäärä.
- Esimerkki. **int**[][] satunnaisluvut = { { 23, 44, 5, 33 },
  { 3, 32, 0, 11 }, { 12, 56, 34, 87 } };

# Kaksi ulottuvuutta

- Arvoon viittaaminen yleisesti: **taulukonNimi[rivi][sarake]**
  - Esim.
```
System.out.println(satunnaisluvut[0][0]);    // 23
System.out.println(satunnaisluvut[1][2]);    // 0
System.out.println(satunnaisluvut[2][3]);    // 87
```

| Indeksi | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 23 | 44 | 5 | 33 |
| 1 | 3 | 32 | 0 | 11 |
| 2 | 12 | 56 | 34 | 87 |

- *Length*-attribuutti ilmoittaa rivien lukumäärän.
  - Sarakkeiden lukumäärä saadaan selville tutkimalla jonkin rivin pituus tai rivin pituus.

# Esimerkki: käsittelyä silmukoilla

```
/* Taulukon käsittelyä silmukoilla: täytetään taulukko satunnaisluvulla ja tulostetaan sisältö.
 */
public class SatunnainenTaulukko2D {
   public static void main(String[] args) {
      // Taulukon suurin luku.
      final int MAX = 100;
      // Taulukon esittely ja luominen.
      int[][] satunnaisluvut = new int[3][4];
      // Täytetään taulukko satunnaisluvuilla väliltä [0, MAX].
      for (int rivi = 0; rivi < satunnaisluvut.length; rivi++)
         for (int sarake = 0; sarake < satunnaisluvut[0].length; sarake++)
            satunnaisluvut[rivi][sarake] = (int)((MAX + 1) * Math.random());
      // Tulostetaan taulukon alkiot.
      for (int rivi = 0; rivi < satunnaisluvut.length; rivi++) {
         for (int sarake = 0; sarake < satunnaisluvut[0].length; sarake++)
            System.out.print(satunnaisluvut[rivi][sarake] +  "\t");
         System.out.println();
      }
   }
}
```

# Taulukko parametrina

- Kaiken tyyppiset parametrit ovat operaation lohkon tunnuksia:
  - Parametrit näkyvät vain lohkossaan: luodaan operaatioon tultaessa ja tuhotaan operaatiosta poistuttaessa.
- Alkeistyyppisen parametrin arvoon tehdyt muutokset eivät jää voimaan, koska arvo on kopio.
- Viitetyyppinen parametri on kopio alkuperäisestä viitteestä ja liittyy samaan olioon kuin operaation kutsupaikassa.
- Kopioidun viitteen kautta taulukko-olion sisältöön operaatiossa tehdyt muutokset säilyvät operaatiosta poistumisen jälkeen, koska sisältö ei ole kopio.
  - Operaatiosta ei tarvitse palauttaa viitettä, kun sisältöä muutetaan.

# Taulukko parametrina

- Koska viitetyyppiselle parametrille pitää varata muistia **new**-operaatiolla, ja ihmisen tiedetään olevan erehtyväinen, on viitetyyppisiä parametreja sisältävissä operaatioissa hyvä tarkistaa, että muistia on todella varattu.
- Tähän tehtävään sopii **null**-arvo, jonka Java antaa alkuarvoksi jokaiselle viitteelle. **Null**-arvon voidaan ajatella olevan keskusmuistin ulkopuolella.
- **if** (viitetyyppisenParametrinTunnus != **null**) { … }

# Taulukko parametrina

```
/* Täytetään taulukko t satunnaisluvuilla.
 */
public static void tayta(int[] t) {
   // Satunnaisluvut välillä [0, MAX[.
   final int MAX = 10;
   // Täytetään, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         t[i] = (int)(MAX * Math.random());
   }
}
```

Ilman **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen (*NullPointer-Exception*) aina kun parametrille ei ole varattu muistia. Huomaa ettei paluuarvoa ole.

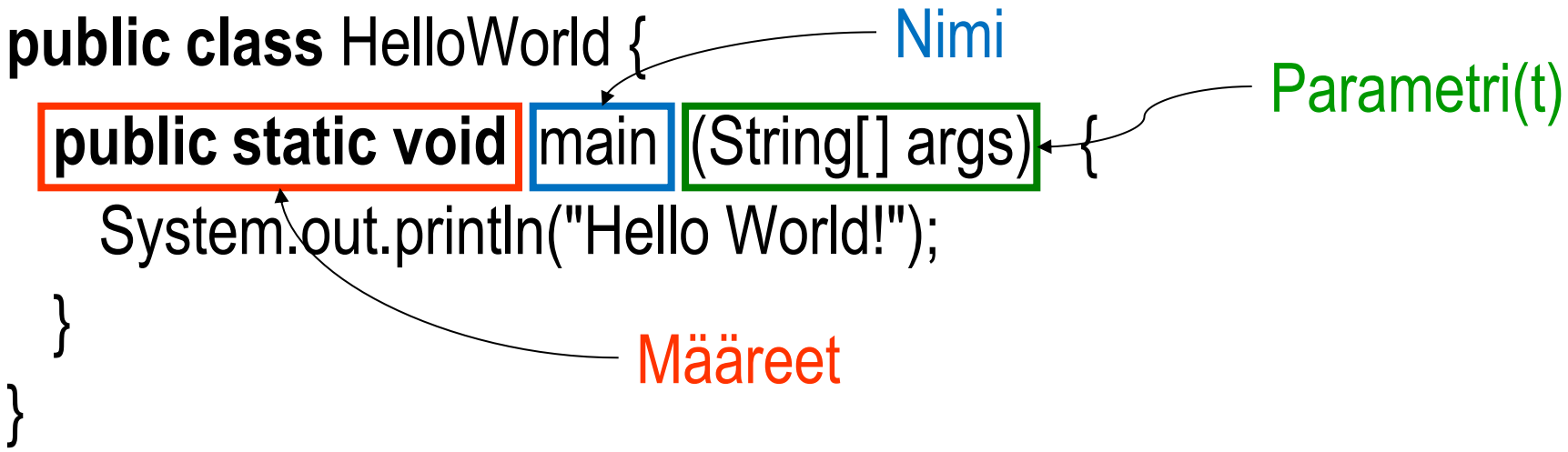
# Taulukko parametrina

```
/* Tulostetaan taulukko t.
 */
public static void tulosta(int[] t) {
   // Tulostetaan, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         System.out.print(t[i] + " ");
      System.out.println();
   }
}
```

# Taulukko parametrina

```
public static void main(String[] args) {
    final int KOKO = 5;
    int[] satunnaisluvut = new int[KOKO];
    // Java alustaa taulukon sisällön automaattisesti.
    tulosta(satunnaisluvut);            // 0 0 0 0 0
    // Täytetään taulukko satunnaisluvuilla.
    tayta(satunnaisluvut);
    // Operaatiossa taulukkoon tehdyt muutokset säilyvät.
    tulosta(satunnaisluvut);            // 8 0 5 7 8
 }
```

# *HelloWorld*-ohjelma




- Parametri *args* on *String*-tyyppisten alkioiden taulukko.

# Taulukko paluuarvona

- Operaatiossa luotu taulukko voidaan palauttaa operaation paluuarvona.
  - Operaatiosta palautetaan taulukko-olioon liittyvä viite.
  - Kutsuvassa operaatiossa taulukko-olioon täytyy kiinnittää uusi viite, jotta oliota voidaan käyttää.
- Joskus yksiulotteista taulukkoa käytetään kahden tai useamman paluuarvon palauttamiseen.
  - Eri tyyppisten arvojen palautus hankalaa.
  - Useampia arvoja voidaan palauttaa joustavammin luokkatyyppisen olion avulla.

# Taulukko paluuarvona

```
/* Luodaan n alkion taulukko, täytetään se satunnaisluvuilla sekä
 * palautetaan viite taulukko-olioon. Paluuarvo on null, jos n < 1.
 */
public static int[] luoJaTayta(int koko) {
   final int MAX = 10;   // Satunnaisluvut välillä [0, MAX[.
   if (koko > 0) {   // Koko on järkevä.
      int[] taulu = new int[koko];   // Luodaan olio ja liitetään siihen viite.
      for (int i = 0; i < taulu.length; i++)   // Täytetään satunnaisluvuilla.
         taulu[i] = (int)(MAX * Math.random());
      return taulu;   // Palautetaan viite taulukko-olioon.
   }
   else   // Virheellinen parametri.
      return null;   // Palautetaan virheen merkkinä null-arvo.
}
```

# Taulukko paluuarvona

```
public static void main(String[] args) {
    // Taulukon alkioiden lukumäärä vakiona.
    final int KOKO = 5;
    // Luodaan taulukko, täytetään se satunnaisluvuilla
    // ja liitetään siihen viite, jotta taulukkoa ei hukata.
    int[] satunnaisluvut = luoJaTayta(KOKO);
    // Operaatiossa taulukkoon sijoitetut arvot säilyvät.
    tulosta(satunnaisluvut);        // 5 6 2 1 4
}
```

# 3. Olio-ohjelmoinista lyhyesti

# Sisällys

- Yleistä.
- Oliot ja luokat.
- Attribuutit.
- Olioiden esittely ja alustus.
- Rakentajat.
- Olion operaation kutsuminen.

# Yleistä

- Olio-ohjelmointia käsitellään hyvin yleisellä tasolla:
  - Tarkastellaan vain yhtä omaa luokkaa.
  - Olio-ohjelmoinnissa keskeiset attribuutit käsitellään lyhyesti.
  - Keskitytään operaatioihin.
- Aihepiiriin palataan tarkemmin kolmannessa periodissa järjestettävällä Olio-ohjelmoinnin perusteet (Oope, 10 op) -kurssilla.

# Oliot ja luokat

- Oliot (object) ja luokat (class) ovat keskeisiä olio-ohjelmoinnin käsitteitä.
- Olio-ohjelmointi on ohjelmointiparadigma, jossa
  - ohjelma kuvataan keskenään kommunikoivina olioina,
  - oliot ajatellaan luokkansa ilmentymiksi (instance) ja
  - luokille voidaan määritellä periytymissuhteita.
- Oliot ja luokat liittyvät siis kiinteästi toisiinsa, mutta ovat kuitenkin eri asia!

# Oliot ja luokat

- Olio-ohjelmoinnissa ohjelman käyttökohteen (ja sen ympäristön) eli sovellusalueen käsitteet (concept) pyritään mallintamaan formaalisti luokkien avulla.
- Luokka vastaa useimmiten sovellusalueen käsitettä hyvin karkealla tasolla.
- Luokkia ei voida yleensä määritellä suoraan, vaan ensin pitää analysoida sovellusalueen kohteita, joiden voidaan ajatella olevan käsitteiden ilmentyminä myös omanlaisiaan olioita.

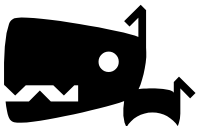
# Oliot ja luokat

- Luokkia rakennettaessa edetään siis usein yksityiskohdista yleiseen määrittelyyn.
- Luokkiin kootaan sovellusalueen olioille yhteisiä
  - tietoja (attribuutteja) ja
  - toiminnallisuutta (operaatioita eli metodeja).
- Oliolla on kaksi roolia:
  - Sovellusalueen käsitteen edustaja.
  - Käsitettä (karkeasti) vastaavan luokan edustaja.

# *Koira*-luokka

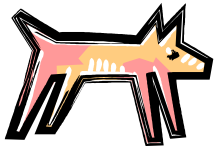
- Oletetaan, että sovellusalueella on koiria. Mallinnetaan ensin luokaksi ja toteutetaan sitten Javalla.

**Teukka**
- kiltti
- sekarotuinen
- "Hau!"
- metsästää rottia



**Hiski**
- rohkea
- bullterrieri
- "RÄYH!"
- repii sohvaa

**Viivi**
- aristokraattinen
- villakoira
- "vuh"
- antaa tassua



**Koira**
- luonne
- rotu
- haukkuu

# *Koira*-luokka

- Kun luokan sisältö on selvillä, se voidaan toteuttaa: Esitellään attribuutit ja toteutetaan metodit.
- Ohessa hyvin alkeellinen *Koira*-luokan toteutus Java-kielellä.

```
public class Koira {
   // Attribuutit.
   private int luonne;
   private String rotu;
   // Metodit.
   public void hauku(String s) {
      System.out.println(s);
   }
}
```

# Attribuutit

- Luokan lohkossa esiteltyjä muuttujia tai vakioita.
- Esitellään lähes samalla tavoin kuin “tavalliset” muuttujat tai vakiot.
- Lisänä määreet. Pyritään yleensä kätkemään luokan ympäristöltä **private**-määreellä.
- Luetaan ja muutetaan aksessorimetodeilla.
- Käytettävä varoen: näkyvät kaikkiin metodeihin ja rikkovat siten modulaarisuusperiaatetta.
  - Älä käytä tällä kurssilla ellei lupaa ole annettu.
  - Vakioituja attribuutteja voi käyttää vapaammin.

```
public class Koira {
  // Attribuutit.
  private int luonne;
  private String rotu;
  // Aksessorit.
  public int luonne() {
    return luonne;
  }
  // Metodit.
  public void hauku(String s) {
    System.out.println(s);
  }
}
```

# Olioiden esittely ja alustus

- Javan oliot eivät ole oliotyyppiä vaan viitetyyppisiä (tarkemmin luokkatyyppisiä) muuttujia.
- Luokkatyyppisiä muuttujia ei voi yleensä ottaen käsitellä kuten alkeistietotyyppisiä muuttujia.
- Viite “osoittaa” jonkin muistipaikan kautta varsinaiseen dataan keskusmuistissa.
- Tästä syystä luokkatyyppisen muuttujan esittely varaa muistia vain viitteen tallentamiseen, ei olion tietojen tallentamiseen.

# Olioiden esittely ja alustus

- Olion tarvitsema muistitilavaraus tehdään alustuksen yhteydessä **new**-operaatiolla, joka palauttaa viitteen ja varaa keskusmuistista muistialueen olion tiedoille.
- Yleisesti:

```
LuokanNimi olionNimi;
olionNimi = new LuokanNimi();
```

tai

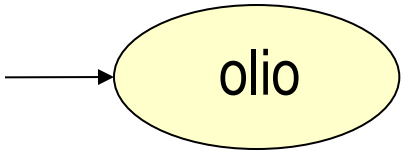
```
LuokanNimi olionNimi = new LuokanNimi();
```

# Olion luominen

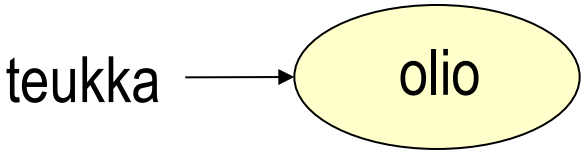
- Koira teukka = **null**;

teukka ⟶ **null**  Olion viitteen esittely ja alustaminen tyhjäksi.

- teukka = **new** Koira();




Lauseke **new** Koira(); luo olion ja palauttaa paluu-arvona olioon liittyvän tunnuksettoman viitteen.




Sijoituksen seurauksena *teukka*-viite viittaa samaan olioon kuin paluuarvona saatu viite.

# Rakentajat

- Attribuuttien arvot alustetaan luokan rakentajassa (constructor), jonka nimi on aina sama kuin luokan nimi. Rakentajalla ei ole tyyppiä.
- Kutsu yleisesti:

  - Oletusrakentaja **LuokanNimi();** tai

  - Parametrillinen rakentaja **LuokanNimi(parametrit);**
- Rakentajaa kutsutaan aina, kun luodaan olio.
- Esimerkiksi: Koira hiski = **new** Koira();

# Rakentajat

- Java luo automaattisesti tyhjän oletusrakentajan.
- Itse kirjoitetulle luokalle voidaan luonnollisesti kirjoittaa myös rakentaja.
- Entä jos on tarvetta sekä parametrittomalle oletusrakentajalle että parametriselle rakentajalle?
- Javassa on kuormittamisena (overloading) tunnettu mekanismi, joka sallii samannimisten metodien esittelyn ja siten myös samannimisen rakentajien esittelyn.

# Olion operaation kutsuminen

- Olion luokan julkisia operaatioita voidaan kutsua olion luokan ulkopuolelta muuttujan nimen ja pistenotaation avulla.
- Esimerkiksi:
```
Koira hiski = new Koira();
hiski.hauku("RÄYH!");
```
- Esimerkiksi:
```
String syote = In.readString();
int merkkienLkm = syote.length();
```
- Esimerkiksi:
```
Scanner lukija = new Scanner(System.in);
String syote = lukija.nextLine();
```
- Esimerkiksi:
```
Random generaattori = new Random();
int arvottu = generaattori.nextInt(10);
```

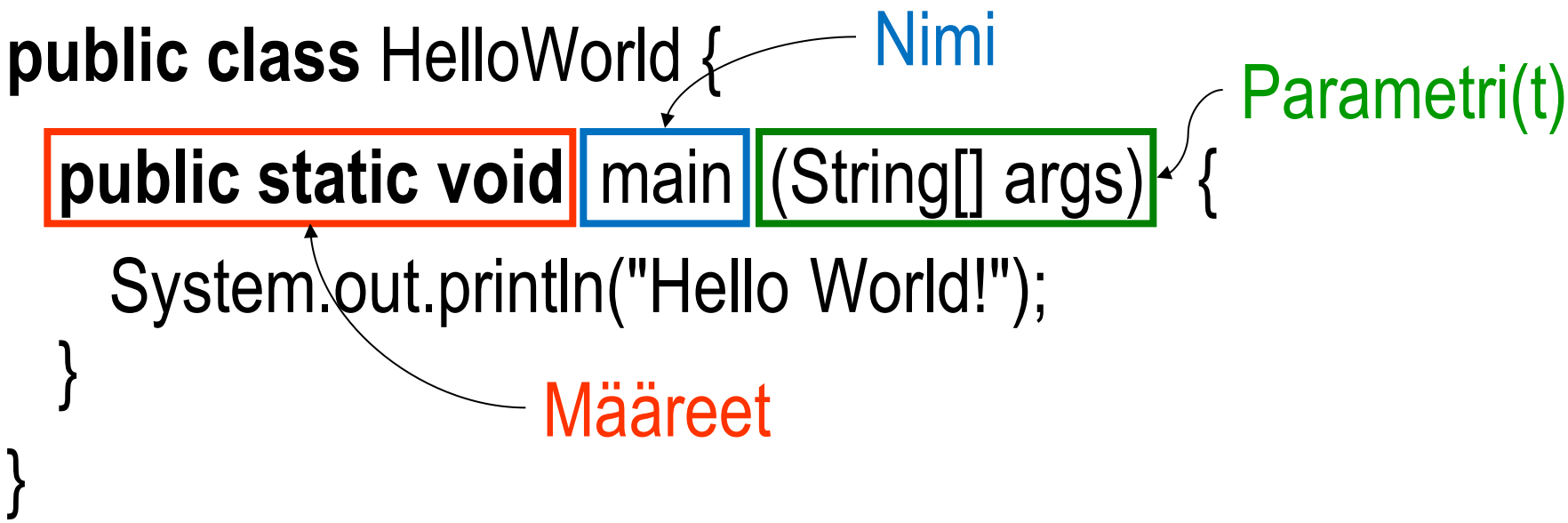
# 4. Komentoriviparametrit

# Komentoriviparametrit

- Ohjelman nimen perään kirjoitettavia merkkijonoja.
- Erotetaan välilyönnein.
- Tuttuja Windows- ja UNIX-komentotulkeista.
- Esimerkkejä komentojen parametreista

| Komento | Ohjelman nimi | Parametri(t) |
|---|---|---|
| copy In.java c:\temp | copy | In.java c:\temp |
| dir *.java | dir | *.java |
| cat -v teksti.txt | cat | -v teksti.txt |

# HelloWorld: main-operaatio




- Parametri *args* on String-tyyppisten alkioiden taulukko, jonka avulla komentoriviparametri(t) välitetään Java-ohjelmalle.

# Komentoriviparametrit

- Komentorivi: java SayHello World
- ```
  public class SayHello {
      public static void main(String[] args) {
          // Tulostetaan, jos (vähintään) yksi parametri.
          if (args.length > 0)
              System.out.println("Hello " + args[0]);
      }
  }
  ```
- Näyttö: Hello World
  ```

# Komentoriviparametrit

```
public class KRPt {
   public static void main(String[] args) {
      System.out.print("Komentoriviparametrejä ");
      System.out.print(args.length);
      System.out.println(" kpl.");
      // Tulostetaan komentoriviparametrit näytölle.
      for (int i = 0; i < args.length; i++)
         System.out.println(args[i]);
   }
}
```

# Komentoriviparametrit

- Komentorivi:

  java KRPt Tämä on testi

- Näyttö:

  Komentoriviparametrejä 3 kpl.

  Tämä

  on

  testi

# 5. Poikkeukset

# Yleistä

- Virheeseen varautuminen tarkoittaa sitä, että ohjelmoija huomioi koodia kirjoittaessaan ajonaikaisen virheen mahdollisuuden.
- Perinteiset tavat käsitellä virheitä:
  - Virhe tunnistetaan ja vältetään, mutta siihen ei reagoida muuten.
  - Virhe tunnistetaan ja vältetään ja sen tapahtumisesta ilmoitetaan operaation paluuarvon avulla.

# Yleistä

- Edellä mainitut keinot antavat riittävät keinot virheiden käsittelyyn, mutta niissä on omat ongelmansa:
  - Paluuarvot on helppo jättää huomiotta.
  - Paluuarvoa ei voi aina käyttää virheen ilmaisemiseen.
  - Virheidenkäsittelyyn käytetty ohjelmakoodimäärä saattaa olla yllättävän suuri ja ohjelman luettavuus saattaa kärsiä.
- Java tarjoaa poikkeuksina (exceptions) tunnetun mekanismin ajonaikaisten virheidenkäsittelyyn.

# Poikkeusten käsittely

- Mikäli jossakin operaatiossa tapahtuu ajonaikaisen virheen aiheuttama poikkeus, on sen käsittelyyn kaksi mahdollisuutta:
  - poikkeus käsitellään paikallisesti pois päiväjärjestyksestä tai
  - poikkeus heitetään (throw) kutsuvalle operaatiolle.
- Ohjelman suoritus pysähtyy, mikäli poikkeusta ei käsitellä viimeistään pääohjelmassa.
- Poikkeusta ei siis voida jättää koskaan huomiotta.
- Tällä kurssilla tutustutaan vain paikalliseen käsittelyyn, jotta tiedostojenkäsittelty olisi mahdollista.

# Poikkeusten käsittely

- Poikkeuksen käsittelyyn tarvitaan **try-catch**-lause:

```
try {
    // Mahdollisesti poikkeuksen tuottavat lauseet.
}
catch (Tyyppi1 tunnus1) {
     // Tässä käsitellään tyypin1 virheet.
}

…
catch (Tyyppi N tunnus N) {
    // Tässä käsitellään tyypin N virheet.
}
```

# Poikkeusten käsittely

- Poikkeusten tyypit ovat Javan luokkia – poikkeukset ovat siis olioita.
- Joitakin poikkeuksia:
  - *NullPointerException*: viitteelle ei ole varattu muistia.
  - *ArrayIndexOutOfBoundsException*: viitataan virheelliseen paikkaan taulukossa.
  - *OutOfMemoryError*: ei riittävästi muistia olion luomiseksi.
  - *Exception*: Mikä tahansa poikkeus.

# Poikkeusten käsittelyä paikallisesti

```
/* Täytetään taulukko t satunnaisilla luvuilla.
 */
public static void tayta(int[] t) {
   try {
      // Satunnaisluvut välillä [0, MAX[.
      final int MAX = 10;
      // Mahdollista virhettä ei vältetä vaan se käsitellään.
      for (int i = 0; i < t.length; i++) {
         t[i] = (int)(MAX * Math.random());
      }
   }
   catch (NullPointerException e) {
      System.out.println("Taulukolle pitää varata muistia!");
   }
}
```

# 6. Tiedostot

# Sisältö

- Johdanto.
- Tiedostojen lukeminen.
- Tiedostojen kirjoittaminen.

# Johdanto

- Tiedostoja on käsitelty uudelleenohjattujen standardisyöte- ja tulostusvirtojen avulla.
- Tiedostoja voidaan käyttää myös ohjelmasta käsin.
- Tutustutaan tekstitiedostojen lukemiseen ja kirjoittamiseen *java.io*-pakkauksen luokkien avulla.
  - Javan versiosta 1.5 alkaen käytettävissä myös *java.util*-pakkauksen *Scanner*- ja *Formatter*-luokat.
  - **import** java.io.*; (tai **import** java.util.*;).
- Lukeminen ja kirjoittaminen tapahtuu vaiheittain: avataan, luetaan tai kirjoitetaan, suljetaan.

# Avaaminen lukemista varten

- Tiedoston avaaminen tarkoittaa yleisesti syötevirran liittämistä tiedostosta ohjelmaan.
- Javassa syötevirta avataan esimerkiksi luomalla *FileInputStream*-luokan olio. Yleisesti:

  FileInputStream olionNimi

  = **new** FileInputStream(tiedostonNimi);
- Esimerkiksi: FileInputStream syotevirranNimi

  = **new** FileInputStream("in.txt");

# Lukeminen

- Syötevirrasta lukemiseen tarvitaan olio *InputStreamReader*-luokasta.
- Esimerkiksi: InputStreamReader lukijanNimi =

  **new** InputStreamReader(syotevirranNimi);
- Usein on kätevämpää käyttää hienostuneempaa lukijaa. Tällaisen tarjoaa *BufferedReader*-luokka.
- Esimerkiksi: BufferedReader puskuroituLukija =

  **new** BufferedReader(lukijanNimi);

# Lukeminen

- Kun syötevirtaan on saatu liitettyä *BufferedReader*-lukija, voidaan tekstitiedostoa lukea riveittäin *readLine*-operaation avulla.
- *Ready*-operaatio kertoo voidaanko lukea uusi rivi.
- Esimerkiksi:

```
while (puskuroituLukija.ready()) {
    String rivi = puskuroituLukija.readLine();
    System.out.println(rivi);
}
```

# Rivien käsittely

- Usein yhdellä rivillä on useampia jollakin merkillä (esimerkiksi pilkku tai välilyönti) erotettuja tietoja, jotka voivat vieläpä olla eri tyyppisiä.
- *String*-luokan *split*-operaatio on tässä tilanteessa hyödyllinen – se pilkkoo rivi osiinsa.
- Esimerkiksi:

```
String rivi = "this is a test";
String[] osat = rivi.split("[ ]");
for (int i = 0; i < osat.length; i++)
   System.out.println(osat[i]);
```

# Sulkeminen lukemisen jälkeen

- Tiedoston lukemisen jälkeen siihen liitetty syötevirta suljetaan *close*-operaatiolla.
- Esimerkiksi: puskuroituLukija.close();
- Tiedoston avaamisen, lukemisen ja/tai sulkemisen yhteydessä voi tapahtua poikkeus.
- Tämän vuoksi Java pakottaa sulkemaan useimmat tiedostonkäsittelyyn liittyvät lauseet **try-catch**-lauseen sisään.

# Esimerkki

```
import java.io.*;
public class Lukeminen1 {
  public static void main(String[] args) {
    String TIEDNIMI = "in.txt";
    try {
      FileInputStream syotevirta =
      new FileInputStream(TIEDNIMI);
      InputStreamReader lukija =
      new InputStreamReader(syotevirta);
      BufferedReader puskuroituLukija =
      new BufferedReader(lukija);
      while (puskuroituLukija.ready()) {
        String rivi = puskuroituLukija.readLine();
        System.out.println(rivi);
      }
      puskuroituLukija.close();
    }
    catch (FileNotFoundException e) {
      System.out.print("Tiedosto hukassa!");
    }
    catch (Exception e) {
      System.out.print("Lukuvirhe!");
    }
  }
}
```

# Esimerkki

```
import java.io.*; import java.util.*;
public class Lukeminen2 {
  public static void main(String[] args) {
    String TIEDNIMI = "in.txt";
    try {
      // Luodaan tiedosto-olio.
      File tiedosto = new File(TIEDNIMI);
      // Luodaan lukija.
      Scanner lukija = new Scanner(tiedosto);
      // Luetaan ja tulostetaan rivit.
      while (lukija.hasNextLine()) {
        String rivi = lukija.nextLine();
        System.out.println(rivi);
      }
      // Suljetaan lukija.
      lukija.close();
    }
    catch (FileNotFoundException e) {
      System.out.print("Tiedosto hukassa!");
    }
    catch (Exception e) {
      System.out.print("Lukuvirhe!");
    }
  }
}
```

# Avaaminen kirjoittamista varten

1) Luodaan tiedosto-olio *File*-luokasta:

```
File tiedostoOlionNimi =
    new File(tiedostonNimi);
```

2) Liitetään tulostusvirta ohjelmasta tiedostoon:

```
FileOutputStream virtaOlionNimi =
    new FileOutputStream(tiedostoOlionNimi);
```

# Kirjoittaminen

- Tulostusvirtaan kirjoittamiseen tarvitaan kirjoittajaolio *PrintWriter*-luokasta. Yleisesti:

  *PrintWriter* kirjoittajaOlionNimi =

  **new** *PrintWriter*(virtaOlionNimi, **true**);

- Kirjoittaminen tapahtuu *PrintWriter*-luokan *print*- ja *println*-operaatioiden avulla.
- Nämä operaatiot on kuormitettu kuten *System.out*-attribuutin kautta kutsuttavat operaatiot, jotka tulostavat esimerkiksi lukuja ja merkkijonoja.

# Sulkeminen kirjoittamisen jälkeen

- Tiedoston kirjoittamisen jälkeen siihen liitetty syötevirta suljetaan kirjoittajan *close*-operaatiolla.
- Myös kirjoittamisen yhteydessä voi tapahtua poikkeus: tarvitaan poikkeuksen käsittelyä **try-catch**-lauseen avulla.

# Tiedostot: esimerkki

```
import java.io.*;
public class Kirjoittaminen {
  public static void main(String [] args) {
    final String TIEDNIMI = "out.txt";
    final int HEIPPALKM = 10;
    try {
      // Luodaan tiedosto-olio
      File tiedosto =
      new File(TIEDNIMI);
      // Luodaan tulostusvirta ja
      // liitetään se tiedostoon
      FileOutputStream tulostusvirta =
      new FileOutputStream(tiedosto);
      // Luodaan virtaan kirjoittaja
      PrintWriter kirjoittaja =
      new PrintWriter(tulostusvirta, true);
      // Kirjoitetaan tiedostoon
      for (int i = 0; i < HEIPPALKM; i++)
        kirjoittaja.println("Heippa " + i);
      // Suljetaan tiedosto
      kirjoittaja.close();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

# 7. Hyvä ohjelmointitapa.

# Yleistä

- Jaa pitkä koodi osiin.
- Käytä attribuutteja säästeliäästi.
- Kommentoi operaatiot ja attribuutit.
- Varaudu operaatioissa virheisiin.
- Tunne ohjelmointikieli.

# Jaa pitkä koodi osiin

- Satoja tai tuhansia rivejä sisältävää ohjelmaa ei ole järkevää kirjoittaa yhdeksi “pötköksi”.
- On hyvä jakaa ohjelma helposti ymmärrettäviksi ja hallittavaksi osiksi (modulaarinen ohjelmointi).
- Operaatiot ovat pääasiallinen ositusmenetelmä.
  - Myös luokat ja pakkaukset ovat keinoja hallita isoja kokonaisuuksia.
- Osat ideaalisesti ympäristöstään riippumattomia.
  - Operaatioiden tulisi vuorovaikuttaa muiden operaatioiden kanssa vain parametrien ja paluuarvojen avulla.

# Käytä attribuutteja säästeliäästi

- Olio-ohjelmoinnissa tärkeät attribuutit ovat ristiriidassa modulaarisuusperiaatteen kanssa, koska ne ovat “globaaleina” muuttujina käytettävissä kaikissa luokan operaatioissa.
  - Attribuutteja on käytettävä varoen.
  - Erityisesti toisessa harjoitustyössä ei saa olla (turhia) attribuutteja.
  - Vakiomuotoisia attribuutteja saa käyttää vapaammin, koska niiden arvoja ei voi muuttaa operaatioissa.

# Kommentoi operaatiot ja attribuutit

- Operaatioihin liitetään yleisluontoiset kommentit operaation tarkoituksesta, parametreista ja paluuarvoista:

```
/* Tulostetaan kaksiulotteinen taulukko t.
 */
public static void tulosta(int[][] t) { … }
```

  - Lyhyitä parametritunnuksia käytettäessä parametrien tarkoitus on ehdottomasti kerrottava kommentissa.
  - Operaatioiden nimet ovat muuttujien nimiä pitempiä ja alkavat usein käskymuotoisella verbillä.
- Attribuutit kommentoidaan muuttujien tapaan:

```
// Ohjelman lopettava komento.
private static final String KOMENTO_LOPETA = "quit";
```

# Varaudu operaatioissa virheisiin

- Maailma ei ole täydellinen – ohjelmasi ei esimerkiksi saa syötteitä juuri siten kuin toivoisit.
- Usein muutama yksinkertainen tarkistus tekee ohjelmasta huomattavasti vakaamman ja samalla miellyttävämmän käyttää.
- Hyvässä ohjelmassa on varauduttu virhetilanteisiin ja ohjelman toimintaa näissä tilanteissa on vieläpä testattu.
- Erityisesti operaatioiden viitetyyppiset parametrit on syytä tarkistaa.
  - Operaation kutsuminen **null**-arvoisen viitteen kautta aiheuttaa ohjelman pysäyttävän ajonaikaisen virheen.

# Varaudu operaatioissa virheisiin

```
/* Täytetään taulukko t satunnaisluvuilla.
 */
public static void tayta(int[] t) {
   // Satunnaisluvut välillä [0, MAX[.
   final int MAX = 10;
   // Täytetään, jos on varattu muistia.
   if (t != null) {
      for (int i = 0; i < t.length; i++)
         t[i] = (int)(MAX * Math.random());
   }
}
```

Ilman **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen (*NullPointer-Exception*) aina, kun parametrille ei ole varattu muistia.

# Varaudu operaatioissa virheisiin

```
// Tutkitaan ovatko merkkijonon m ensimmäinen
// ja viimeinen merkki sama merkki.
public static boolean ekaJaVikaSamat(String m) {
   // Arvataan, että merkit eivät ole samoja.
   boolean ovatSamat = false;
   // Edetään, jos viitteeseen liittyy olio
   // ja merkkijonossa on vähintään yksi merkki.
   if ((m != null) && (m.length() > 0))
      // Käännetään lippu, jos merkit ovat samat.
      if (m.charAt(0) == m.charAt(m.length() - 1))
         ovatSamat = true;
   // Palautetaan tulos.
   return ovatSamat;
}
```

Ilman ulompaa **if**-lausetta ohjelman suoritus keskeytyy ajonaikaiseen virheeseen, kun parametrille ei ole varattu muistia (*NullPointer-Exception*) tai kun merkkijonon pituus on nolla (*StringIndexOutOf-BoundsException*).

# 8. Rekursio

# Johdanto

- Rekursiivinen operaatio kutsuu itseään.
- Rekursio etenee askelittain “alaspäin” kunnes kohdataan alkeistapaus (base case), joka voidaan ratkaista.
- Tämän jälkeen palataan “ylöspäin” yhdistäen kussakin askeleessa tehtyjen rekursiivisten kutsujen tulokset.
- Rekursion avulla voidaan ratkaista vaikeitakin ongelmia yksinkertaisesti.
  - Käytetään esimerkiksi lajittelualgoritmeissa.
- Rekursio on haastava ohjelmointitekniikka.
  - Ikuinen rekursio mahdollinen.
- Rekursio on laskennallisesti tehoton menetelmä.
- Luonteva ratkaisu vain osaan ongelmista.

# Esimerkki

- Tarkastellaan esimerkkinä luvun $n$ kertoman $n!$ ($n \geq 0$) laskemista sekä iteratiivisesti että rekursiivisesti .
- Määritellään erikoistapaukset: $0! = 1$ ja $1! = 1$.
- Iteratiivinen ratkaisu:
  - $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$
  - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
- Rekursiivinen ratkaisu:
  - $n! = n \cdot (n - 1)!$
  - $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1!$

  $= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

# Esimerkki

```
/* Lasketaan luvun n kertoma iteratiivisesti.
 */
public static int laskeKertoma(int n) {
   // 0! = 1 ja 1! = 1.
   int kertoma = 1;
   // Silmukoidaan, jos n > 1.
   while (n > 1) {
      // Päivitetään tuloa ja tuotetaan seuraava kerrottava.
      kertoma = kertoma * n;
      n = n - 1;
   }
   // Palautetaan tulos.
   return kertoma;
}
```

# Esimerkki

```
/* Lasketaan luvun n kertoma rekursiivisesti.
 */
public static int laskeKertoma(int n) {
   // Käytetään perustapauksena
   // luvun 1 kertomaa.
   if (n <= 1)
      return 1;
   // Palautetaan rekursiivisen kutsun tulos
   // luvulla n kerrottuna.
   else
      return n * laskeKertoma(n - 1);
}
```