14

Delegaatit ja tapahtumakäsittelijät

Toinen käyttökelpoinen keksintö C#-kielessä ovat delegaatit. Periaatteessa ne palvelevat samaa tarkoitusta kuin C++:n funktio-osoittimet. Delegaatit ovat kuitenkin tyyppiturvattuja, turvallisia hallittuja objekteja. Tämä tarkoittaa sitä, että ajonaikainen ympäristö varmistaa, että delegaatti osoittaa kelvolliseen metodiin, joka puolestaa tarkoittaa sitä, että sinulla on käytössäsi kaikki funktio-osoittimien edut ilman niihin liittyviä vaaroja, esimerkiksi virheellistä osoitetta tai vaaraa, että delegaatti vioittaisi muiden objektien muistialueita. Tässä luvussa tutkimme delegaatteja, vertaamme niitä rajapintoihin, käymme läpi delegaatin määrittelyn syntaksin ja erilaiset ongelmat, joiden ratkaisemiseksi ne on kehitetty. Näemme myös useita delegaattien käyttöesimerkkejä sekä takaisinkutsumetodeissa että asynkronisessa tapahtumakäsittelyssä.

Luvussa 9, "Rajapinnat," näimme, miten C#:ssa määritellään ja toteutetaan rajapintoja. Kuten muistat, käsitteellisessä mielessä rajapinnat ovat yksinkertaisesti kahden erillisen koodin välisiä sopimuksia. Rajapinnat ovat paljon luokkien kaltaisia siinä, että ne määritellään käännöksen aikana ja ne voivat sisältää metodeja, ominaisuuksia, indeksoijia ja tapahtumia. Delegaatit toisaalta viittaavat yhteen metodiin ja ne määritellään suorituksen aikana. Delegaateilla on kaksi tärkeää käyttöpaikkaa C#-ohjelmissa: takaisinkutsuissa ja tapahtumakäsittelyssä. Aloitetaan puhumalla takaisinkutsumetodeista.

Delegaattien käyttö takaisinkutsumetodeina

Microsoft Windows -ohjelmoinnissa laajalti käytettyjä takaisinkutsumetodeja tarvitaan, kun sinun pitää välittää funktio-osoitin toiselle funktiolle, joka sitten kutsuu takaisin ensimmäistä metodia funktio-osoittimen avulla. Esimerkkinä tästä vaikkapa Win32 APIn

EnumWindows-funktio. Tämä funktio luetteloi kaikki ruudulla olevat ylimmän tason ikkunat kutsuen kunkin ikkunan sisältämää funktiota. Takaisinkutsu palvelee monia tarkoituksia, joista seuraavassa on lueteltu yleisimmät:

- **Asynkroninen käsittely** Takaisinkutsumetodeja käytetään asynkronisessa käsittelyssä, kun kutsuttavan koodin suoritus kestää ajallisesti kauan. Tavallinen tilanne on tämä: Asiakasohjelma kutsuu metodia välittäen sille takaisinkutsumetodin. Kutsuttu metodi käynnistää säikeen ja palaa välittömästi sen jälkeen. Säie tekee sitten varsinaisen työn ja kutsuu takaisinkutsufunktiota tarpeen mukaan. Tästä on se ilmiselvä etu, että asiakasohjelma voi jatkaa toimintaansa eikä se jää jumiin odottamaan mahdollisesti kauan kestävän funktion suoritusta.
- **Koodin lisääminen luokan koodiksi** Toinen yleinen takaisinkutsumetodien käyttötilanne on silloin, kun luokka sallii asiakasohjelman määrittelevän metodin, jota luokka kutsuu erikoistoiminnon suorittamiseksi. Katsotaan selventävää esimerkkiä Windowsista. Käyttämällä Windowsin *ListBox*-luokkaa, voit määritellä, että sen rivit lajitellaan nousevaan tai laskevaan järjestykseen. Peruslajittelun lisäksi *ListBox*-luokka ei anna sinulle lajittelun suhteen liikkumavaraa. Sen sijaan se antaa sinulle mahdollisuuden määritellä takaisinkutsufunktion lajittelun suorittamiseksi. Tällöin, kun *ListBox* tekee lajittelun, se kutsuukin takaisinkutsufunktiota ja koodisi voi silloin tehdä lajittelun omien tarpeidesi mukaan.

Katsotaan nyt esimerkkiä delegaatin määrittelystä ja käytöstä. Tässä esimerkissä meillä on tietokannan hallintaluokka, joka pitää kirjaa kaikista tietokannan aktiivisista yhteyksistä ja tarjoaa metodin yhteyksien luetteloimiseksi. Oletetaan, että hallintaluokka on etäpalvelimella, jolloin on järkevää tehdä metodista asynkroninen ja sallia asiakasohjelman käyttävän takaisinkutsumetodia. Todellisessa elämässä tekisit monisäikeisen sovelluksen, jotta se olisi aito asynkroninen sovellus. Pitääksemme sovelluksen kuitenkin yksinkertaisena ja koska emme ole vielä käsitelleet monisäikeistä ohjelmointia, jätämme sen pois.

Määritellään ensin kaksi pääluokkaa: *DBManager* ja *DBConnection*.

```
class DBConnection
{
    §
}
```

```
class DBManager
{
  static DBConnection[] activeConnections;

  §

  public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}
```

EnumConnectionsCallback-metodi on delegaatti ja se määritellään kirjoittamalla avainsana *delegate* metodin esittelyn eteen. Näet, että delegaatin paluuarvoksi on määritelty *void* ja että se ottaa yhden parametrin: *DBConnection*-objektin. *EnumConnections*-metodi on määritelty ottamaan ainoaksi parametrikseen *EnumConnectionsCallback*-metodin. Kutsuessamme *DBManager.EnumConnections*-metodia, meidän pitää välittää sille parametrina vain instantioitu *DBManager.EnumConnectionCallback*-delegaatti.

Se tehdään käyttämällä *new*-avainsanaa välittämällä sille parametrina sen metodin nimi, jolla on sama otsikkorivi kuin delegaatilla. Tässä esimerkki:

```
DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

DBManager.EnumConnections(myCallback);
```

Voit myös yhdistää nämä yhdeksi kutsuksi näin:

```
DBManager.EnumConnections(new
    DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));
```

Siinä kaikki delegaatin perusrakenteesta. Katsotaan nyt täydellistä esimerkkisovellusta:

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }
```

(jatkuu)

```
        protected string Name;
        public string name
        {
            get
            {
                return this.Name;
            }
            set
            {
                this.Name = value;
            }
        }
    }

    class DBManager
    {
        static DBConnection[] activeConnections;
        public void AddConnections()
        {
            activeConnections = new DBConnection[5];
            for (int i = 0; i < 5; i++)
            {
                activeConnections[i] =
new DBConnection("DBConnection " + (i + 1));
            }
        }

        public delegate void EnumConnectionsCallback(DBConnection connection);
        public static void EnumConnections(EnumConnectionsCallback callback)
        {
            foreach (DBConnection connection in activeConnections)
            {
                callback(connection);
            }
        }
    }

    class Delegate1App
    {
        public static void ActiveConnectionsCallback(DBConnection connection)
        {
            Console.WriteLine("Callback method called for "
                              + connection.name);
        }
```

```
      public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

    DBManager.EnumConnectionsCallback myCallback =
      new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

    DBManager.EnumConnections(myCallback);
    }
}
```

Tämän sovelluksen kääntäminen ja ajaminen saa aikaan seuraavat tulokset:

```
Callback method called for DBConnection 1
Callback method called for DBConnection 2
Callback method called for DBConnection 3
Callback method called for DBConnection 4
Callback method called for DBConnection 5
```

Delegaattien määritteleminen staattisiksi jäseniksi

Koska on aika massiivinen operaatio, että asiakasohjelman pitää instantioida delegaatti joka kerta, kun sitä käytetään, C# antaa mahdollisuuden määritellä staattisen jäsenmetodin, jota käytetään delegaatin luontiin. Seuraavassa on edellisen kappaleen esimerkki muutettuna tähän muotoon. Huomaa, että delegaatti määritellään nyt *myCallback*-luokan staattiseksi jäseneksi ja huomaa myös, että tätä jäsentä voidaan käyttää *Main*-metodissa ilman, että asiakasohjelman pitää instantioida delegaattia.

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
```

(jatkuu)

```
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] = new
DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class Delegate2App
{
    public static DBManager.EnumConnectionsCallback myCallback =
        new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine ("Callback method called for " +
                           connection.name);
    }

    public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(myCallback);
    }
}
```

Huomaa Koska nimistandardi delegaateille suosittelee sanan *Callback* lisäämistä sen metodin nimeen, joka ottaa delegaatin parametrikseen, on helppoa epähuomiossa käyttää metodin nimeä delegaatin nimen sijasta. Silloin saat jotakuinkin hämäävän kääntäjän virheilmoituksen, joka ilmoittaa, että olet käyttänyt metodia paikassa, jossa odotetaan luokkaa. Jos saat tämän virheen, muista, että todellinen virhe on siinä, että olet määritellyt metodin delegaatin sijasta.

Delegaattien luominen vain tarpeen vaatiessa

Kahdessa tähän mennessä olleessa esimerkissä delegaatti on luotu, on sitä tarvittu tai ei. Se sopii näissä esimerkeissä, koska tiedän, että niitä tullaan joka kerta kutsumaan. Kun määrittelet delegaatteja, on kuitenkin tärkeää miettiä milloin ne luodaan. Sanotaan vaikkapa, että jonkun delegaatin luominen kestää kauan etkä halua tehdä sitä tarpeettomasti. Tilanteissa, joissa tiedät, että asiakasohjelma ei välttämättä kutsu takaisinkutsumetodia, voit viivyttää delegaatin luontia kunnes sitä tarvitaan sijoittamalla sen instantioinnin ominaisuuteen. Esimerkki tästä on seuraavassa, jossa olen muokannut *DBManager*-luokkaa siten, että se käyttää vain-luku-ominaisuutta (koska setter-metodi puuttuu) delegaatin instantiointiin. Delegaattia ei luoda ennen kuin ominaisuutta tarvitaan.

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
        {
            this.Name = value;
        }
```

(jatkuu)

```
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] = new
                DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class Delegate3App
{
    public DBManager.EnumConnectionsCallback myCallback
    {
        get
        {
            return new DBManager.EnumConnectionsCallback
                            (ActiveConnectionsCallback);
        }
    }

    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine
            ("Callback method called for " + connection.name);
    }

    public static void Main()
    {
        Delegate3App app = new Delegate3App();
```

```
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(app.myCallback);
    }
}
```

Delegaattikooste

Mahdollisuus tehdä delegaattikooste (luomalla yksi delegaatti useasta) on yksi niistä ominaisuuksista, jotka eivät aluksi tunnu kovin käyttökelpoisilta, mutta jos tulet sellaista joskus tarvitsemaan, olet onnellinen, että C#-kehitystiimi ajatteli sitä. Katsotaanpa joitakin esimerkkejä, joissa delegaattikooste on tarpeellinen. Ensimmäisessä esimerkissä sinulla on hajautettu sovellus ja luokka, joka käy läpi annetun varastopaikan osia kutsuen takaisinkutsumetodia jokaisella osalla, jonka vapaa-saldo on vähemmän kuin 50. Todenmukaisemmassa sovelluksessa kaavassa tulisi ottaa huomioon myös tilauksessa ja matkalla-tilassa olevat määrät ja niin edelleen. Mutta pidetään esimerkki yksinkertaisena: jos osan vapaa-saldo on vähemmän kuin 50, tapahtuu jotain.

Juju on siinä, että haluamme kaksi erilaista metodikutsua, jos osan saldo on alle rajan: haluamme tehdä tapahtumamerkinnän ja sen jälkeen lähettää sähköpostiviestin hankintapäällikölle. Katsotaan nyt, miten luot ohjelmallisesti yhden delegaattikoosteen useasta delegaatista:

```
using System;
using System.Threading;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }

    protected string Sku;
    public string sku
    {
```

(jatkuu)

```
            get
            {
                return this.Sku;
            }
            set
            {
                this.Sku = value;
            }
        }

        protected int OnHand;
        public int onhand
        {
            get
            {
                return this.OnHand;
            }
            set
            {
                this.OnHand = value;
            }
        }
    }

    class InventoryManager
    {
        protected const int MIN_ONHAND = 50;

        public Part[] parts;
        public InventoryManager()
        {
            parts = new Part[5];
            for (int i = 0; i < 5; i++)
            {
                Part part = new Part("Part " + (i + 1));

                Thread.Sleep(10); // Randomizer is seeded by time.

                parts[i] = part;
                Console.WriteLine("Adding part '{0}' on-hand = {1}",
                                  part.sku, part.onhand);
            }
        }

        public delegate void OutOfStockExceptionMethod(Part part);
        public void ProcessInventory(OutOfStockExceptionMethod exception)
        {
```

```
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum on-hand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate1App
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tlogging event...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\temailing Purchasing manager...");
    }

    public static void Main()
    {
        InventoryManager mgr = new InventoryManager();

        InventoryManager.OutOfStockExceptionMethod LogEventCallback =
            new InventoryManager.OutOfStockExceptionMethod(LogEvent);

        InventoryManager.OutOfStockExceptionMethod
            EmailPurchasingMgrCallback = new
            InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

        InventoryManager.OutOfStockExceptionMethod
            OnHandExceptionEventsCallback =
            EmailPurchasingMgrCallback + LogEventCallback;

        mgr.ProcessInventory(OnHandExceptionEventsCallback);
    }
}
```

Sovelluksen ajaminen tuottaa seuraavat tulokset:

```
Adding part 'Part 1' on-hand = 16
Adding part 'Part 2' on-hand = 98
Adding part 'Part 3' on-hand = 65
Adding part 'Part 4' on-hand = 22
Adding part 'Part 5' on-hand = 70

Processing inventory...
Part 1 (16) is below minimum on-hand 50
    logging event...
    emailing Purchasing manager...
Part 4 (22) is below minimum on-hand 50
    logging event...
    emailing Purchasing manager...
```

Käyttämällä kielen tätä ominaisuutta, voimme siis dynaamisesti havaita, mitkä metodit sisältävät takaisinkutsumetodin, yhdistää ne yhdeksi delegaatiksi ja välittää delegaattikooste kuin se olisi yksi delegaatti. Ajonaikainen ympäristö huolehtii automaattisesti, että metodeita kutsutaan järjestyksessä. Lisäksi voit poistaa delegaatteja delegaattikoosteesta käyttämällä minus-operaattoria.

Se tosiasia, että näitä metodeja kutsutaan peräkkäin, saa kysymään tärkeän kysymyksen: miksi ei voi yksinkertaisesti ketjuttaa metodeja yhteen antamalla kunkin metodin kutsua seuraavaa? Tämän kappaleen esimerkissä, jossa meillä on siis vain kaksi metodia ja molempia kutsutaan aina parina peräkkäin, voisimme tehdäkin niin. Mutta tehdään esimerkistä hieman monimutkaisempi. Sanotaan, että meillä on useita varastopaikkoja, joista kukin määrää, mitä metodeja kutsutaan. Esimerkki Varasto1 voi olla keskusvarasto, joten haluamme tehdä tapahtumamerkinnän ja lähettää sähköpostiviestin hankintapäällikölle, kun taas muissa varastopaikoissa osan määrän ollessa alle rajan, teemme tapahtumamerkinnän ja lähetämme sähköpostiviestin tuon varaston päällikölle.

Voimme helposti täyttää nämä vaatimukset luomalla dynaamisesti delegaattikoosteen, joka perustuu käsiteltävään varastopaikkaan. Ilman delegaatteja meidän pitäisi kirjoittaa metodi, joka ei pelkästään määrittelisi, mitä metodeita pitää kutsua vaan myös pitäisi kirjaa, mitä metodeja on jo kutsuttu ja mitä vielä pitää tämän kierroksen aikana kutsua. Kuten näet seuraavasta koodista, delegaattien avulla tästä monimutkaisesta operaatiosta tulee hyvin yksinkertainen.

```
using System;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;
```

```
            Random r = new Random(DateTime.Now.Millisecond);
            double d = r.NextDouble() * 100;

            this.OnHand = (int)d;
        }

        protected string Sku;
        public string sku
        {
            get
            {
                return this.Sku;
            }
            set
            {
                this.Sku = value;
            }
        }

        protected int OnHand;
        public int onhand
        {
            get
            {
                return this.OnHand;
            }
            set
            {
                this.OnHand = value;
            }
        }
    }

    class InventoryManager
    {
        protected const int MIN_ONHAND = 50;

        public Part[] parts;
        public InventoryManager()
        {
            parts = new Part[5];
            for (int i = 0; i < 5; i++)
            {
                Part part = new Part("Part " + (i + 1));
                parts[i] = part;
```

(jatkuu)

```
                Console.WriteLine
                    (“Adding part ‘{0}’ on-hand = {1}",
                     part.sku, part.onhand);
            }
        }

        public delegate void OutOfStockExceptionMethod(Part part);
        public void ProcessInventory(OutOfStockExceptionMethod exception)
        {
            Console.WriteLine(“\nProcessing inventory...”);
            foreach (Part part in parts)
            {
                if (part.onhand < MIN_ONHAND)
                {
                    Console.WriteLine
                        (“{0} ({1}) is below minimum onhand {2}",
                        part.sku, part.onhand, MIN_ONHAND);

                    exception(part);
                }
            }
        }
    }

    class CompositeDelegate2App
    {
        public static void LogEvent(Part part)
        {
            Console.WriteLine(“\tlogging event...”);
        }

        public static void EmailPurchasingMgr(Part part)
        {
            Console.WriteLine(“\temailing Purchasing manager...”);
        }

        public static void EmailStoreMgr(Part part)
        {
            Console.WriteLine(“\temailing store manager...”);
        }

        public static void Main()
        {
            InventoryManager mgr = new InventoryManager();

            InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
                = new InventoryManager.OutOfStockExceptionMethod[3];
```

```
            exceptionMethods[0] = new
                InventoryManager.OutOfStockExceptionMethod
                    (LogEvent);
            exceptionMethods[1] = new
                InventoryManager.OutOfStockExceptionMethod
                    (EmailPurchasingMgr);
            exceptionMethods[2] = new
                InventoryManager.OutOfStockExceptionMethod
                    (EmailStoreMgr);

            int location = 1;

            InventoryManager.OutOfStockExceptionMethod compositeDelegate;

            if (location == 2)
            {
                compositeDelegate =
                    exceptionMethods[0] + exceptionMethods[1];
            }
            else
            {
                compositeDelegate =
                    exceptionMethods[0] + exceptionMethods[2];
            }

            mgr.ProcessInventory(compositeDelegate);
        }
}
```

Nyt sovelluksen kääntäminen ja suorittaminen tuottaa erilaisia tuloksia riippuen arvosta, jonka annat *location*-muuttujalle.

Tapahtumien määrittely delegaateilla

Melkein kaikilla Windows-sovelluksilla on jonkinlaisia tarpeita asynkroniseen tapahtumakäsittelyyn. Jotkin näistä tapahtumista ovat yleisiä, kuten Windowsin lähettämät viestit sovelluksen viestijonoon, kun käyttäjä on ohjannut sovellusta jollakin tavalla. Jotkut ovat ongelmaläheisempiä, kuten tarve tulostaa päivitetty lasku.

C#:n tapahtumat noudattavat ”julkaise/tilaa”-menetelmää (publish-subscribe), jossa luokka julkaisee tapahtuman, jonka se voi laukaista ja joukko luokkia voi sen jälkeen tilata ilmoituksen tuosta tapahtumasta. Kun tapahtuma laukeaa, ajonaikainen ympäristö huolehtii, että kaikki sen tilanneet luokat saavat siitä ilmoituksen.

Metodin, jota tapahtuman laukeamisen johdosta kutsutaan, on määritellyt delegaatti. Pidä mielessä tarkat säännöt, jotka koskevat tällä tavalla käytettävää delegaattia: Ensinnäkin,

delegaatti pitää määritellä ottamaan kaksi parametria. Toiseksi, nämä parametrit ovat aina kaksi objektia: objekti, joka laukaisi tapahtuman (julkaisija) ja tapahtuman informaatio-objekti. Lisäksi tämän toisen objektin tulee periytyä .NET Frameworkin *EventArgs*-luokasta.

Sanotaan, että haluamme tarkkailla varastosaldojen muutoksia. Voimme luoda luokan nimeltä *InventoryManager*, jota käytetään aina varastosaldojen päivittämiseen. Tämä *InventoryManager*-luokka julkaisee tapahtuman, joka laukaistaan joka kerta, kun saldo muuttuu esimerkiksi varastokuittauksen, myynnin tai inventointipäivityksen takia. Sitten jokainen luokka, jonka pitää pysyä ajan tasalla, voi tilata tuon tapahtuman. Tämä koodataan seuraavasti C#:lla käyttäen delegaatteja ja tapahtumia:

```
using System;

class InventoryChangeEventArgs : EventArgs
{
    public InventoryChangeEventArgs(string sku, int change)
    {
        this.sku = sku;
        this.change = change;
    }

    string sku;
    public string Sku
    {
        get
        {
            return sku;
        }
    }

    int change;
    public int Change
    {
        get
        {
            return change;
        }
    }
}

class InventoryManager // Publisher.
{
    public delegate void InventoryChangeEventHandler
        (object source, InventoryChangeEventArgs e);
    public event InventoryChangeEventHandler OnInventoryChangeHandler;

    public void UpdateInventory(string sku, int change)
```

```
    {
        if (0 == change)
            return; // No update on null change.

        // Code to update database would go here.

        InventoryChangeEventArgs e = new
            InventoryChangeEventArgs(sku, change);

        if (OnInventoryChangeHandler != null)
            OnInventoryChangeHandler(this, e);
    }
}

class InventoryWatcher // Subscriber.
{
    public InventoryWatcher(InventoryManager inventoryManager)
    {
        this.inventoryManager = inventoryManager;
        inventoryManager.OnInventoryChangeHandler += new
InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
    }
    void OnInventoryChange(object source, InventoryChangeEventArgs e)
    {
        int change = e.Change;
        Console.WriteLine("Part '{0}' was {1} by {2} units",
            e.Sku,
            change > 0 ? "increased" : "decreased",
            Math.Abs(e.Change));
    }
    InventoryManager inventoryManager;
}

class Events1App
{
    public static void Main()
    {
        InventoryManager inventoryManager =
            new InventoryManager();

        InventoryWatcher inventoryWatch =
            new InventoryWatcher(inventoryManager);

        inventoryManager.UpdateInventory("111 006 116", -2);
        inventoryManager.UpdateInventory("111 005 383", 5);
    }
}
```

Katsotaan *InventoryManager*-luokan kahta ensimmäistä jäsentä:

```
public delegate void InventoryChangeEventHandler
                            (object source, InventoryChangeEventArgs e);
public event InventoryChangeEventHandler OnInventoryChangeHandler;
```

Ensimmäinen koodirivi on delegaatti, joka on, kuten nyt tiedät, metodin määrittely. Kuten aiemmin mainitsin, kaikki delegaatit, joita käytetään tapahtumissa, pitää määritellä ottamaan kaksi parametria: julkaisijaobjektin (tässä tapauksessa *source*) ja tapahtuman informaatio-objektin (*EventArgs*-objektista periytyvä). Toinen rivi käyttää *event*-avainsanaa, jäsentyyppiä, jolla määrittelet delegaatin, eli metodin (metodit) jota kutsutaan, kun tapahtuma laukaistaan.

InventoryManager-luokan viimeinen metodi on *UpdateInventory*, jota kutsutaan joka kerta, kun varastosaldo muuttuu. Kuten näet, tämä metodi luo *InventoryChangeEventArgs*-tyyppisen objektin. Tämä objekti välitetään kaikille tilaajille ja sitä käytetään kuvaamaan tapahtumaa.

Katsotaan nyt seuraavia koodirivejä:

```
if (OnInventoryChangeHandler != null)
    OnInventoryChangeHandler(this, e);
```

if-käsky tarkistaa on tapahtumalla yhtään tilaajaa liitettynä *OnInventoryChangeHandler*-metodiin. Jos on, eli *OnInventoryChangeHandler* ei ole *null*, tapahtuma laukaistaan. Siinä todella kaikki, joka tapahtuman julkaisijan puolella pitää tehdä. Katsotaan seuraavaksi tilaajapuolen koodia.

Tilaaja on tässä tapauksessa luokka *InventoryWatcher*. Sen pitää tehdä vain kaksi yksinkertaista asiaa. Ensiksikin sen tulee lisätä itsensä tilaajaksi instantioimalla uusi *InventoryManager.InventoryChangeEventHandler*-tyyppinen delegaatti ja lisäämällä sen *InventoryManager.OnInventoryChangeHandler*-tapahtumaan. Huomaa erityisesti käytetty syntaksi: luokka käyttää *+=*-yhdistelysijoitusoperaattoria lisätessään itsensä tilaajaluetteloon, jotta ei poistaisi edellisiä tilaajia.

```
inventoryManager.OnInventoryChangeHandler
+= new InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
```

Ainoa tarvittava parametri on sen metodin nimi, jota kutsutaan, jos ja kun tapahtuma laukeaa.

Ainoa tehtävä, joka tilaajan tämän lisäksi pitää tehdä, on toteuttaa tapahtumakäsittelijä. Tässä tapauksessa tapahtumakäsittelijä on *InventoryWatcher.OnInventoryChange*, joka tulostaa viestissä osanumeron ja saldon muutoksen.

Tämän sovelluksen suorittava koodi instantioi *InventoryManager* ja *InventoryWatcher*-luokat. Joka kerta, kun *InventoryManager.UpdateInventory*-metodia kutsutaan, tapahtuma laukeaa automaattisesti ja se aiheuttaa *InventoryWatcher.OnInventoryChanged*-metodin kutsumisen.

Yhteenveto

C#:n delegaatit ovat tyyppiturvattuja, turvallisia hallittuja objekteja, jotka palvelevat samaa tarkoitusta kuin C++:n funktio-osoittimet. Delegaatit eroavat luokista ja rajapinnoista siinä, että niitä ei määritellä käännöksen aikana vaan ohjelman suorituksen aikana ja ne viittaavat yhteen metodiin. Delegaatteja käytetään yleisesti asynkronisissa toiminnoissa ja kun pitää saada yksilöllinen toiminta asiakasluokkaan. Delegaatteja voidaan käyttää monissa tarkoituksissa, esimerkiksi takaisinkutsumetodeissa, staattisten metodien määrittelyssä tapahtumien määrittelyssä.