11

Ohjausrakenteet

Käskyt, joiden avulla voit ohjata ohjelman suoritusta C#-sovelluksessa, voidaan jakaa kolmeen pääryhmään: valintakäskyt, toistokäskyt ja hyppykäskyt. Kussakin tapauksessa suoritetaan looginen lauseke ja sen tulosta käytetään ohjelman suorituksen ohjaamiseen. Tässä luvussa opit käyttämään näitä eri tyyppisiä ohjausrakenteita ohjelman suorituksen ohjaamiseen

Valintakäskyt

Käytät valintakäskyjä määrätäksesi, mikä koodin osa tulee suorittaa ja milloin se suoritetaan. C#:n valikoimiin kuuluu kaksi valintakäskyä: *switch*, jota käytetään koodin suorittamiseen arvon perusteella ja *if*, jota käytetään suoritettavan koodin valintaan vertailuehdon mukaan. Yleisimmin käytetty näistä valintakäskyistä on *if*.

if-käsky

if-käsky suorittaa yhden tai useamman käskyn, jos tutkittavan lausekkeen arvo on *true*. *if*-käskyn syntaksi on seuraavassa. Hakasulkeet osoittavat valinnaisen *else*-osan käyttöä (tästä kerrotaan kohta lisää):

```
if (lauseke)
    käsky1
[else
    käsky2]
```

lauseke on mikä tahansa testi, joka tuottaa loogisen arvon. Jos lausekkeen tulos on *true* (tosi), ohjelman suoritus siirtyy *käskyyn1*. Jos tulos on *false* (epätosi) ja *else*-osa on

kirjoitettu, ohjelman suoritus siirtyy *käskyyn2*. Huomaa, että *käsky1* ja *käsky2* voi koostua yhdestä käskystä, joka päättyy puolipisteeseen (yksinkertainen käsky) tai useasta aaltosulkujen välissä olevasta käskystä (yhdistetty käsky). Seuraavassa yhdistetty käsky, joka suoritetaan, kun *lauseke1* on tosi.

```
if (lauseke1)
{
    käsky1
    käsky1
}
```

Seuraavassa esimerkissä sovellus pyytää käyttäjää syöttämään luvun väliltä 1..10. Sen jälkeen muodostetaan satunnaisluku ja käyttäjälle kerrotaan, oliko hänen syöttämänsä luku sama kuin satunnaisluku. Tämä yksinkertainen esimerkki havainnollistaa *if*-käskyn käyttöä C#-ohjelmassa:

```
using System;

class IfTest1App
{
    const int MAX = 10;

    public static void Main()
    {
        Console.Write("Guess a number between 1 and {0}...", MAX);
        string inputString = Console.ReadLine();

        int userGuess = inputString.ToInt32();

        Random rnd = new Random();
        double correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write("The correct number was {0} and you guessed {1}...",
                      correctNumber, userGuess);
        if (userGuess == correctNumber) // They got it right!
        {
            Console.WriteLine("Congratulations!");
        }
        else // Väärä vastaus!
        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

Useita *else*-ehtoja

if-käskyn *else*-ehdon avulla voit määritellä vaihtoehtoisia siirtymiskohtia toiminnolle, jos *if*-käskyn lausekkeen arvo on *false*. Numeronarvausesimerkissä sovellus suoritti yksinkertaisen vertailun käyttäjän arvaaman numeron ja satunnaisluvun kesken. Tällöin on olemassa vain kaksi vaihtoehtoa: käyttäjä on oikeassa tai väärässä. Voit myös yhdistää *if* ja *else* -osat käsittelemään tilanteita, joissa haluat tutkia usempaa kuin kahta vaihtoehtoa. Alla olevassa esimerkissä kysyn käyttäjältä, mitä kieltä hän on par'aikaa käyttämässä (mukaan lukien C#). Olen antanut mahdollisuuden valita kolmesta kielestä, joten *if*-käskyn pitää selviytyä neljästä mahdollisesta vastauksesta: kolmesta eri kielestä ja tapauksesta, jossa käyttäjä valitsee tuntemattoman kielen. Tässä yksi tapa ohjelmoida tämä *if/else*-käskyllä:

```
using System;

class IfTest2App
{
    const string CPlusPlus = “C++";
    const string VisualBasic = “Visual Basic";
    const string Java = “Java";

    public static void Main()
    {
        Console.Write(“What is your current language of choice “ +
                      “(excluding C#)?”);
        string inputString = Console.ReadLine();

        if (0 == String.Compare(inputString, CPlusPlus, true))
        {
            Console.WriteLine(“\nYou’ll have no problem picking “ +
                              “up C# !”);
        }
        else if (0 == String.Compare(inputString, VisualBasic, true))
        {
            Console.WriteLine(“\nYou’ll find lots of cool VB features “ +
                              “in C# !”);
        }
        else if (0 == String.Compare(inputString, Java, true))
        {
            Console.WriteLine(“\nYou’ll have an easier time “ +
                              “picking up C# <G> !!”);
        }
        else
        {
            Console.WriteLine(“\nSorry - doesn’t compute.”);
        }
    }
}
```

Huomaa ==-operaattorin käyttö vertailtaessa arvoa 0 *String.Compare*-metodin paluuarvoon. Syy on se, että *String.Compare* palauttaa arvon -1, jos ensimmäinen merkkijono on pienempi (aakkosjärjestyksessä, ei pituudessa) kuin toinen, arvon 1 jos ensimmäinen merkkijono on suurempi kuin toinen ja arvon 0, jos merkkijonot ovat samat. Tämä osoittaa muutamia mielenkiintoisia yksityiskohtia siitä, miten C# suorittaa *if*-käskyn. Näistä puhutaan tarkemmin seuraavassa kappaleessa.

Miten C# toteuttaa *if*-käskyn

Yksi seikka if-käskyssä, joka saa uudet C#-ohjelmoijat varuilleen, on se, että tutkittavien lausekkeiden tulee palauttaa Boolean-arvo. Tämä eroaa C++:n tapaisista kielistä, joissa voit tutkia if-käskyssä eroaako minkä tahansa muuttujan arvo nollasta. Seuraava esimerkki näyttää useita yleisiä virheitä, joita C++-ohjelmoijat tekevät, kun yrittävät käyttää C#:n if-käskyä ensimmäistä kertaa:

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class InvalidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo) // VIRHE: yritys muuntaa int bool-tyypiksi.
        {
        }

        TestClass t = GetTestClass();
        if (t) // VIRHE: yritys muuntaa TestClass bool-tyypiksi.
        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i) // VIRHE: yritys  muuntaa ITest bool-tyypiksi.
            {
                // ITest-metodit
```

```
            }
        }
    }
}
```

Jos yrität kääntää tätä koodia, saat C#-kääntäjältä seuraavat virheilmitukset:

```
invalidIf.cs(22,7): error CS0029: Cannot implicitly convert type ‘int’ to ‘bool’
invalidIf.cs(27,7): error CS0029: Cannot implicitly convert type ‘TestClass’ to ‘bool’
invalidIf.cs(31,14): warning CS0183:
The given expression is always of the provided (‘ITest’) type
invalidIf.cs(32,8): error CS0029: Cannot implicitly convert type ‘ITest’ to ‘bool’
```

Kuten näet, kääntäjä parkaisee kolme kertaa vastauksena kolmeen yritykseen käyttää *if*-käskyä operaatiossa, joka ei palauta Boolean-arvoa. Syy tähän on se, että C#:n kehittäjät halusivat auttaa sinua välttämään virhealttiin koodin kirjoittamista ja uskoivat, että kääntäjän tulee pakottaa ohjelmoija *if*-käskyn alkuperäiseen käyttöön eli ohjaamaan ohjelman suoritusta loogisen lausekkeen tuloksen perusteella. Ylläoleva esimerkki on kirjoitettu alla uudelleen ilman virheitä. Kukin edellisessä ohjelmassa virheen aiheuttanut rivi on muokattu sisältämään Boolean-arvon palauttavan lausekkeen tyynnyttämään kääntäjää.

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class ValidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo > 0)
        {
        }
```

(jatkuu)

```
            TestClass t = GetTestClass();
            if (t != null)
            {
                Console.WriteLine("{0}", t);

                ITest i = t as ITest;
                if (i != null)
                {
                    // ITest methods.
                }
            }
        }
}
```

switch-käsky

Käyttämällä *switch*-käskyä voit määritellä lausekkeen, joka palauttaa kokonaisluvun ja yhden tai useamman koodipätkän, joista jokin suoritetaan lausekkeen arvon perusteella. Se on sama asia kuin usean *if/else*-käskyn käyttäminen, mutta vaikka voitkin määritellä useita (mahdollisesti erillisiä) ehtolauseita useassa *if/else*-käskyssä, *switch*-käsky on kätevämpi, koska se muodostuu vain yhdestä ehtolauseesta, jonka jälkeen tulevat kaikki ne tulokset, jotka koodisi tulee käsitellä. Tässä käskyn syntaksi:

switch (*valintalauseke*)
{
    case *vakiolauseke*:
        käsky
        hyppykäsky
    §
    case *vakiolausekeN*:
        käskyN
    [*default*]
}

Tässä on kaksi mielessä pidettävää asiaa. Ensinnäkin valintalausekkeen tulee olla jokin seuraavista tyypeistä (tai automaattisesti muunnettavissa sellaiseksi): *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char* tai *string* (tai *enum*, joka perustuu johonkin edellämainittuun tyyppiin). Toiseksi, sinun tulee kirjoittaa hyppykäsky kullekin case-tapaukselle, jos se ei ole rakenteen alimmainen, mukaan lukien default-osa. Koska toiminta on erilainen kuin useimmissa muissa kielissä, käyn sen yksityiskohtaisesti läpi myöhemmin tässä luvussa otsikon "Ei läpijuoksua switch-käskyssä" alla.

Käsitteellisesti *switch*-käsky toimii samoin kuin *if*-käsky. Ensin ratkaistaan *valintalausekkeen* arvo ja sen tulosta verrataan jokaiseen *vakiolausekkeeseen*, jotka on määritelty

eri *case*-kohdissa. Kun vastaava arvo löytyy, ohjelman suoritus siirtyy sen *case*-kohdan ensimmäiselle koodiriville.

Sen lisäksi, että voit määritellä erilaisia *case*-kohtia, voit *switch*-käskyssä määritellä myös *default*-kohdan (oletus). Tämä toimii kuin *else*-ehto *if*-käskyssä. Huomaa, että yhdessä *switch*-käskyssä voi olla vain yksi *default*-kohta. Jos valintalausekkeen mukaista *case*-kohtaa ei löydy, eikä *default*-kohtaa ole määritelty, siirtyy ohjelman suoritus *switch*-käskyn päättävän aaltosulun jälkeiselle riville. Tässä esimerkki, *Payment*-luokka käyttää *switch*-käskyä määritellessään, mikä maksutapa on valittu:

```
using System;

enum Tenders : int
{
    Cash = 1,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
```

(jatkuu)

```
        {
            case (int)Tenders.Cash:
                Console.WriteLine("\nCash - Accepted");
                break;

            case (int)Tenders.Visa:
                Console.WriteLine("\nVisa - Accepted");
                break;

            case (int)Tenders.MasterCard:
                Console.WriteLine("\nMastercard - Accepted");
                break;

            case (int)Tenders.AmericanExpress:
                Console.WriteLine("\nAmerican Express - Accepted");
                break;

            default:
                Console.WriteLine("\nSorry - Invalid tender");
                break;
        }
    }
}

class SwitchApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}
```

Sovelluksen suorittaminen antaa seuraavan tulosteen, koska instantioimme *Payment* -luokan välittämällä sille arvon *Tenders.Visa*:

```
Visa - Accepted.
```

Combining Case Labels

Payment-esimerkissä käytimme eri *case*-kohtaa jokaista mahdollista *Payment.tender*-kentän arvoa kohden. Entä, jos haluamme yhdistää toimintoja? Haluat esimerkiksi näyttää luottokortin tunnistusikkunan jokaisella luottokorttityypillä mutta et tietenkään käteisellä maksettaessa. Tällöin voit sijoittaa kolme case-kohtaa alekkain näin:

```
using System;
```

```
enum Tenders : int
{
    Cash = 1,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.Cash:
                Console.WriteLine
                          (“\nCash - Everyone’s favorite tender.”);
                break;

            case (int)Tenders.Visa:
            case (int)Tenders.MasterCard:
            case (int)Tenders.AmericanExpress:
                Console.WriteLine
(“\nDisplay Credit Card Authorization Dialog.”);
                break;

            default:
                Console.WriteLine(“\nSorry - Invalid tender.”);
```

(jatkuu)

```
                break;
            }
        }
    }

    class CombiningCaseLabelsApp
    {

        public static void Main()
        {
            Payment payment = new Payment(Tenders.MasterCard);
            payment.ProcessPayment();
        }
    }
```

Jos instantioit *Payment*-luokan arvolla *Tenders.Visa*, *Tenders.MasterCard* tai *Tenders.AmericanExpress*, saat seuraavan tulosteen:

```
Display Credit Card Authorization Dialog.
```

Ei läpijuoksua *switch*-käskyssä

Koko C#-kielen kehityksen ajan sen suunnittelijat ovat tietoisesti soveltaneen erilaisia testejä, kun ovat päättäneet, mitä ominaisuuksia kieleen otetaan mukaan. Läpijuoksu (fall-through) on yksi, joka ei läpäissyt testiä. Yleensä C++:ssa *case*-osa suoritetaan, kun sen vakiolauseke täsmää valintalausekkeen arvoon. Sen jälkeen *switch*-käsky katkaistaan *break*-käskyllä. Läpijuoksussa seuraava *switch*-käskyn *case*-osa suoritetaan, jos *break*-käsky puuttuu.

Vaikka ominaisuutta ei tueta C#:ssa, kerron, että läpijuoksua käytetään yleensä tilanteissa, joissa sinulla on kaksi *case*-osaa ja toinen sisältää toiminnot, jotka tulee tehdä molemmissa tapauksissa. Tein esimerkiksi C++:lla tietokantaeditorin, jonka avulla loppukäyttäjä pystyi luomaan taulunsa ja kenttänsä graafisesti. Kukin taulu näytettiin puurakenteessa Windows Explorerin tapaisessa käyttöliittymässä. Jos käyttäjä napautti puuta kakkospainikkeella, halusin näyttää paikallisvalikon, jossa on sellaisia vaihtoehtoja kuin Tulosta kaikki taulut ja Luo uusi taulu. Jos käyttäjä napautti kakkospainikkeella määrättyä taulua, halusin näyttää valikossa sen taulun paikallisvalikon. Tuossa valikossa piti olla kuitenkin myös kaikki puurakenteen vaihtoehdot. Koodini oli rakenteellisesti tällainen:

```
// Luodaan C++:ssa dynaamisesti valikko.
switch(itemSelected)
{
    case TABLE:
    // Lisätään kyseisen taulun valikkokohdat.
    // break jätetty pois tarkoituksella.

    case TREE_VIEW:
```

```
    // Lisätään puun valikkokohdat.
    break;
}
// Näytetään valikko.
```

Ensimmäinen *case*-kohta yhdistää kahden kohdan toiminnot ilman, että minun pitää kirjoittaa koodia kahdesti tai lisätä kahta saman metodin kutsua. C#-kielen sunnittelijat päättivät kuitenkin jättää ominaisuuden pois. Vaikka se on kätevä, se ei riittänyt voittamaan riskejä, jotka johtuivat siitä, että suurin osa tilanteista, joissa *break*-käsky puuttuu, se on jäänyt pois epähuomiossa, joka aiheuttaa vaikeasti löydettäviä virheitä. Jotta saisit edellä olleen toimimaan C#:ssa, sinun pitää käyttää *if*-käskyä:

```
// Luodaan valikko dynaamisesti.
if (itemSelected == TABLE)
{
    // Lisätään kyseisen taulun valikkokohdat.
}
// Lisätään puun valikkokohdat.
// Näytetään valikko.
```

Toistokäskyt

C#:ssa voit käskyjen *while*, *do/while*, *for* ja *foreach* avulla suorittaa käskyjen toistoa eli tehdä silmukoita. Kussakin tapauksessa yksinkertainen tai yhdistetty käsky suoritetaan, kunnes Boolean-lausekkeen arvoksi tulee *true*, lukuunottamatta *foreach*-käskyä, joka suoritetaan jokaisella sen määrittämällä objektilla.

while-käsky

while-käskyn syntaksi on seuraava:

while (*Boolean-lauseke*)
käsky(t)

Voimme muokata aiemmin tässä luvussa esitettyä numeronarvausesimerkkiä *while*-käskyn avulla niin, että jatkat peliä, kunnes arvaat oikean numeron tai päätät lopettaa pelin. Muutettu koodi alkaa seuraavan sivun alusta.

```
using System;

class WhileApp
{
```

```
const int MIN = 1;
const int MAX = 10;
const string QUIT_CHAR = “Q";

public static void Main()
{
    Random rnd = new Random();
    double correctNumber;

    string inputString;
    int userGuess;

    bool correctGuess = false;
    bool userQuit = false;

    while (!correctGuess && !userQuit)
    {
        correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write
            (“Guess a number between {0} and {1}...({2} to quit)",
            MIN, MAX, QUIT_CHAR);
        inputString = Console.ReadLine();

        if (0 == string.Compare(inputString, QUIT_CHAR, true))
            userQuit = true;
        else
        {
            userGuess = inputString.ToInt32();
            correctGuess = (userGuess == correctNumber);

            Console.WriteLine
                (“The correct number was {0}\n",
                 correctNumber);
        }
    }

    if (correctGuess && !userQuit)
    {
        Console.WriteLine(“Congratulations!”);
    }
    else
    {
        Console.WriteLine(“Maybe next time!”);
    }
```

```
    }
}
```

Ohjelman kirjoittaminen ja suorittaminen aiheuttaa esimerkiksi seuraavanlaisen tuloksen:

```
C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)3
The correct number was 5

Guess a number between 1 and 10...(Q to quit)5
The correct number was 5

Congratulations!

C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)q
Maybe next time!
```

do/while-käsky

Kun katsot *while*-käskyn syntaksia, huomaat yhden mahdollisen ongelman. Boolean-lausekkeen arvo selvitetään ennen kuin silmukassa oleva käsky suoritetaan. Tämän vuoksi edellisen kappaleen esimerkkiohjelmassa *correctGuess* ja *userQuit* -muuttujat alustettiin arvoon *false*, jotta varmistettiin pääsy *while*-silmukkaan. Silmukan sisällä arvot ohjaavat sitä, onko käyttäjä arvannut oikein vai haluaako hän lopettaa. Mutta entä jos haluamme ilman erillisiä alustuksia varmistaa, että silmukan käskyt suoritetaan vähintään kerran? Tätä varten C#:ssa on *do/while* -käsky.

do/while-käskyn syntaksi on seuraava:

do
käsky(t)
while (*Boolean-lauseke*)

Koska *while*-käskyn Boolean-lausekkeen arvo selvitetään vasta silmukan lopussa, suoritetaan silmukan sisällä olevat käskyt vähintään kerran. Numeronarvausesimerkki uudelleenkirjoitettuna *do/while*-käskyä hyödyntäen on seuraavalla sivulla.

```
using System;

class DoWhileApp
```

```
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = “Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess = -1;

        bool userHasNotQuit = true;

        do
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                (“Guess a number between {0} and {1}...({2} to quit)",
                MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
                userHasNotQuit = false;
            else
            {
                userGuess = inputString.ToInt32();
                Console.WriteLine
                    (“The correct number was {0}\n", correctNumber);
            }
        } while (userGuess != correctNumber
            && userHasNotQuit);

        if (userHasNotQuit
            && userGuess == correctNumber)
        {
            Console.WriteLine(“Congratulations!”);
        }
        else // wrong answer!
        {
            Console.WriteLine(“Maybe next time!”);
        }
    }
}
```

Tämän sovelluksen toiminnallisuus on sama kuin *while*-esimerkin. Ainoa ero on silmukan ohjauksessa. Käytännössä olen huomannut, että *while*-käskyä käytetään useammin

kuin *do/while*-käskyä. Koska voit kuitenkin helposti ohjata silmukkaan pääsyä alustamalla Boolean-muuttujan, on valinta käskyjen välillä kiinni henkilökohtaisesta mieltymyksestä.

for-käsky

Yleisin silmukka, jonka olet nähnyt ja jota olet käyttänyt, on *for*-käsky. Se muodostuu kolmesta osasta. Yksi suorittaa alustukset silmukan alussa. Tämä osa suoritetaan vain kerran. Toinen osa on ehtolause, joka määrittelee vieläkö silmukka suoritetaan. Ja viimeinen osa on yleensä (mutta ei välttämättä) se, jossa silmukan jatkumista ohjaavaa laskuria kasvatetaan. Tämä laskuri on se, mitä kakkososan ehtolauseessa yleensä tutkitaan. *for*-käskyn syntaksi on seuraava:

for (*alustus*; *Boolean-lauseke*; *laskurin kasvatus*)
    käsky(t)

Mikä tahansa kolmesta osasta voi olla tyhjä. Kun Boolean-lausekkeen arvo on *false*, ohjelman suoritus siirtyy silmukan alusta silmukan loppusulkeen jälkeiselle riville. Siten *for*-käsky toimii aivan kuin *while*-käsky, mutta se antaa sinulla kaksi lisäosaa (alustuksen ja laskurin kasvattamisen). Seuraavassa *for*-silmukkaesimerkki, joka näyttää tulostuvat ASCII-merkit:

```
using System;

class ForTestApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i++)
        {
            Console.WriteLine(“{0}={1}", i, (char)i);
        }
    }
}
```

Tapahtumien järjestys tässä *for*-silmukassa on seuraava:

1. Arvotyyppiselle muuttujalle (*i*) varataan tila pinosta ja se alustetaan arvoon 33. Huomaa, että tämän muuttujan näkyvyysalue on vain silmukka eli se lakkaa olemasta heti kun silmukka päättyy.
2. Silmukan yhdistettyä käskyä suoritetaan niin kauan kuin muuttujan *i* arvo on pienempi kuin 126. Tässä esimerkissä käytän silmukassa yhdistettyä käskyä.

Koska *for*-silmukka kuitenkin sisältää vain yhden käskyrivin, voisin kirjoittaa silmukan myös ilman sulkuja ja lopputulos olisi sama.

3. Jokaisen silmukan käskyn suorituksen (kierroksen) jälkeen *i*-muuttujan arvoa kasvatetaan yhdellä.

Sisäkkäiset silmukat

For-silmukassa olevien käskyjen joukossa sinulla voi olla toisia *for*-silmukoita. Tällaista rakennetta kutsutaan *sisäkkäisiksi silmukoiksi* (nested loops). Olen lisännyt edellisen kappaleen esimerkkiin sisäkkäisen silmukan, jolla saan riville tulostumaan kolme merkkiä entisen yhden sijaan:

```
using System;

class NestedForApp
{
    const int StartChar = 33;
    const int EndChar = 125;
    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i+=CharactersPerLine)
        {
            for (int j = 0; j < CharactersPerLine;  j++)
            {
                Console.Write("{0}={1} ", i+j, (char)(i+j));
            }
            Console.WriteLine("");
        }
    }

}
```

Huomaa, että ulommassa silmukassa määrittelemäni i-muuttuja on käytettävissä sisemmässä silmukassa eli sisempi silmukka on sen näkyvyysalueella. Sen sijaan muuttuja j ei ole käytettävissä ulommassa silmukassa.

Pilkku-operaattorin käyttäminen

Pilkkua voi käyttää parametriluettelon erottimen lisäksi for-käskyssä operaattorina. Sekä *for*-käskyn alustus että laskurin kasvatus -osassa pilkku-operaattoria voidaan käyttää erottamaan peräkkäin suoritettavia käskyjä. Seuraavassa esimerkissä olen muokannut edellisen kappaleen esimerkkiä ja tehnyt siitä yhden silmukan ohjelman käyttämällä pilkkua operaattorina:

```
using System;
```

```
class CommaOpApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar, j = 1; i <= EndChar; i++, j++)
        {
            Console.Write("{0}={1} ", i, (char)i);
            if (0 == (j % CharactersPerLine))
            {
                // New line if j is divisible by 3.
                Console.WriteLine("");
            }
        }
    }
}
```

Pilkku-operaattorin käyttäminen *for*-käskyssä voi olla tehokasta, mutta se voi myös johtaa inhottavaan ja vaikeasti ylläpidettävään koodiin. Vakioiden lisäyksestä huolimatta seuraava koodi on esimerkki teknisesti oikeasta mutta muuten ikävän hankalasta pilkku-operaattorin käytöstä:

```
using System;

class CommaOp2App
{
    const int StartChar = 33;
    const int EndChar = 125;

    const int CharsPerLine = 3;
    const int NewLine = 13;
    const int Space = 32;

    static public void Main()
    {
        for (int i = StartChar, extra = Space;
            i <= EndChar;
            i++, extra = ((0 == (i - (StartChar-1)) % CharsPerLine)
            ? NewLine : Space))
        {
            Console.Write("{0}={1} {2}", i, (char)i, (char)extra);
        }
    }
}
```

foreach-käsky

Vuosia Visual Basicin tapaisissa kielissä on ollut erikoiskäsky, joka on suunniteltu erityisesti taulukoiden ja kokoelmien läpikäyntiin. C#:ssa on myös sellainen rakenne, nimittäin *foreach*-käsky, jonka syntaksi on seuraava:

foreach (*tyyppi* in *lauseke*)
käsky(t)

Katsotaan seuraavaa taulukko-luokkaa:

```
class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add(“foo”);
        words.Add(“bar”);
        words.Add(“baz”);
    }
}
```

Tiedät, että tämä taulukko voidaan käydä läpi usealla eri silmukkakäskyllä. Useimmille Java ja C++-ohjelmoijalle loogisin tapa toteuttaa tämä tehtävä on kirjoittaa seuraavanlainen koodi:

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add(“foo”);
        words.Add(“bar”);
        words.Add(“baz”);
    }
}

class Foreach1App
{
    public static void Main()
```

```
    {
        MyArray myArray = new MyArray();

        for (int i = 0; i < myArray.words.Count; i++)
        {
            Console.WriteLine("{0}", myArray.words[i]);
        }
    }
}
```

Tämä menetelmä on kuitenkin täynnä ongelmapesäkkeitä:

- Jos *for*-käskyn alustusmuuttujaa (*i*) ei alusteta oikein, voi osa luettelosta jäädä unohduksiin.
- Jos *for-käskyn* Boolean-lauseketta ei tehdä oikein, voi osa luettelosta jäädä unohduksiin.
- Jos *for*-käskyn muuttujan lisäysosa ei ole kunnollinen, voi osa luettelosta jäädä unohduksiin.
- Kokoelmalla ja taulukolla on eri metodit ja ominaisuudet lukumääränsä selvittämiseen.
- Kokoelmalla ja taulukolla on erilainen tapa määrätyn elementin käsittelemiseen.
- *for*-silmukan käskyjen tulee hakea elementti oikean tyyppiseen muuttujaan käsittelyä varten.

Koodi voi siis mennä monella tavalla vikaan. *foreach*-käskyllä voit välttää nämä ongelmapesäkkeet ja käydä minkä tahansa kokoelman tai taulukon alkiot läpi yleisellä tavalla. Edellinen esimerkki voidaan *foreach*-käskyä käyttämällä kirjoittaa uusiksi näin:

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}
```

(jatkuu)

```
class Foreach2App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        foreach (string word in myArray.words)
        {
            Console.WriteLine(“{0}", word);
        }
    }
}
```

Huomaa, miten paljon fiksumpi ohjelmasta tuli *foreach*-käskyn ansiosta. Käyt varmasti jokaisen elementin läpi, koska sinun ei tarvitse itse asettaa silmukkaa ja pyytää elementtien määrää ja lisäksi silmukka sijoittaa automaattisesti elementin nimeämääsi muuttujaan. Sinun pitää silmukan käskyssä vain viitata tuohon muuttujaan.

Haarautuminen hyppykäskyillä

Minkä tahansa edellisissä kappaleissa esiteltyjen silmukkakäskyjen sisällä voit ohjata ohjelman suoritusta seuraavilla, hyppykäskyjen nimellä tunnetuilla käskyillä: *break*, *continue*, *goto* ja *return*.

break-käsky

break-käskyn avulla päätät sen sisimmän silmukan tai ehtokäskyn suorituksen, jossa *break*-käsky sjaitsee. Ohjelman suoritus siirtyy silmukkaa tai ehtokäskyä seuraavalle koodiriville. *break*-käskyllä on yksinkertaisin mahdollisin rakenne, koska sillä ei ole parametreja. Seuraavalla tavalla pääset välittömästi ulos silmukasta tai ehtorakenteesta:

```
break
```

Seuraavassa esimerkissä sovellus tulostaa jokaisen numeron väliltä 1..100, joka on tasan jaollinen luvulla 6. Kun laskurin arvo saavuttaa 66, keskeytyy *for*-silmukan suoritus kuitenkin *break*-käskyn ansiosta.

```
using System;

class BreakTestApp
{
    public static void Main()
```

```
    {
        for (int i = 1; i <= 100; i ++)
        {
            if (0 == i % 6)
            {
                Console.WriteLine(i);
            }

            if (i == 66)
            {
                break;
            }
        }
    }
}

}
```

Päättymättömän silmukan keskeyttäminen

Toinen käyttöpaikka *break*-käskylle löytyy päättymättömästä silmukasta, josta ohjelman suoritus pääsee ulos vain sen avulla. Seuraava esimerkki näyttää menetelmän, jolla aiemmin tässä luvussa toteutettu numeronarvausesmerkki toteutetaan päättymättömällä silmukalla ja *break*-käskyllä. Huomaa, että muutin *while*-käskyn *while(true)*-käskyksi, jolloin se ei pääty ennen kuin ohjelma suoritus kohtaa *break*-käskyn.

```
using System;

class InfiniteLoopApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = “Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess;

        bool correctGuess = false;
        bool userQuit = false;
```

(jatkuu)

```
            while(true)
            {
                correctNumber = rnd.NextDouble() * MAX;
                correctNumber = Math.Round(correctNumber);

                Console.Write
                    (“Guess a number between {0} and {1}...({2} to quit)",
                    MIN, MAX, QUIT_CHAR);
                inputString = Console.ReadLine();

                if (0 == string.Compare(inputString, QUIT_CHAR, true))
                {
                    userQuit = true;
                    break;
                }
                else
                {
                    userGuess = inputString.ToInt32();
                    correctGuess = (userGuess == correctNumber);

                    if ((correctGuess = (userGuess == correctNumber)))
                    {
                        break;
                    }
                    else
                    {

                        Console.WriteLine
                            (“The correct number was {0}\n", correctNumber);
                    }
                }
            }

            if (correctGuess && !userQuit)
            {
                Console.WriteLine(“Congratulations!”);
            }
            else
            {
                Console.WriteLine(“Maybe next time!”);
            }
        }
}
```

Vielä yksi seikka: olisin voinut tässä käyttää tyhjää *for*-käskyä muodossa *for(;;)* käskyn *while(true)* sijasta. Ne toimivat samalla tavalla, joten kyse on makuasiasta.

continue-käsky

Voit *continue*-käskyllä ohjata silmukan suoritusta *break*-käskyn tapaan. Sen sijaan, että lopettaisi koko silmukan suorittamisen, *continue*-käsky pysäyttää menossa olevan "kierroksen" ja palauttaa suorituksen takaisin silmukan alkuun uutta kierrosta varten. Seuraavassa esimerkissä minulla on merkkijonotaulukko, josta haluan tutkia tupla-arvot. Yksi tapa olisi käydä taulukko läpi sisäkkäisissä silmukoissa ja verrata elementtejä toisiinsa. En kuitenkaan halua verrata elementtiä itseensä, koska silloin saisin virheellisen lopputuloksen. Jos ensimmäinen indeksi (*i*) taulukkoon on sama kuin toinen indeksi (*j*), niin minulla on sama elementti enkä halua verrata niitä. Tällöin käytän *continue*-käskyä menossa olevan kierroksen katkaisemiseen ja suorituksen palauttamiseen takaisin silmukan alkuun.

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
        words.Add("bar");
        words.Add("ba");
        words.Add("foo");
    }
}

class ContinueApp
{
    public static void Main()
    {
        MyArray myArray = new MyArray();
        ArrayList dupes = new ArrayList();

        Console.WriteLine("Processing array...");
        for (int i = 0; i < myArray.words.Count; i++)
        {
            for (int j = 0; j < myArray.words.Count; j++)
```

(jatkuu)

```
            {
                if (i == j) continue;

                if (myArray.words[i] == myArray.words[j]
                    && !dupes.Contains(j))
                {
                    dupes.Add(i);
                    Console.WriteLine("'{0}' appears on lines {1} and {2}",
                        myArray.words[i],
                        i + 1,
                        j + 1);
                }
            }
        }
        Console.WriteLine("There were {0} duplicates found",
            ((dupes.Count > 0) ? dupes.Count.ToString() : "no"));
    }
}
```

Olisin voinut käyttää taulukon läpikäyntiin myös *foreach*-silmukkaa. Tällä kerralla halusin kuitenkin pitää kirjaa elementistä, jossa olen ja siihen *for*-silmukka on paras tapa.

Epäsuosittu *goto*-käsky

Luultavasti mitään muuta rakennetta ohjelmoinnin historiassa ei ole parjattu niin paljon kuin *goto*-käskyä. Ennenkuin tutkimme *goto*-käskyn syntaksia ja muutamia käyttöpaikkoja, kannattaakin ehkä tarkastella, miksi jotkut ihmiset ovat niin kovasti tämän käskyn käyttöä vastaan ja ongelmatilanteita, joita sen avulla voi ratkaista.

goto-käsky: (Hyvin) lyhyt historia

goto-käsky joutui epäsuosioon Edsger W. Dijkstran vuonna 1968 julkaiseman artikkelin "Go To Statement Considered Harmful" takia. Siihen aikaan käytiin väittelyä rakenteisesta ohjelmoinnista. Ikävä kyllä paljon vähemmän mielenkiintoa kohdistui rakenteiseen ohjelmointiin yleensä kuin sen suhteellisen pieneen yksityiskohtaan: pitäisikö määrättyjen käskyjen, kuten *goto*-käskyn, olla nykyaikaisessa ohjelmointikielessä? Kuten turhan usein on tapana, monet ihmiset ottivat Dijkstran neuvon kirjaimellisesti kuvitellen, että kaikki *goto*-käskyn käyttö on pahasta ja että sen käyttöä tulee välttää kaikin mahdollisin tavoin.

goto-käskyn käytön ongelmat eivät johdu itse sanasta vaan sen käytöstä väärissä paikoissa. *goto*-käsky voi olla hyödyllinen työkalu rakenteisessa ohjelmoinnissa ja sen avulla voidaan tehdä ilmaisevampaa ohjelmakoodia kuin muilla valinta- tai toistorakenteilla. Yksi

esimerkki sellaisesta on “puolitoista silmukkaa” -esimerkki, jonka Dijkstra esitti. Tässä ongelman perinteinen ratkaisu pseudokoodilla:

```
loop
    read in a value
    if value == sentinel then exit
    process the value
end loop
```

Ulostulo silmukasta tapahtuu vain silmukan keskellä olevan exit-käskyn kautta. Tämä *loop/exit/end loop* -kierto on kuitenkin eräille ihmisille melko häiritsevä. Nämä ihmiset, joiden mielestä *goto*-käskyä käyttävät ihmiset ovat paholaisista seuraavia (kumpaan suuntaan?), tekisivät saman toiminnon seuraavasti:

```
read in a value
while value != sentinel
    process the value
    read in a value
end while
```

Valitettavasti, kuten Eric S. Roberts Stanford Universitysta osoitti, tällä jälkimmäisellä ratkaisulla on kaksi heikkoutta. Ensinnäkin se tarvitsee kaksi kertaa arvon lukukäskyn. Aina, kun sama koodi kirjoitetaan kahdesti, aiheutetaan ylläpito-ongelma, koska jos toista muutetaan, pitää muistaa tehdä muutos myös toiseen. Toinen ongelma on hienovaraisempi ja luultavasti vahingollisempi. Avain selkeän ja siten helposti ylläpidettävän koodin kirjoittamiseen on kirjoittaa koodia luonnollisen ajattelutavan mukaan. Ei ohjelmakoodilla kirjoitetussa kuvauksessa äskeisen pseudokoodin suorittamien toimenpiteiden selostus voisi olla tällainen: Ensinnäkin minun pitää lukea arvo. Jos arvo on “sentinel”, lopetan. Jos ei ole, käsittelen arvoa ja jatkan seuraavaan arvoon. Siten jälkimmänen koodi jättää huomioimatta *exit*-käskyn, vaikka koodin on hyvä toistaa ongelman ratkaisun luonnollista ajatuskulkua. Katsotaan nyt muutamia tilanteita, joissa *goto*-käsky on paras tapa ohjata ohjelman suoritusta.

goto-käskyn käyttäminen

goto-käsky voi esiintyä seuraavissa muodoissa:

goto *tunnistin*
goto case *vakiolauseke*
goto default

Ensimmäisessä *goto*-käskyn muodossa tunnistin on nimeämiskäsky. Sen muoto on yksinkertaisesti

tunnistin:

Jos tunnistinta ei ole olemassa metodissa, tuloksena on käännösvirhe. Toinen tärkeä sääntö muistaa on se, että *goto*-käskyä voidaan käyttää ulostuloon sisäkkäisistä silmukoista. Jos *goto*-käsky ei kuitenkaan ole tunnistimen näkyvyysalueella, saadaan kääntäjän virhe. Siksi et voi hypätä sisäkkäiseen silmukkaan.

Seuraavassa esimerkissä sovellus käy läpi yksinkertaisen taulukon lukien kunkin arvon, kunnes kohtaa etsimänsä, jolloin tullaan silmukasta ulos. Huomaa, että *goto*-käsky todella käyttäytyy kuin *break*-käsky siinä mielessä, että se aiheuttaa ohjelman suorituksen siirtymisen ulos *foreach*-silmukasta.

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;
    public const string TerminatingWord = “stop";

    public MyArray()
    {
        words = new ArrayList();

        for (int i = 1; i <= 5; i++) words.Add(i.ToString());
        words.Add(TerminatingWord);
        for (int i = 6; i <= 10; i++) words.Add(i.ToString());
    }
}

class Goto1App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        Console.WriteLine(“Processing array...”);

        foreach (string word in myArray.words)
        {
            if (word == MyArray.TerminatingWord) goto finished;
            Console.WriteLine(word);
        }

        finished:
            Console.WriteLine(“Finished processing array”);
    }
}
```

Tähän *goto*-käskyn käyttöön liittyen voisi joku sanoa, että *break*-käskyä olisi voinut käyttää yhtä tehokkaasti, eikä olisi tarvinnut määritellä tunnistinta. Katsomme nyt muita *goto*-käskyn muotoja ja tulet näkemään sellaisia ongelmia, jotka voidaan ratkaista vain *goto*-käskyllä.

switch-käskyn kappaleessa puhuin siitä, että C# ei tue läpijuoksua. Vaikka C# tukisikin sitä, se ei ratkaisisi seuraavaa ongelmaa. Meillä on *Payment*-luokka (tuttu edellisistä kappaleista), joka hyväksyy useita erilaisia maksutapoja: Visa, American Express, MasterCard, käteinen ja laskutus. Koska Visa, American Express ja MasterCard ovat kaikki luottokortteja, voimme yhdistää ne yhteen case-kohtaan ja käsitellä niitä samalla tavalla. Laskutuksen ollessa kyseessä meidän tulee kutsua sen hoitavaa metodia ja käteismaksun tapauksessa meidän pitää ainoastaan tulostaa kuitti. Meidän tulee tulostaa kuitti myös kaikissa muissakin tapauksissa. Miten meillä voi olla kolme erillistä case-kohtaa siten, että kahdesta ensimmäisestä (luottokorttimaksusta ja laskutuksesta) molemmista siirrytään käteinen-haaraan, kun ne on suoritettu? Kuten näet seuraavassa koodissa, tämä on hyvä esimerkki sellaisesta ongelmasta, jossa tulee käyttää goto-käskyä:

```
using System;

enum Tenders : int
{
    ChargeOff,
    Cash,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
```

(jatkuu)

```
        set
        {
            this.tender = value;
        }
    }

    protected void ChargeOff()
    {
        Console.WriteLine("Charge off.");
    }

    protected bool ValidateCreditCard()
    {
        Console.WriteLine("Card approved.");
        return true;
    }

    protected void ChargeCreditCard()
    {
        Console.WriteLine("Credit Card charged");
    }

    protected void PrintReceipt()
    {
        Console.WriteLine("Thank you and come again.");
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.ChargeOff:
                ChargeOff();
                goto case Tenders.Cash;

            case (int)Tenders.Visa:
            case (int)Tenders.MasterCard:
            case (int)Tenders.AmericanExpress:
                if (ValidateCreditCard())
                    ChargeCreditCard();
                goto case Tenders.Cash;

            case (int)Tenders.Cash:
                PrintReceipt();
                break;
```

```
            default:
                Console.WriteLine("\nSorry - Invalid tender.");
                break;
        }
    }
}

class GotoCaseApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}
```

Sen sijaan että olisimme ratkaisseet ongelman "epäluonnollisesti", yksinkertaisesti kerromme kääntäjälle, että kun luottokortin tai laskutuksen case-kohta on suoritettu, haluamme siirtyä käteisen case-kohtaan. Huomaa vielä, että jos C#:ssa siirryt pois case-kohdasta, et saa käyttää *break*-käskyä, sillä se aiheuttaa kääntäjävirheen "unreachable code."

Viimeisen *goto*-käskyn muodon avulla voit siirtyä *switch*-käskyn *default*-tunnisteeseen ja siten se antaa sinulle yhden lisäsyyn kirjoittaa koodilohko, joka voidaan suorittaa *switch*-käskyssä useiden vertailujen tuloksena

return-käsky

return-käskyllä on kaksi merkitystä. Se määrittelee arvon, joka suoritettavasta koodista palautetaan kutsuvalle ohjelmalle (kun suoritettavan koodin paluuarvoksi ei ole määritelty void) ja se aiheuttaa välittömän paluun kutsuvaan ohjelmaan. *return*-käskyn syntaksi on seuraava:

return [*return-lauseke*]

Kun kääntäjä kohtaa metodin *return*-käskyn, jossa on määritelty *return-lauseke*, se tutkii, voidaanko *return-lauseke* automaattisesti muuntaa nykyisen metodin määritellyksi paluuarvoksi. Tuon muunnoksen tulos palautetaan sitten kutsuvalle ohjelmalle.

Kun käytetään *return*-käskyä poikkeusten käsittelyssä, sinun pitää ymmärtää muutamat säännöt. Jos *return*-käsky on *try*-lohkossa, johon on liitetty *finally*-lohko, ohjelman suoritus siirtyy todellisuudessa *finally*-lohkon ensimmäiselle riville ja kun se tulee suoritettua, ohjelman suoritus palaa kutsujalle. Jos *try*-lohko on toisen *try*-lohkon sisällä, ohjelman suoritus "kuplii" näin kutsuketjua ylös, kunnes viimeinen *finally*-lohko on suoritettu.

Yhteenveto

C#:n ohjauskäskyillä voit ohjata ohjelman suoritusta. Ohjauskäskyt voidaan jakaa kolmeen ryhmään. Valintakäskyihin kuuluvat mm. *if* ja *switch*, toistokäskyihin *while*, *for* ja *foreach* ja erilaisiin hyppykäskyihin *break*, *continue*, *goto* ja *return*. Parhaan käskyn valitseminen tämän luvun esimerkkien perusteella auttaa sinua kirjoittamaan paremmin jäsenneltyjä ja ylläpidettävämpiä sovelluksia.